ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Projet de Bachelor

Rocket Science

Brunner Eric
Burkhard Jonathan
Eckstein Maxime
Mingard Emilien
Rey Julien

21 avril 2018

Table des matières

I Généralités **2**

1 Introduction **3**

2 Cahier des charges **4**

II Simulation 2D **7**

1 Objectifs **8**

2 Manuel d'utilisation **9**
2.1 Capacités du programme 9
2.2 Limitations 9
2.3 Configurations gérées 10
2.4 Définition des entrées 10
2.5 Objet : Fusée 10
2.5.1 Exemple d'un script fonctionnel 15
2.6 Objet : Simulation 16
2.6.1 Exemple définissant les paramètres de la simulation 17
2.7 Définition des sorties 18
2.8 Exécution de la simulation 21
2.8.1 Exemple d'une classe principale 21
2.9 Messages d'erreur 22

3 Implémentation **23**
3.1 Inerties 23
3.1.1 Théorie 23
3.1.2 Modèles 24
3.2 Aérodynamique 28
3.3 Moteur 31
3.4 Atmosphère 32
3.5 Equation d'état 33
3.5.1 Phase 1 : Lancement le long du rail 35
3.5.2 Phase 2 : Vol avec poussée 37
3.5.3 Phase 3 : Vol sans poussée jusqu'à l'apogée 39
3.6 Structure du code 40

4 Validation **43**
4.1 Cas 1 : MadCow Super DX3 43
4.1.1 Comparaison avec les mesures 43
4.1.2 Comparaison avec *OpenRocket* 44
4.1.3 Comparaison avec la Simulation 1D 44
4.2 Cas 2 : Fusée type pour IREC 47
4.3 Discussion 47

III Analyse structurelle **49**

1 Cas de charge **50**
1.1 Introduction 50
1.2 Flambage dû à la poussée 50
1.3 Déploiement du parachute 52
1.3.1 Energie 52
1.3.2 Drag force 53
1.4 Atterrissage 54
1.5 Charge latérale 54
1.5.1 Correction aérodynamique 54
1.5.2 Transport 55

2 Analyse par éléments finis **59**
2.1 Simulation du tube 59
2.1.1 Présentation 59
2.1.2 Outils 60
2.1.3 Protocole 60
2.1.4 Paramètres utilisateurs 61
2.1.5 Limitations 61
2.1.6 Éléments à ajouter 62
2.1.7 Conclusion & Exemple 62
2.2 Analyse des coupleurs 64
2.2.1 Présentation 64
2.2.2 Optimisation de la taille 65
2.2.3 Assemblage 68
2.2.4 Résultats et conclusions 69

IV Expérimentation **71**

1 Introduction **72**
1.1 Pourquoi 72
1.2 Objectifs 72

2 Matériel & set-up **74**
2.1 Capteurs 74
2.1.1 Accéléromètre & Boussole 74

2.1.2 Gyroscope . . . 75
2.1.3 Baromètre . . . 75
2.2 Microcontrôleur . . . 76
2.3 Circuit . . . 77
2.3.1 Protocole de communication . . . 77
2.3.2 Branchements . . . 78
2.4 Programmation . . . 79
2.4.1 Arduino & Teensyduino IDE . . . 81
2.4.2 MATLAB . . . 82

3 Résultats **85**
3.1 Lancement 1, 1 Avril 2017 . . . 85
3.1.1 Fusées . . . 85
3.1.2 Moteurs . . . 85
3.1.3 Altitude . . . 86
3.1.4 Accélération . . . 87
3.1.5 Champs magnétiques . . . 88
3.2 Lancement 2, 20 Mai 2017 . . . 89
3.2.1 Fusées . . . 89
3.2.2 Moteurs . . . 91
3.2.3 Altitude . . . 91
3.2.4 Accélération . . . 92
3.3 Conclusion module de mesure . . . 93

V Divers et conclusion **95**

1 Activités annexes **96**
1.1 Journées de lancement . . . 96
1.2 Meeting Ruag . . . 97
1.3 Discussion aéro-freins . . . 97

2 Conclusion **98**

VI Annexes **99**

A Formules aéro **100**
A.1 Introduction . . . 100
A.2 Variables and abreviations . . . 100
A.3 Subsonic flow, basic revolution solids . . . 102
A.3.1 C_{N_α} of body components . . . 102
A.3.2 C_{m_α} of body components . . . 102
A.3.3 Effect of body lift . . . 102
A.3.4 Center of pressure of body components . . . 103
A.3.5 Planar fins . . . 103
A.3.6 Roll dynamics . . . 105

A.3.7 Drag forces 106

B MATLAB (classes et méthodes principales) 109
B.1 MainTest.m 109
B.2 Rocket.m 109
B.3 Simulate.m 122
B.4 DrawGraph.m 128

C Code source Teensy 132

D Script Python pour Abaqus 136

E Divers acquis 143

F Collaboration multi-projets 151

G Certification niveau 1 158

H Certification niveau 2 166

I Posters 174

Remerciements

Au Docteur Joël Cugnoni pour nous avoir suivi tout le semestre.
Au Docteur Nassima Nasri pour sa précieuse aide en matière de colle et composites.
Au laboratoire LMAF pour nous avoir prêté un espace de travail et proposé une aide financière pour les expériences.

Première partie

Généralités

Chapitre 1

Introduction

Ce projet de Bachelor propose une incursion dans le domaine de la fuséologie. Les modèles de fusées étudiés dans de ce document sont des plus simples et dépourvus de tout système de guidage. Il faut par conséquent prédire leur comportement avant le lancement et s'assurer que le vol ne mènera pas à l'échec.

Pour y parvenir, nous nous proposons de décrypter les fusées en commençant par les bases. Premièrement, nous nous intéresserons à leurs géométries et aux effets de celles-ci. En effet, les forces aérodynamiques qui en découlent leur confèrent certaines propriétés permettant d'obtenir une stabilité primordiale.

Ensuite, en résolvant les équations de la physique classique, nous pouvons décrire le comportement de la fusée. Celles-ci incluent notamment tous les effets non linéaires tels que les forces aérodynamiques, la poussée non constante et le changement des caractéristiques inertielles.

Finalement, pour garantir l'intégrité de la fusée, nous procédons à une analyse structurelle des éléments majeurs qui la composent. Nous détaillons plus particulièrement les tubes, constituant le corps principal, et les coupleurs, éléments liant les différentes parties tubulaires. Nous souhaitons vérifier que la fusée ne cède pas lors d'événements critiques, tels que le transport, la poussée initiale ou l'ouverture du parachute.

Objectifs

Le premier but de ce projet est la mise en place d'un code de simulation simple adapté aux vols suborbitaux. Cette tâche inclut la création d'un modèle paramétrique décrivant au mieux la géométrie d'une fusée ainsi que les conditions de vol afin de pouvoir simuler une mission.

Une deuxième phase est dédiée au calcul des critères de pré-design structurels minimaux qui permettent d'assurer de l'intégrité de la fusée lors de toutes les phases du vol. Ces calculs nous permettront également d'évaluer la solidité des pièces de liaison entre des sections tubulaires, dont nous avons imaginé le design.

Pour finir, nous désirons vérifier la fiabilité de notre modèle théorique ainsi que de nos simulations. Cette validation passe par la construction d'un module de mesure embarqué et par le lancement de fusées prototypes.

Chapitre 2

Cahier des charges

Cahier des charges

Projet: Matterhorn
Système : Aerostructure
Version document : 1.0
Dernière mise à jour : 14.05.2017
Responsabilité du document : Equipe d'aérostructure

Notes:
FP: Primary Function
SI: System integration
FC: Constraint Function
IREC: Given by Rules

ID	Fonction	Critère	Objectif	Source	Fléxibilité	Méthode de vérification	Commentaires
FP1	Stabilité statique	Marge de stabilité	Entre 1 à 2 diamètres de fusée	IREC		Modélisation du CM et du CP pendant le vol	
FP2	Stabilité dynamique			IREC		Simulation	
FP3	Accessibilité aux composants interne	Temps	Max 5 min pour ouvrir/fermer	SI			
FP4	Tenue des sous-systèmes	Bagues d'adaptations		SI			
FP5	Atteindre l'apogée	Altitude	3048 m	IREC	0 à 300 m	Simulation et test de vol	Objectif du concours
FC1	Otimisation de la masse	Masse	Max 30 kg	SI		Catia	
FC2	Optimisation de la taille	Volume	Max possible	SI		Caita	
FC2	Optimisation de placement	Position des capteurs		SI		Design et discussion	
FC3		Contrainte dans la structure		SI		ABAQUS	
FC4	Optimisation de placement	Position des sous-systèmes		SI		Assemblage sur Catia	

Cahier des charges

FC5		Contraintes max dues à la température		SI		ABAQUS	
FC6	Optimisation de placement	Positions des actuateurs		SI		Assaemblage sur Catia	
FC7		Coupleurs		SI		ABAQUS	
FC8	Contraintes de coûts	Budget		SI		Tableau excel	
FC9		Ventilation adéquate	Perçage d'un trou entre 3.2 - 4.8 mm de diamètre dans la partie du booster, nose cone ou payload	IREC			Prévenir une pression interne non voulue
FC10		L'intégrité structurelle	La fusée peut résister aux contraintes de vol et manipulations	IREC		Calcul et ABAQUS	Manipulation : transport
FC11		Restrictions matériels	PVC n'est pas permis dans aucune structure supportant une charge	IREC			
FC12		Oeillet forgé	Tous oeillet sous charges doit être	IREC			
FC13		Implémentation de couplage entre les tubes	Couplage doit être plus petit qu'un calibre	IREC			
FC14	Fixation de la fusée au rail	Guide de lancement		IREC			Les guides de lancement et leur fixation à la structure principale doit être suffisamment fort pour être capable de soulever la fusée verticalement à partir du guide inférieure

FC15		Marques d'identifications	L'ID doit être visible sur les 4 quadrant de la fusée	IREC			
FC16	Fusée doit rester sub-sonic	Vitesse max M < 0.7	Projet		Dimension du moteur	Simulation	Hypothèse : Fluide incompressible => Théorie des flux potentiels
FC17		Angle d'ataque	Max +/- 5° (lacets)	Projet		Simulation	Pour rester dans le théorème des flux potentiel
FC18		Maximum impulsion du moteur		SI		Simulation	
FC19		Maximum de taille parachute		SI		Simulation	
FC20		Vitesse en sortie du rail	>100ft/s (30.5m/s)	IREC	Doit être prouver pour les vitesses plus faible	Simulation et test de vol	ESRA Fournit à 18ft rail de lancement. Soit 5,5 m de longueur
FC21		Parachute doit être ouvert même dans le pire cas de scénario	Pire cas : 46 m/s	IREC		Design et discussion	
FC22		Avoir un système d'aérofreins	Ne déstabilise pas	IREC + Projet			

Deuxième partie

Simulation 2D

Chapitre 1

Objectifs

Le principal objectif de la simulation est de mettre en place des outils pour sélectionner et valider les options de design pour la fusée.

Plus particulièrement :

1. Etude de stabilité statique (CM et CP en fonction du temps et de l'angle d'attaque).
2. Prédiction de l'apogée, des accélérations, du nombre de Mach et autres variables de vol.
3. Étude de la stabilité dynamique (oscillations contrôlées, raisonnables, robustesse face à une perturbation externe comme le vent,...).

Pour répondre à ces questions, nous avons implémenté un code en 2 dimensions. Il utilise un axe horizontal dans la direction du vent (x) ainsi qu'un axe vertical représentant l'altitude (z). Par souci de flexibilité, requis pour des outils de prédesign, le code s'articule autour d'une fusée type dont chaque élément constitutif est totalement paramétrable. Ces éléments possèdent tous des caractéristiques géométriques comme la taille ou le positionnement ainsi que des attributs inertiels comme la masse ou la densité. Notons encore que certains de ces attributs peuvent évoluer dans le temps. Ceci permet de calculer les efforts en jeu, de vérifier certains dimensionnements et d'optimiser les différents éléments constitutifs du prototype.

Concernant le langage, nous avons choisi de travailler avec le logiciel MATLAB surtout pour sa disponibilité. Ce choix se voit renforcé par le fait que ce travail a également pour but de jeter quelques bases pour de futurs projets associatifs de l'EPFL (compétitions de fusées), et donc que ce code soit utilisé par d'autres étudiants ayant accès aux mêmes logiciels.

Chapitre 2

Manuel d'utilisation

2.1 Capacités du programme

La simulation calcule l'altitude que peut atteindre une fusée que l'on définit dans le code. La simulation commence à t = 0 lors de l'allumage du moteur et fini au moment où elle atteint l'apogée.

Voici une liste non exhaustive de ce que le programme peut calculer :

— l'angle de la fusée en tout temps
— l'angle d'attaque de la fusée en tout temps
— la position en x et en z de la fusée en tout temps
— la vitesse de la fusée en tout temps

Pour lancer une simulation, il y a deux objets à définir

1. Une fusée
2. Les paramètres de la simulation

Ces deux objets permettent de créer l'ensemble des paramètres nécessaires au lancement de la simulation.

Le premier objet contient tous les paramètres définissant la géométrie et la répartition des masses de la fusée.

Le second objet contient la longueur de la rampe de lancement, l'angle par rapport à la verticale, la vitesse du vent au sol, le temps maximum de la simulation, etc. Il contient aussi un mode de sélection des graphes que nous souhaitons afficher afin de pouvoir les analyser par la suite.

2.2 Limitations

La principale limitation de la simulation est qu'elle couvre uniquement la phase ascendante du vol, c'est à dire la phase de poussée ainsi que le vol balistique jusqu'à l'apogée. Les limitations du programme sont :

— La vitesse de la fusée doit rester subsonique (M $<$ 1)
— La vitesse du vent est constante et dans la direction x par rapport au repère de la fusée
— Le vent ne doit pas dépasser 25 $\frac{m}{s}$
— l'angle du rail est limité entre $[-\frac{\pi}{2}, \frac{\pi}{2}]$

2.3 Configurations gérées

Plusieurs types de composants sont à disposition pour modéliser la fusée à simuler. Certains de ces composants prédéfinis peuvent être multiples :

— Tubes
— Masses ponctuelles
— Parachutes
— Coupleurs
— Cylindres

Alors que d'autres sont uniques :

— Cône
— Queue
— Un jeu d'ailerons
— Moteur

Le nombre d'ailerons est choisi entre 1 et 8. La fusée peut avoir deux types de moteurs :

1. Moteur solide avec une consommation radiale
2. Moteur avec un liquide ou un gaz liquéfié se vidant par le bas

Les configurations qui ne sont pas gérées sont les fusées à étages multiples et les fusées avec des moteurs extérieurs de type booster.

2.4 Définition des entrées

Le code a besoin d'avoir les entrées suivantes afin de pouvoir lancer la simulation :

— Une courbe de poussée sous forme *.mat*
— Les données géométriques afin de créer la fusée
— Les paramètres pour la simulation (longueur de la rampe, etc.)

2.5 Objet : Fusée

Afin de définir la fusée utile pour la simulation, il faut commencer par ouvrir un script et lui donner le nom souhaité (par exemple : fusée_EPFL).

Le script est structuré en 4 parties successives :

1. La définition des propriétés de densité massique ρ pour chacun des matériaux utilisés dans chacun des éléments constituant la fusée.
2. Ajouter la courbe du moteur choisi
3. Il faut ensuite créer l'objet $Rocket()$ afin de pouvoir y ajouter chacun des composants qui la constituent (par exemple : un tube).
4. Il faut maintenant ajouter chacun des éléments de la fusée. Il n'y a pas d'ordre de placement des éléments.

Il faut définir une courbe de poussée du moteur. Pour créer la courbe de poussée, il faut charger les données depuis un fichier *.mat* avec $load(arg1)$ que l'on peut trouver sur le site du fournisseur du moteur.

```
data = load('AeroTech_K270W');
```

Afin d'avoir une courbe qu'avec des nombres et éviter les erreurs du type (NaN), l'utilisation de la fonction $cleanThrustCurve(arg1)$ est fortement conseillée. L'argument $arg1$ est la courbe stockée sous forme $.data$.

```
thrustCurve = cleanThrustCurve(data.data);
```

Pour créer un objet, il faut appeler les méthodes contenues dans le tableau 2.1.

Par exemple pour créer un nez à la fusée, on appelle $nose(L, D, e, rho)$ précédée du nom de l'objet fusée : i.e.

```
R.nose(0.5, 0.15, 2, 2600);
```

Et ainsi de suite pour chacun des éléments. Rappelons que certains objets peuvent être appelés plusieurs fois afin de rendre plus réaliste la simulation.

En fin de script, il y a la possibilité de visualiser certains paramètres de la fusée pendant la simulation en appelant la méthode $printSpecs()$.

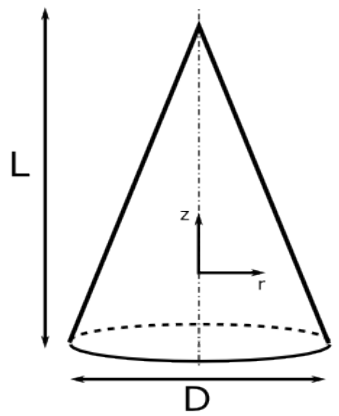
Voici un script vide afin de pouvoir remplir facilement les différents composants de la fusée.

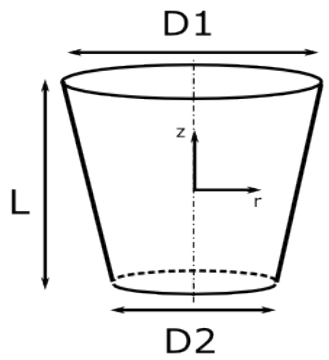
```
function R = Rocket_nom()

    % Parametre
    % Densite

    % Moteur : Courbe a importer


    % Creation de la fusee
    R = Rocket();
    % nose(L, D, e, rho)

    % stage(id, z, L, Dout, e, rho)

    % tail(D1, D2, L, e, z, rho)

    % motor(id, z, D, L, e, m, mp, rho, thrustCurve, bt)

    % payload(id, z, m, L, D)

    % parachute(id, z, L, m, D, Cd)

    % fins(z, N, Ct, Cr, xt, S, r, e, rho);

    % point(id, z, L, m)

    % coupler(obj, id, z, L, Dout, e, rho)

    % Parametre additionel

    % Facteur de correction de lift
    R.k = 1.1; % Par defaut

```

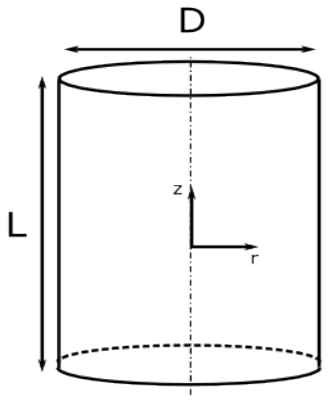
```
        % Cofficient de drag
        R.CD = 0.55;

        % Control que tout les arguments sont bien defini
        R.check();

        % Affiche certain parametre dans le terminal
        R.printSpecs();

    end
```

Dans le tableau 2.1, chacune des méthodes utiles pour définir la fusée est explicitée avec un schéma explicatif de la géométrie. Ces arguments sont importants afin de pouvoir calculer toutes valeurs nécessaires à la simulation numérique.

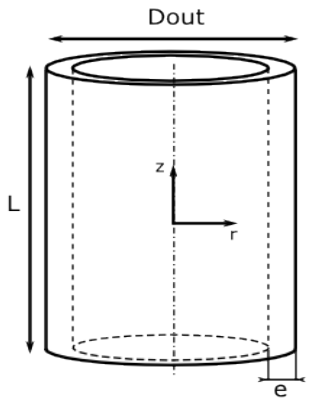
Table 2.1 – Méthodes de la classe Rocket

	`nose(L, D, e, rho)`
	
	`Calcule la masse, le centre de masse, les moments d'inertie, le centre de pression et le coefficient aérodynamique du cÃ´ne.`
	
	`Prend en entrée L la longueur du cÃ´ne, D le diamètre de base, e l'épaisseur, rho [kg/m^3] la densité de sa matière.`

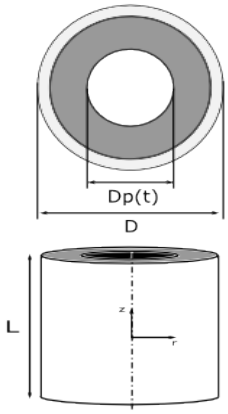
	`tail(D1, D2, L, e, z, rho)`
	
	`Calcule la masse, le centre de masse, les moments d'inertie, le centre de pression et le coefficient aérodynamique de la queue.`
	
	`Prend en entrée D1 le diamètre avant de la queue , D2 le diamètre arrière de la queue, L la longueur de l'élément, e l'épaisseur, z la distance entre la pointe de la fusée et le haut de l'élément et rho la densité de sa matière.`



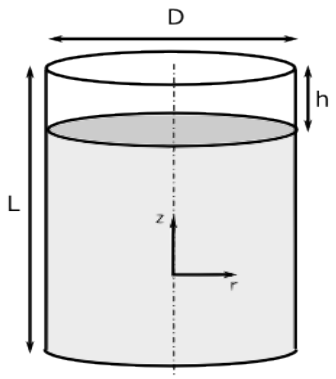

```
stage(id, z, L, Dout, e, rho)
Calcule la masse, le centre de masse, les moments
    d'inertie, le centre de pression et le
    coefficient aérodynamique d'un élément de tube
    .

Prend en entrée z la distance entre la pointe de
    la fusée et le haut de l'élément, L la
    longueur de l'élément, Dout le diamètre
    externe, e l'épaisseur et rho la densité de sa
    matière.
```



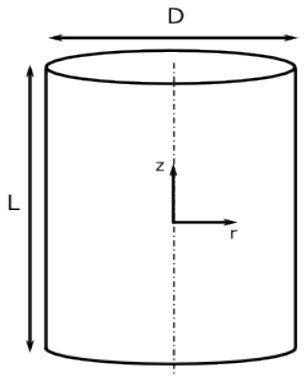

```
coupler(id, z, L, Dout, e, rho)
Calcule la masse, le centre de masse et les
    moments d'inertie d'un élément de couplage
    cylindrique.

Prend en entrée un id (string), z la distance
    entre la pointe de la fusée et le haut de l'él
    ément, L la longueur de l'élément, Dout le
    diamètre externe, e l'épaisseur et rho la
    densité de sa matière.
```



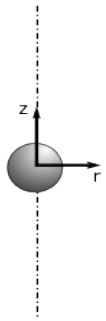

```
motor(id, z, D, L, e, m, mp, rho, thrustCurve, bt
    ) (id = 'motor')
Calcule la masse, le centre de masse et les
    moments d'inertie d'un cylindre de masse qui
    se consume depuis son centre.

Prend en entrée un id (string), z la distance
    entre la pointe et le haut de l'élément, L la
    longueur de l'élément, D le diamètre externe,
    e l'épaisseur du tube moteur, m la masse
    initiale de l'élément, mp la masse de
    combustible qui aura brÃ»lé à la fin du temps
     de combustion bt, rho la densité du
    combustible (SRB =~1772 kg/m^3) et les données
     de la courbe de poussée, thrustCurve.
```



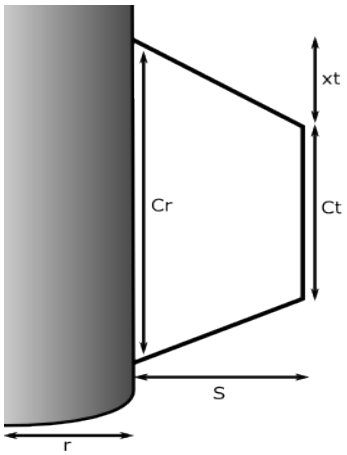

```
motor(id, z, D, L, e, m, mp, rho, thrustCurve, bt
    ) (id = 'tank')
Calcule la masse, le centre de masse et les
    moments d'inertie d'un cylindre de masse qui
    se consume depuis son centre.

Les entrées sont les mêmes que pour le cas où l'
    id = 'motor'.
```



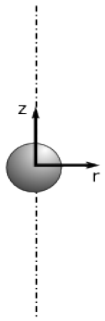

```
cylinder(id, z, m, L, D)
Calcule la masse, le centre de masse et les
    moments d'inertie d'un cylindre plein.

Prend en entrée un id (string), z la distance
    entre la pointe et le haut de l'élément, L la
    longueur de l'élément, D le diamètre externe
    et m la masse de l'élément.
```




```
parachute(id, z, L, m, D, Cd)
Calcule la masse, le centre de masse et les
    moments d'inertie d'un parachute plié. Permets
     également de stocker le diamètre du parachute
     et son coefficient de traÃ®née.

Prend en entrée un id (string), z la distance
    entre la pointe et le haut de l'élément, L la
    longueur de l'élément, m la masse de l'élément
    , D le diamètre effectif du parachute ouvert
    et Cd le coefficient de traÃ®née.
```




```
fins(z, N, Ct, Cr, xt, S, r, e, rho)
Calcule la masse, le centre de masse, les moments
    d'inertie, le centre de pression et le
    coefficient aérodynamique d'un aileron.

Prend en entrée z la distance entre la pointe et
    le haut de l'élément, N le nombre d'ailerons,
    Ct la longueur de corde à la pointe de l'
    aileron, Cr la longueur de corde à sa base, xt
    la longueur de balayage, S la portée, r, le
    rayon moyen du corps de la fusée au niveau de
    la base de l'aileron, e l'épaisseur de l'
    aileron et rho la densité de la matière.
```




```
point(id, z, L, m)
Calcule la masse, le centre de masse et les
    moments d'inertie d'un point de masse.

Prend en entrée un id (string), z la distance
    entre la pointe et le haut de l'élément, L la
    longueur de l'élément et m la masse de l'élé
    ment. La longueur L est requise pour les
    calculs de moments de flexion.
```

2.5.1 Exemple d'un script fonctionnel

```
    function R = RocketJuju()

        % parameters
        rho_polystyr    = 1050; % kg/m^3
        rho_kraft       = 950;
        rho_birch       = 630;

        % Motor
        data = load('AeroTech_H148R.mat');
        thrustCurve = cleanThrustCurve(data.data);
        bt = 1.5; % burn time

        % Create a rocket
        R = Rocket();
        % nose(L, D, e, rho)
        R.nose(0.42, 0.102, 0.0035, rho_polystyr);
```

```
        % stage(id, z, L, Dout, e, rho)
        R.stage('tube', 0.42, 1.24, 0.102, 0.0035, rho_kraft);
        % tail(D1, D2, L, e, z, rho)
        R.tail(0.102, 0.101, 0.01, 0.0035, 1.66, 0);
        % motor(id, z, D, L, e, m, mp, rho, thrustCurve, bt)
        R.motor('motor', 1.47, 0.038, 0.17, 1.5e-3, 0.309, 0.148, 1772,
    thrustCurve, bt);
        % payload(id, z, m, L, D)
        R.cylinder('main payload', 0.52, 0.4, 0.3, 0.102);
        % parachute(id, z, L, m, D, Cd)
        R.parachute('main', 1.12, 0.14, 0.050, 1, 1.5);
        % fins(z, N, Ct, Cr, xt, S, r, e, rho);
        R.fins(1.35, 3, 0.086, 0.3, 0.166, 0.096, 0.051, 0.008, rho_birch)
    ;
        % point(id, z, L, m)
        R.point('trim_nose', 0.3, 0.1, 0.3);
        % coupler(obj, id, z, L, Dout, e, rho)
        R.coupler('coupler', 0.81, 0.2, 0.098, 0.003, rho_kraft);
        R.coupler('nose shoulder', 0.42, 0.1, 0.098, 0.003, rho_polystyr)
        R.coupler('motor tube', 1.26, 0.4, 0.042, 0.003, rho_kraft);

        % Additional parameters:
        % lifting bodys correction factor
        R.k = 1.1; % default
        % Drag coefficient
        R.CD = 0.55;

        % Check definition
        R.check();

        % Print specs
        R.printSpecs();

    end
```

2.6 Objet : Simulation

Il nous faut encore définir l'objet *Simulation* afin de pouvoir lancer le calcul. Cet objet comprend les paramètres suivants :

1. Longueur de la rampe de lancement
2. L'angle du rail par rapport à la verticale
3. La vitesse du vent au sol
4. Le temps maximum de la simulation
5. Un vecteur de temps (utile pour le calcul des efforts en flexion)
6. Un nombre de points de mesure sur la longueur de la fusée pour le calcul du moment de flexion
7. Une sélection des graphes à générer

Il faut commencer par créer un objet $Simulation(arg1, arg2, arg3, arg4, arg5, arg6)$. Par exemple comme ceci :

```
S = Simulation(4, pi/6, 10, 30, [1 3 5 7 8], 10);
```

Où les arguments de la fonction sont dans le même ordre qu'énoncé ci-dessus.

Il y a 9 graphiques visualisables présentés dans le tableau 2.2

Nom	Abscisse	Ordonnée	Graphe
mass	Temps [s]	Masse [Kg], Centre de masse [m], Inertie [$Kg \cdot m^2$], Force de propulsion [N]	2.1a
aero	Angle d'attaque [°]	C_n, position du centre de pression [m]	2.1b
rocket	Longueur [m]	Largeur [m]	2.1c
altitude	Temps [s]	Altitude [m]	2.2
trajectory	Distance x [m]	Altitude [m]	2.3
verticalAngle	Temps [s]	Angle par rapport à la verticale [°]	2.4b
attackAngle	Temps [s]	Angle d'attaque [°]	2.4a
stability	Temps [s]	Calibre [-]	2.4c
flexion	Distance par rapport au cône [m]	Moment de flexion [Nm]	2.5

TABLE 2.2 – Graphes à choisir

Pour afficher un graphique, il faut appeler la fonction $setGraph(arg1, \quad arg2)$ comme suit :

```
S.setGraph('mass', 1);
```

où S est l'objet simulation, le premier argument est le nom du graphe et le second prend 1 si on souhaite l'afficher et 0 sinon.

Pour afficher l'environnement créé, il faut appeler la fonction $printSpec()$:

```
S.printSpecs();
```

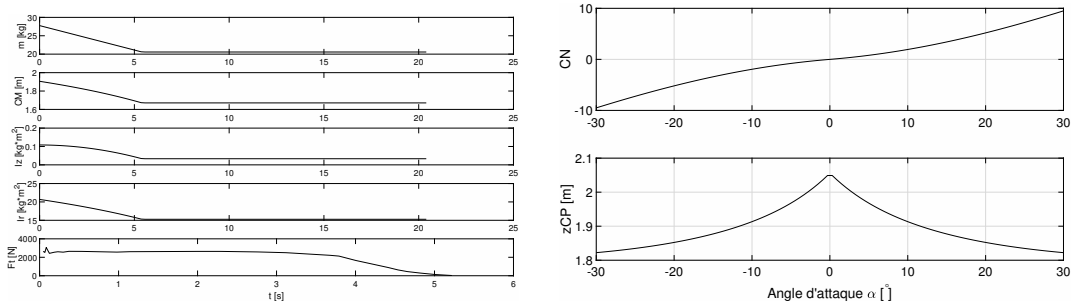
2.6.1 Exemple définissant les paramètres de la simulation

```
function S = SimulationZurich()
% Environment de Zurich
% INPUT
%   - R     : Rocket
%   - v_vent: un vent positif va de gauche a droite donc dans les x positifs
%                       negatif va de droite a gauche

```

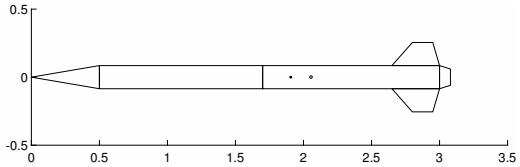
```
        % Iinitalisation des variables
        tfin = 11;
        phi0 = 0;
        v_vent = 10;
        tquer = [1 3 5 7 8];
        nquer = 10;
        L_ramp = 2;

        %initialisation de l environement
        S = Simulation(L_ramp, phi0, v_vent, tfin, tquer, nquer);

        % Choix des graphes
        S.setGraph('mass', 1);
        S.setGraph('aero', 1);
        S.setGraph('rocket', 1);
        S.setGraph('altitude', 1);
        S.setGraph('trajectory', 1);
        S.setGraph('verticalAngle', 1);
        S.setGraph('attackAngle', 1);
        S.setGraph('stability', 1);
        % S.setGraph('flexion', 1);

        % Afficher environment
        S.printSpecs();

    end
```

2.7 Définition des sorties

Les sorties, ou résultats de cette simulation sont présentés dans cette section : les valeurs utilisées pour illustrer sont celles de la fusée que nous avons pré dimensionné pour le concours IREC. Les sorties de notre simulation sont principalement des graphes. Ceux-ci permettent de visualiser facilement l'évolution de la fusée pendant la durée du vol simulé. La figure 2.1 montre l'influence des paramètres massiques et géométriques au cours du temps. Ces résultats sont essentiels pour comprendre ce qui se passe lors de la combustion du moteur. Les graphiques 2.2 et 2.3 permettent de connaître et de visualiser la trajectoire de la fusée au cours du temps ainsi que le déplacement en x en fonction de l'altitude.



(a) Propriétés massique (b) Propriétés aéro à M = 0 et $\theta = 0$



(c) Représentation de la fusée

FIGURE 2.1 – Résultat de la simulation sur les données géométriques, massique

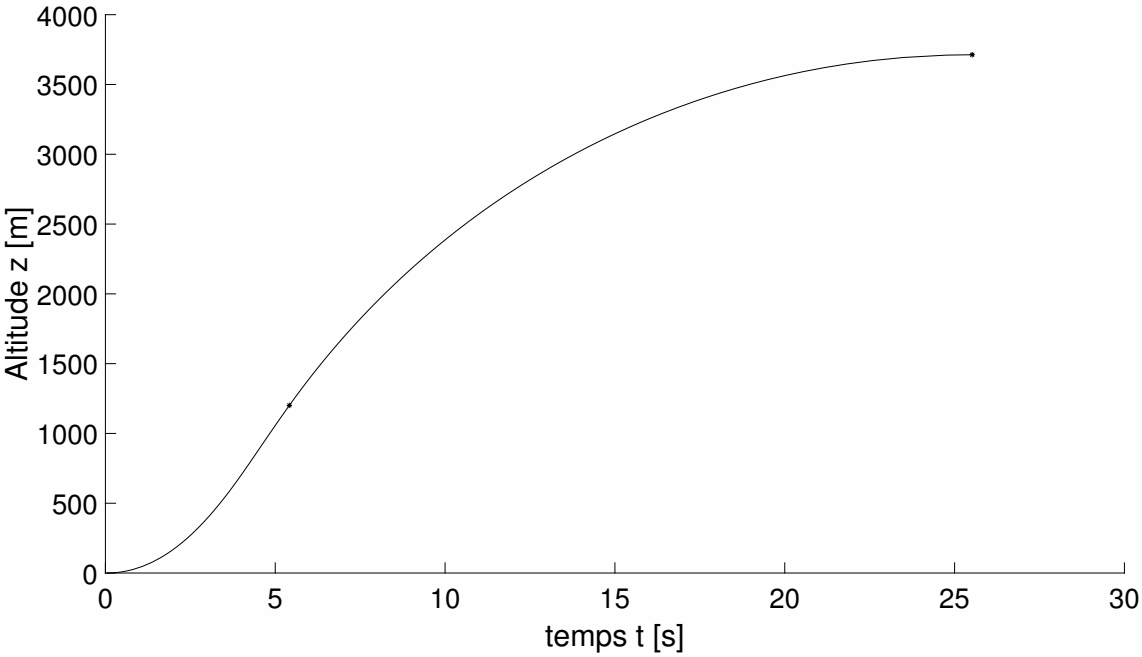


FIGURE 2.2 – Trajectoire de la fusée par rapport au temps

Dans la figure 2.4, nous pouvons voir trois graphiques très intéressants qui permettent d'observer au cours du temps les paramètres suivants : l'évolution de l'angle d'attaque (2.4a), la rotation de la fusée par rapport à son centre de masse (2.4b) et la stabilité statique en vol (2.4c).

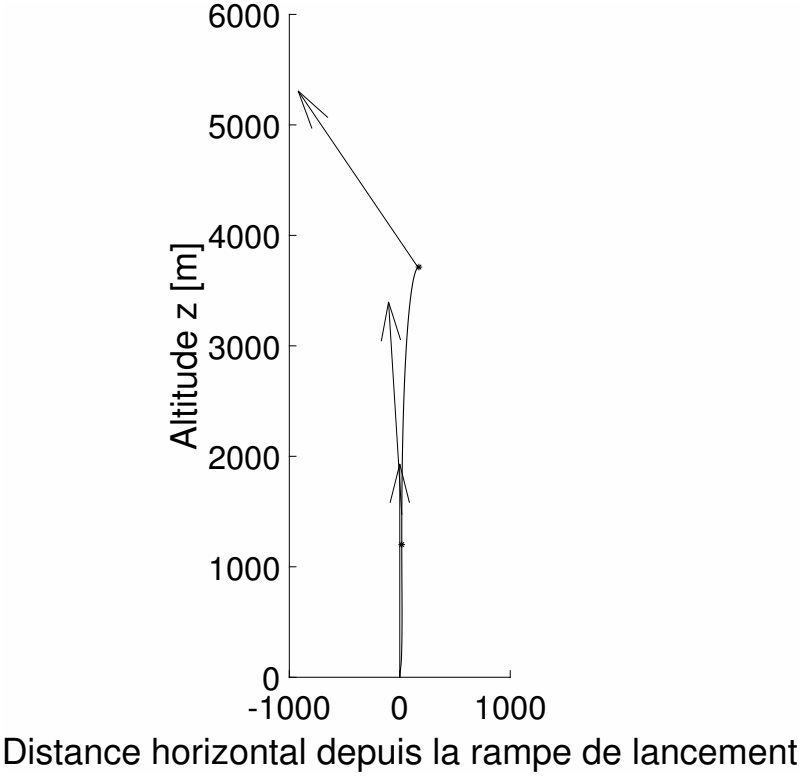
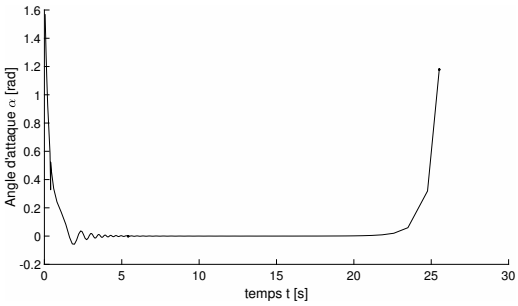
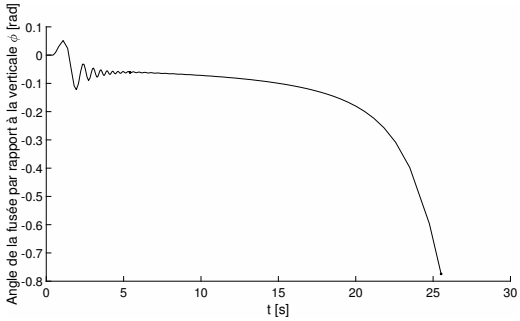


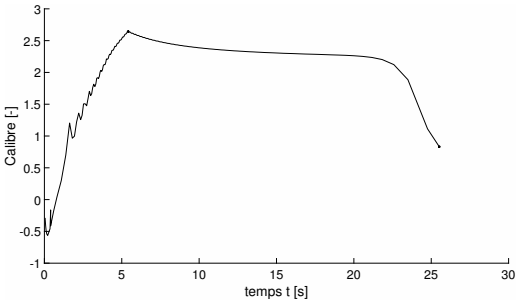
FIGURE 2.3 – Déplacement latéral et orientation de la fusée



(a) Angle d'attaque



(b) Angle de la fusée par rapport à la verticale



(c) Stabilité statique au court du temps

FIGURE 2.4 – Résultat de la simulation pour la fusée IREC

Le dernier graphe 2.5 représente le moment de flexion calculé le long de la fusée.

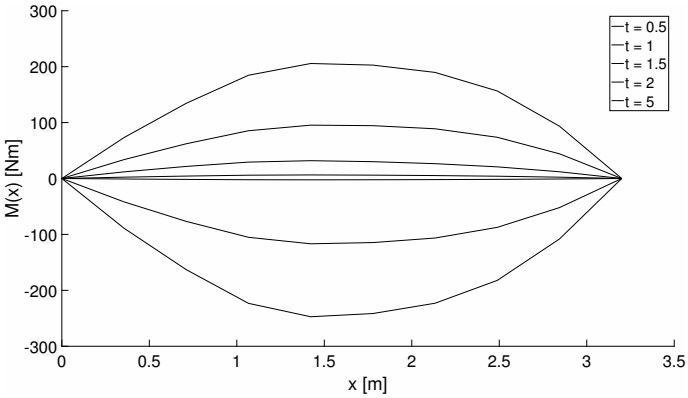


FIGURE 2.5 – Moment de Flexion

Les autres données de sorties disponibles sont dans la structure *results* qui contient les matrices et vecteurs suivants :

— Étape de temps tSim
— Matrice contenant les variables d'états Xsim (x, z, v_x, v_z, ϕ, $\dot{\phi}$)
— L'angle d'attaque α
— Le calibre
— La structure flexion
— Le vecteur de position xquer des endroits où le moment de flexion est calculé
— Une structure accélération accel contenant le vecteur d'accélération en x et z

Afin de pouvoir utiliser ces données, il faut entrer dans le terminal de Matlab la commande suivante : par exemple pour afficher le vecteur de temps :

```
    results.tSim
```

Cela peut être effectué pour chacun des différents éléments de la structure *results*.

2.8 Exécution de la simulation

Une fois que les différentes étapes ci-dessous ont été effectuées, il faut ouvrir la classe principale : mainTest ou en créer une, afin de lancer la simulation. Ce fichier s'écrit comme dans la sous-section 2.8.1.

1. Avoir une courbe de poussée
2. Créer l'objet *Rocket*()
3. Créer l'objet *Simulation*()

Ces différents fichiers doivent être situés dans le même dossier que celui de la classe principale.

2.8.1 Exemple d'une classe principale

```
    % Main test for simulation of any rocket and mission

close all;
clear all;
```

```
clc;

% Creation des deux objets de la simulation
R = RocketJuju();
S = SimulationZurich();

% Simulate part
results = Simulate( R, S);

% DrawGraphs
DrawGraph(R, S, results);
```

2.9 Messages d'erreur

La gestion des erreurs est faite par l'affichage de celles-ci directement dans le terminal. La liste des erreurs possibles est dans le tableau 2.6.

#	Type	Classe	Partie	Objet	Message
1	Error	Rocket	Définition	Moteur	Le volume du combustible depasse la place disponible. Solution: reduire la masse de combustible ou augmenter sa densite. Le Diametre interne depasse de XXX m
2	Error	Rocket	Définition	Moteur	Motor element must either be a motor or a tank.
3	Error	Rocket	check()	Cône	Must define a nose cone
4	Warning	Rocket	check()	Etage	No stage defined
5	Warning	Rocket	check()	Queue	No tail defined
6	Warning	Rocket	check()	Ailerons	No fins defined
7	Error	Rocket	check()	Moteur	Must define a motor
8	Warning	Rocket	check()	Cylindre	No Cylinder defined
9	Warning	Rocket	check()	Coupleur	No Coupler defined
10	Warning	Rocket	check()	Parachute	No parachute defined
11	Warning	Rocket	check()	Masse ponctuelle	No point mass defined
12	Error	Rocket	check()	Diamètre d	Reference diameter d is not defined yet or has a value of 0. Define a nose cone with non-zero base diameter.
13	Error	Rocket	cna_fins()	Diamètre d	Reference diameter d is not defined yet or has a value of 0. Define a nose cone with non-zero base diameter
14	Warning	Rocket	cna_fins()	Nombre de Mach	Mach number must be < 1. Code cannot guarantee validity for supersonic speeds.
15	Error	Rocket	cna_fins()	Nombre de fins	The number of fins must be between 1 and 8
16	Error	Simulation	setGraph()	ID Graphe	Id: XX doesn't correspond to an existing plot or more than one plot with the given id exists.
17	Error	Simulation	setGraph()	Graphe	Unexpected number of arguments given to setGraph
18	Error	Simulation	getGraphValue()	Graphe	Id: id doesn't correspond to an existing plot or more than one plot with the given id exists
19	Error	stdAtmos	Global	Altidude maximum	stdAtmos:outOfRange', 'The altitude is out of range: max 10km
20	Warning	cleanThrust Curve	Global	Courbe moteur	Thrust curve time steps are not monotonicaly ascending
21	Error	Flexion	Global	Taille matrice	m must be a column vector
22	Error	Flexion	Global	Taille matrice	l must be a column vector
23	Error	Flexion	Global	Taille matrice	xs must be a column vector
24	Error	Flexion	Global	Taille matrice	x must be a line vector

FIGURE 2.6 – Tableau : Gestion des erreurs

Chapitre 3

Implémentation

3.1 Inerties

3.1.1 Théorie

Pour déterminer la masse des éléments de densité homogène ρ, on procède par intégration sur le volume V :

$$m = \iiint_V \rho dV \tag{3.1}$$

Le centre de masse est positionné le long de l'axe z de l'élément. Sa position est donnée par :

$$z_G = \frac{1}{m} \iiint_V \vec{\mathrm{OP}} \rho dV \tag{3.2}$$

où $\vec{\mathrm{OP}}$ indique la distance entre l'origine du repère de la pièce et la position de l'élément de volume.

Pour les moments d'inertie, on part de la définition : [1]

$$I = \sum_\alpha m_\alpha d_\alpha^2 \tag{3.3}$$

où m_α est un élément de masse dont le centre d'inertie se trouve à une distance d_α de l'axe de rotation.

Pour cette simulation on définit pour chaque élément deux axes de rotation r et z passant par le centre de masse. L'un le long de l'axe de la fusée I_z et l'autre selon l'axe perpendiculaire I_r.

$$I_z = \iiint_V r'^2 \rho dV \tag{3.4}$$

$$I_r = \iiint_V z'^2 \rho dV \tag{3.5}$$

r' représente la distance de l'élément de volume dV à l'axe z et z' la distance de dm à l'axe r. Ceci nous permet de former le tenseur d'inertie,

$$I = \begin{pmatrix} I_r & 0 & 0 \\ 0 & I_r & 0 \\ 0 & 0 & I_z \end{pmatrix} \tag{3.6}$$

qui pourrait s'avérer utile pour des développements futurs du code qui impliqueraient une 3ème dimension.

Pour assembler les éléments d'inertie, on utilise le théorème des axes parallèles dont la forme tensorielle générale est [10] :

$$\mathbf{J}_{ij} = \mathbf{I}_{ij} + m(||\mathbf{r}||^2\delta_{ij} - \mathbf{r}_i\mathbf{r}_j) \tag{3.7}$$

avec $\mathbf{I}$ le tenseur des moments d'inertie autour du centre de masse, $\mathbf{r} = r_1\mathbf{e_1} + r_2\mathbf{e_2} + r_3\mathbf{e_3}$ la distance entre les deux axes. Vu que notre simulation se passe dans le plan, on peut poser $\mathbf{r}_i\mathbf{r}_j = 0$.

3.1.2 Modèles

Les formules utilisées pour le calcul de l'inertie de chaque élément est explicité dans le tableau 3.1. Le démarche pour calculer l'inertie des ailerons est détaillée dans le tableau 3.2. Certaines formules ont été prises de collections existantes [7]. Pour évaluer les propriétés globales de la fusée, nous utilisons les formules spécifiées dans le tableau 3.3.

Table 3.1 – Formules d'inertie

Éléments	Formules
Cone	$V = \frac{1}{2}DLe\pi\sqrt{1+(\frac{D}{2L})^2}$ $m = \rho V$ $z_G = \frac{2}{3}L$ $I_z = \frac{1}{8}\rho V D^2$ $I_r = \frac{1}{4}\rho V((\frac{D}{2})^2 + 2L^2)$
Queue	$V = \frac{1}{2}(D_1+D_2)Le\pi\sqrt{1+(\frac{D_2-D_1}{2L})^2}$ $m = \rho V$ $z_G = L\frac{2D_2+D_1}{3(D_2+D_1)}$ $R_{1,2} = \frac{D_{1,2}}{2}$ $\phi = \frac{R_2-R_1}{L}$ $I_z = \frac{1}{10}\frac{1}{\phi}\pi\rho((R_2^5-R_1^5)-((R_2-e)^5-(R_1-e)^5))$ $I_r = \frac{1}{2}I_z + \pi\rho(\frac{1}{2}\phi L^4 e + \frac{1}{3}L^3(R_1^2-(R_1-e)^2))$

Tube	$V = L\pi((\frac{D_1}{2})^2 - (\frac{D_2}{2})^2)$ $m = \rho V$ $z_G = \frac{1}{2}L$ $I_z = \frac{1}{2}L\pi\rho((\frac{D_1}{2})^4 - (\frac{D_2}{2})^4)$ $I_r = \frac{1}{12}L\pi\rho(3((\frac{D_1}{2})^4 - (\frac{D_2}{2})^4) + L^2((\frac{D_1}{2})^2 - (\frac{D_2}{2})^2))$
Moteur, Réservoir	$m(t) = \begin{cases} (m - (mp/t_b)t) & \text{si } 0 < t < t_b \\ 0 & \text{sinon} \end{cases}$ $\rho_c = \frac{4(m_0 - m_p)}{(L\pi(D^2 - (D-2e)^2))}$ $h(t) = \frac{\rho\pi m(t)}{(D-2e)^2}$ $z_{Gc}(t) = \frac{L}{2}$ $z_{Gp}(t) = L - \frac{1}{2}h(t)\frac{m(t) - m_0 + m_p}{m_p}$ $z_G(t) = \frac{1}{m(t)}(z_{Gc}(t)(m_0 - m_p) + z_{Gp}(t)(m(t) - m_0 + m_p))$ $I_z(t) = \frac{1}{2}\pi\rho_c L((\frac{D}{2})^4 - (\frac{D}{2} - e)^4) + \frac{1}{2}\pi\rho_p h(t)((\frac{D}{2} - e)^4)$ $I_{rc}(t) = \frac{1}{12}L\pi\rho(3((\frac{D}{2})^4 - (\frac{D}{2} - e)^4) + L^2((\frac{D}{2})^2 - (\frac{D}{2} - e)^2))$ $I_{rp}(t) = \frac{1}{12}(m(t) - m_0 + m_p)(3(\frac{D}{2} - e)^2 + h(t)^2)$ $I_r = I_{rp}(t)(z_G(t) - z_{Gp}(t))^2(m(t) - m_0 + m_p)$ $+I_{rc}(t) + (z_G(t) - z_{Gc})^2(m_0 - m_p)$

Moteur, Combustible	$m(t) = \begin{cases} (m-(mp/t_b)t) & \text{si} 0 < t < t_b \\ 0 & \text{sinon} \end{cases}$ $\rho_c = \frac{4(m_0-m_p)}{(L\pi(D^2-(D-2e)^2))}$ $D_i(t) = \sqrt{(D-e)^2 - \frac{4m(t)}{\pi L \rho_p}}$ $z_G(t) = \frac{L}{2}$ $I_z(t) = \frac{1}{2}\pi\rho_c L((\frac{D}{2})^4 - (\frac{D}{2}-e)^4) + \frac{1}{2}\pi\rho_p L((\frac{D}{2})^4 - (\frac{D_i(t)}{2})^4)$ $I_{rc}(t) = \frac{1}{12}L\pi\rho_c(3((\frac{D}{2})^4 - (\frac{D}{2}-e)^4) + L^2((\frac{D}{2})^2 - (\frac{D}{2}-e)^2))$ $I_{rp}(t) = \frac{1}{12}L\pi\rho_c(3((\frac{D}{2})^4 - (\frac{D_i}{2})^4) + L^2((\frac{D}{2})^2 - (\frac{D_i}{2})^2))$ $I_r(t) = I_{rp} + I_{rc}$
Cylindre	$z_G = \frac{1}{2}L$ $I_z = \frac{1}{8}mD^2$ $I_r = \frac{1}{12}m(3(\frac{D}{2})^2 + L^2) + m(\frac{L}{2})^2$
Parachute, Point	$z_G = 0$ $I_z = 0$ $I_r = 0$

Table 3.2 – Formules d'inertie pour les ailerons

Densité surfacique	$\lambda = \rho e$

Calculs sur le triangle 1	$I_{z1} = \frac{1}{36}\lambda x_t S^3$ $I_{r1} = \frac{1}{36}\lambda x_t^3 S$ $A_1 = \frac{1}{2}x_t S$ $z_{G1} = \frac{2}{3}x_t$ $r_{G1} = \frac{1}{3}S$
Calculs sur le rectangle	$I_{z1} = \frac{1}{3}\lambda C_t S^3$ $I_{r1} = \frac{1}{3}\lambda C_t^3 S$ $A_1 = C_t S$ $z_{G1} = \frac{1}{2}C_t$ $r_{G1} = \frac{1}{2}S$
Calculs sur le triangle 2	$x_e = C_r - C_t - x_t$ $I_{z1} = \frac{1}{36}\lambda x_e S^3$ $I_{r1} = \frac{1}{36}\lambda x_e^3 S$ $A_1 = \frac{1}{2}x_e S$ $z_{G1} = \frac{2}{3}x_e$ $r_{G1} = \frac{1}{3}S$
Aire totale	$A_{tot} = \frac{1}{2}(C_r + C_t)S$

Propriétés d'inertie	$m = \lambda A_{tot}$ $z_G = \frac{1}{A_{tot}}(A_1 z_{G1} + A_2(x_t + z_{G2}) + A_3(x_t + C_t + z_{G3}))$ $r_G = \frac{1}{A_{tot}}(A_1 r_{G1} + A_2 r_{G2} + A_3 r_{G3})$ $I_z = I_{z1} + I_{z2} + I_{z3} + \lambda(A_1 r_{G1}^2 + A_2 r_{G2}^2 + A_3 r_{G3}^2)$ $I_r = I_{r1} + I_{r2} + I_{r3} + \lambda(A_1 z_{G1}^2 + A_2(x_t + z_{G2})^2 + A_3(x_t + C_t + z_{G3})^2)$ $I_\theta = I_r + I_z$

TABLE 3.3 – Formules d'inertie Globales

Masse	$m = \sum_i m_i$
Centre de masse	$z_G = \frac{1}{m}\sum_i m_i z_{Gi}$
Moment d'inertie axial	$I_z = \sum_i (I_{zi} + m_i(2r_{Gi}r_i + r_i^2))$
Moment d'inertie radial	$I_r = \sum_i (I_{ri} + m_i(2z_{Gi}z_i + z_i^2))$

Pour compléter le tableau de formules d'inerties globales 3.3, il faut encore ajouter au moment d'inertie perpendiculaire I_r l'effet des ailerons qui sont hors du plan du vent. Pour cela, on ajoute un terme à la somme du moment d'inertie axial :

$$I_r^{\text{Ailerons}} = \cos^2(\phi) I_\theta + \sin^2(\phi) I_r. \tag{3.8}$$

3.2 Aérodynamique

La théorie derrière les formules d'aérodynamique utilisées pour calculer le centre de pression et la force normale se trouve dans le document correspondant du rapport de travail en annexe A et le papier original de James. S. Barrowman [2]. Celles-ci sont résumées dans le tableau 3.4. Pour les ailerons, la démarche est détaillée dans le tableau 3.5. L'implémentation de ceux-ci est inspirée de OpenRocket [9], sans considérer les variations pour le cas supersonique. Un facteur correctif est apporté aux corps de révolution pour prendre en compte les effets de portance négligés par Barrowman[6]. Ceux-ci ont un effet important sur la variation de la position du centre de pression en fonction de l'angle d'attaque. La fonction corrective est

$$CN_{\alpha^2} = 4\frac{K}{\pi}\frac{A_p}{D^2}\alpha \tag{3.9}$$

avec K un facteur entre 1.1 et 1.5, D le diamètre de référence et A_p les aires projetées telles que déclarées dans le tableau 3.6. Les forces aérodynamiques globales sont calculées telles qu'explicitées dans le tableau 3.7.

TABLE 3.4 – Formules d'aérodynamique

Cone	$CP_z = \frac{2}{3}L$ $CN_\alpha = 2\frac{\sin\alpha}{\alpha}$
Queue	$CP_z = \frac{1}{3}L(1 + \frac{1-\frac{D_1}{D_2}}{1-(\frac{D_1}{D_2})^2})$ $CN_\alpha = 2\frac{1}{d^2}(D_2^2 - D_1^2)$
Tube	$CP_z = 0$ $CN_\alpha = 0$

TABLE 3.5 – Formules aérodynamiques pour les ailerons

Aire de référence	$A_{\text{ref}} = \frac{1}{4}\pi d^2$
Paramètre de Mach	$\beta = \sqrt{1 - \mathrm{M}^2}$
Angle de mi-corde	$\gamma_c = \arctan(\frac{1}{S}(\frac{x_t+C_t}{2} - \frac{C_r}{2}))$
Aire de l'aileron	$A = \frac{1}{2}(C_r + C_t)S$
Facteur d'interférence	$K_{TB} = 1 + \frac{r_t}{r_t+S}$
Coefficient aérodynamique unique	$CN_{\alpha 1} = \frac{2\pi\frac{S^2}{A_{\text{ref}}}}{1+\sqrt{1+(\frac{\beta S^2}{A\cos\gamma_c})^2}}$

Coefficient aérodynamique total	$CN_\alpha = \begin{cases} CN_{\alpha 1}\sin(\theta)^2 & N=1 \\ CN_{\alpha 1}(\sin(\theta)^2+\sin(\theta+\pi)^2) & N=2 \\ 1.5CN_{\alpha 1}(1-0.15\cos^2(\frac{3}{2}\theta)) & N=3 \\ 2CN_{\alpha 1}(1-0.06\sin^2(2\theta)) & N=4 \\ 2.37CN_{\alpha 1} & N=5 \\ 2.74CN_{\alpha 1} & N=6 \\ 2.99CN_{\alpha 1} & N=7 \\ 3.24CN_{\alpha 1} & N=8 \end{cases}$

Table 3.6 – Aire projetées et leurs centres géométriques.

Élément	A_p	X_p
Cône	$\frac{1}{2}L_N D$	$\frac{2}{3}L_n$
Queue	$\frac{1}{2}(D_1+D_2)L_T$	$X_{TN}+\frac{1}{3}L_T\frac{(D_1+2D_2)}{(D_1+D_2)}$
Tube	$L_B D_B$	$\frac{1}{2}L_B$

Table 3.7 – Formules pour le calcul des propriétés aérodynamiques globales.

Coefficient aérodynamique normal	$CN_\alpha = \sum_i CN_{\alpha i} + \sum_i CN_{\alpha^2 i}$
Centre de pression	$CP_z = \frac{\sum_i CP_{zi}CN_{\alpha i}+\sum_i X_{pi}CN_{\alpha^2 i}}{CN_\alpha}$
Force normale	$F_N = \frac{1}{2}\rho S CN_\alpha V_i^2$
Force de frottement	$F_D = \frac{1}{2}\rho S C_D V_i^2$

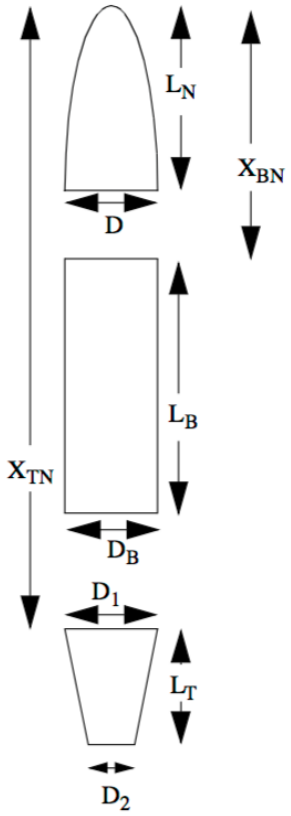


FIGURE 3.1 – Schéma pour le calcul du facteur correctif de portance des éléments de révolution.

3.3 Moteur

La poussée du moteur est modélisée grâce à sa courbe de poussée. Pour des moteurs commerciaux, celle-ci est fournie par le fabriquant. Elle se présente sous la forme d'un couple temps-poussée. Il faut prendre garde aux unités de la poussée étant donné qu'elle peut être fournie en Newton ou en Livres. Une base de données de courbes de moteurs certifiés existe auprès de `http://www.thrustcurve.org`. Les formats de fichier les plus pratiques sont les .eng et .txt . Un exemple de fichier trouvé sur leur site est présenté dans le listing suivant :

```
THRUST
; AeroTech H123W
; converted from TMT test stand data 1999 (www.tripoli.org)
; provided by ThrustCurve.org (www.thrustcurve.org)
; total weight: 279g, propellant weight: 126g
0.00   0.00
0.02   63.48
0.07   133.66
 [...]
2.32   6.03
2.37   4.69
−1.00
```

Dans le cas où aucun de ces fichiers ne sont disponibles, le script MATLAB *getTdat.m* sera utile pour prélever les points de la courbe sur une image de celle-ci. L'image de la courbe de poussée du moteur présenté ci-dessus se trouve sous la forme montrée en figure 3.2. Pour déterminer la valeur de la poussée entre les couples de données, la simulation fait une interpolation linéaire.

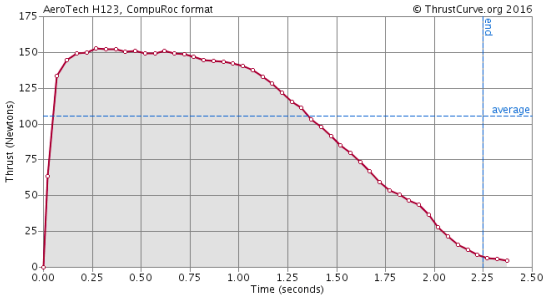


FIGURE 3.2 – Courbe de poussée typique d'un moteur à poudre

3.4 Atmosphère

L'atmosphère standard américaine de 1977 est utilisée [11]. La température varie linéairement de $\frac{dT}{dh} = -6.5°[K/km]$ jusqu'à une altitude de $10[km]$, correspondant à la limite de la troposphère. Le lieu du concours se trouvant à $1400[m]$ d'altitude, ce modèle suffit pour nos besoins. Les conditions initiales au niveau de la mer sont données par l'atmosphère standard :

— $T_0 = 288°K$
— $p_0 = 101'325Pa$
— $\rho_0 = 1.225kg/m^3$

On définit également certaines constantes :

— $r = 287$, la valeur réelle de la constante des gaz parfaits pour l'air.
— $\gamma = 1.4$ le rapport des chaleurs spécifiques pour l'air.
— $g = 9.80665$ l'accélération gravifique (constante).

La température évolue comme :

$$T = T_0 + \frac{dT}{dh} h/1000$$

où h est l'altitude en mètres. En faisant l'hypothèse d'une atmosphère hydrostatique, la pression est :

$$p = p_0(1 + \frac{dT}{dh}\frac{h}{1000T_0})^{(\frac{-1000g}{rdT/dh})}$$

La densité est alors obtenue par l'équation d'état des gaz parfaits :

$$\rho = \frac{p}{rT}$$

Pour calculer le nombre de mach il faut connaître la vitesse du son à chaque altitude :

$$a = \sqrt{\gamma r T}$$

3.5 Equation d'état

Voici un tableau contenant la nomenclature utilisée dans cette section.

Description	Symbole	Unités
Angle d'attaque	α	$[rad]$
Diamètre de la fusée	d	$[m]$
Force de pesanteur de la fusée	F_p	$[N]$
Force de poussée du moteur	F_t	$[N]$
Force du rail sur la fusée	F_{rail}	$[N]$
Force (aérodynamique) normale à la fusée	F_n	$[N]$
Force de trainée	F_D	$[N]$
Masse totale de la fusée	m_{tot}	$[kg]$
Nombre de Mach	Ma	[-]
Vitesse de la fusée	v	$[\frac{m}{s}]$
Vitesse du vent	v_vent	$[\frac{m}{s}]$
Vitesse du vent relatif (incident)	v_i	$[\frac{m}{s}]$
Surface de référence de la fusée	A_{ref}	$[m^2]$

Voici un tableau définissant certaintes constantes utilisées.

Description	Symbole	Valeur	Unités
Accélération gravifique	g	9.81	$[\frac{m}{s^2}]$
Coefficient de traînée	C_d	0.85	[-]

Nous avons trois phases de vol.

1. Phase de décollage le long du rail
2. Phase de vol avec la poussée du moteur
3. Phase de vol sans poussée jusqu'à l'apogée

Ces trois phases nous donnent 3 équations d'états. Celles-ci découlent de la seconde loi de Newton et du théorème du moment cinétique.

$$\sum_{i=1}^{n} \vec{F_i} = \frac{\delta^2(m \cdot \vec{x})}{\delta t^2} \tag{3.10}$$

$$\sum_{i=1}^{n} \vec{M_i} = \frac{\delta^2(J \cdot \vec{\theta})}{\delta t^2} \tag{3.11}$$

Nous allons définir chaque équation et formule utilisée. Commençons par la vitesse du vent incident qui se calcule comme suit

$$\vec{v_i} = \begin{pmatrix} v_{vent} - v_x \\ \\ -v_z \end{pmatrix} \tag{3.12}$$

Où v_x et v_z sont respectivement la vitesse de la fusée en x et z. Les forces dans notre modèle varient au cours du temps. La plupart de nos paramètres dépendent du temps par exemple : la variation de l'angle d'attaque, la variation de la masse lors de la combustion du moteur, du vent. Chacun de ces paramètres est calculé à chaque pas de temps et nous permet d'avoir la force appliquée sur la fusée tout au long de la simulation.

La force de poussée est définie en fonction de la courbe de poussée en fonction du temps décrite à la section 3.3 ainsi que l'angle ϕ de la fusée.

$$\vec{F_t} = Poussée(t) \cdot \begin{pmatrix} sin(\phi) \\ \\ cos(\phi) \end{pmatrix} \tag{3.13}$$

Où Poussée prend les valeurs de la courbe de poussée en $[N]$.

La force normale est appliquée au centre de pression. Elle se calcule :

$$\vec{F_n} = \begin{pmatrix} 0.5 \cdot \rho \cdot < \vec{v_i}, \vec{v_i} > \cdot A_{ref} \cdot CN_\alpha \cdot \alpha \\ \\ 0 \end{pmatrix} \tag{3.14}$$

La force de traînée s'applique au centre de pression et est définie :

$$F_D = D \cdot \begin{pmatrix} sin(\alpha) \\ \\ -cos(\alpha) \end{pmatrix} \tag{3.15}$$

Où la force de drag se calcule

$$D = 0.5\rho \cdot < \vec{v_i}, \vec{v_i} > \cdot A_{ref} \cdot C_d \tag{3.16}$$

Où C_d est le coefficient de traînée. La force de pesanteur de la fusée est tout simplement :

$$F_g = m_{tot} \cdot \begin{pmatrix} 0 \\ -g \end{pmatrix} \tag{3.17}$$

La matrice de rotation Q $\in M_{2x2}$ entre le repère de la fusée vers le repère terrestre est :

$$Q = \begin{pmatrix} cos(\phi) & sin(\phi) \\ -sin(\phi) & cos(\phi) \end{pmatrix} \tag{3.18}$$

3.5.1 Phase 1 : Lancement le long du rail

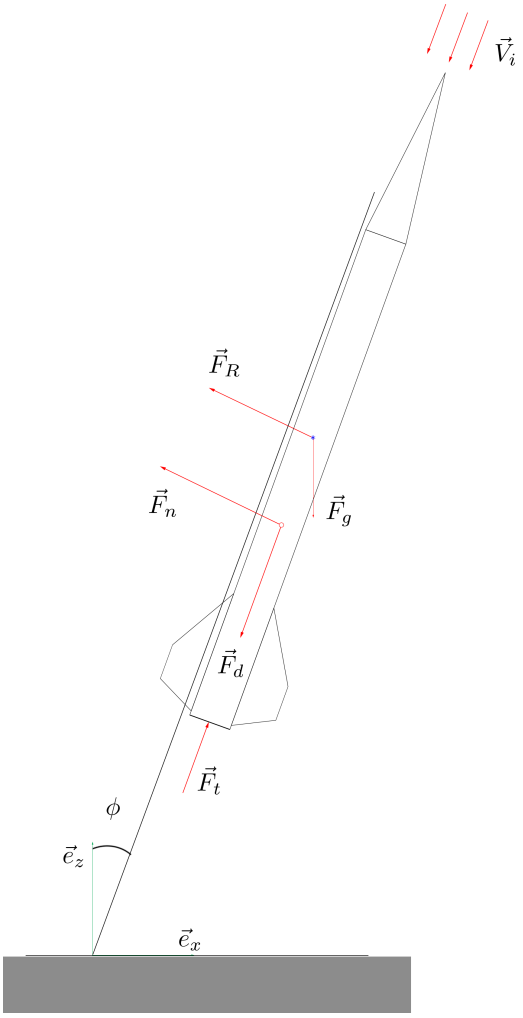


FIGURE 3.3 – Diagramme des forces sur la phase de lancement

Nous avons les conditions initiales suivantes :

$$\begin{cases} z(t=0) = 0 & (3.19) \\ x(t=0) = 0 & (3.20) \\ \phi(t=0) = constante & (3.21) \end{cases}$$

L'angle ϕ est l'inclinaison du rail et selon le vent qu'il y a au moment du lancement peut être changé afin d'optimiser la hauteur à atteindre. On définit la force de réaction du rail :

$$\vec{F}_R = mg \cdot \begin{pmatrix} sin(\phi) \\ \\ 0 \end{pmatrix} \tag{3.22}$$

Cette force permet que la fusée soit retenue par le rail afin qu'elle ne tombe pas par son propre poids, mais qu'il y est une force pesanteur uniquement le long de l'axe de la fusée. Cette force apparaît seulement dans la phase 1. Le calcul de l'accélération est :

$$\dot{\vec{v}}_1 = \frac{1}{m}(Q \cdot (\vec{F}_t + \vec{F}_n + \vec{F}_D + \vec{F}_g - \vec{F}_R) - \dot{m} \cdot \vec{v}) \tag{3.23}$$

L'équation d'état pour le lancement de la fusée le long du rail est :

$$\begin{cases} \dot{x} = V_x & (3.24) \\ \dot{z} = V_z & (3.25) \\ \ddot{x} = \dot{\vec{v}}_1 \cdot \vec{e_x} & (3.26) \\ \ddot{z} = \dot{\vec{v}}_1 \cdot \vec{e_z} & (3.27) \\ \dot{\phi} = 0 & (3.28) \\ \ddot{\phi} = 0 & (3.29) \end{cases}$$

3.5.2 Phase 2 : Vol avec poussée

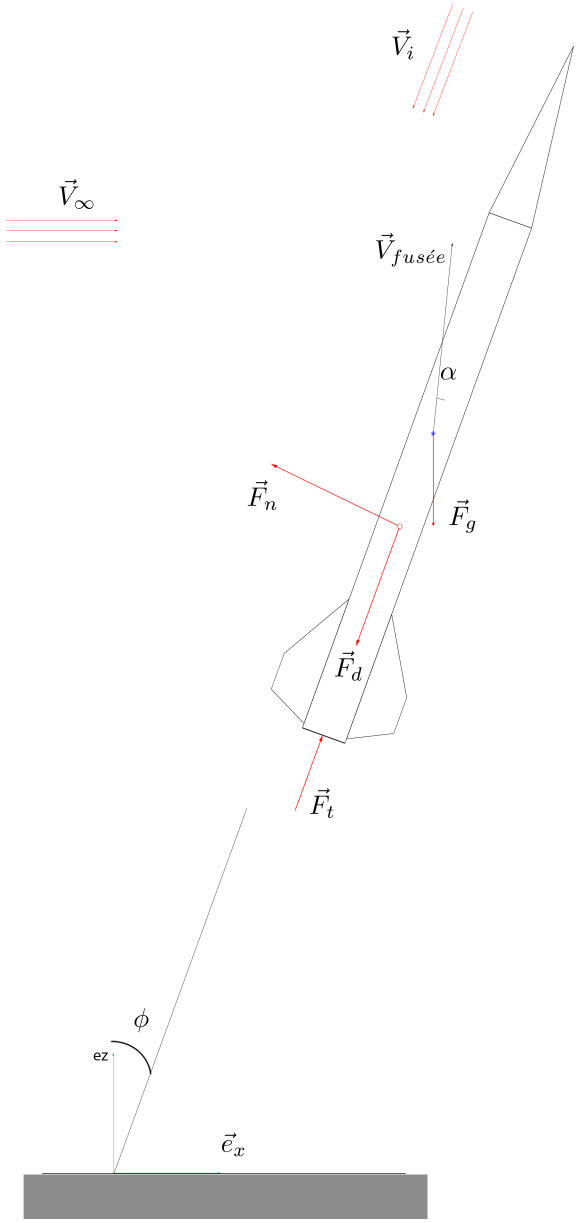


FIGURE 3.4 – Diagramme des forces sur la phase de vol avec la poussée du moteur

Les conditions initiales sont les suivantes :

$$\begin{cases} z(t=t_1) = z_1 & (3.30) \\ \dot{z}(t=t_1) = \dot{z}_1 & (3.31) \\ x(t=t_1) = x_1 & (3.32) \\ \dot{x}(t=t_1) = \dot{x}_1 & (3.33) \\ \phi(t=t_1) = \phi_1 & (3.34) \\ \dot{\phi}(t=t_1) = 0 & (3.35) \end{cases}$$

Où x_1, z_1, ϕ_1, $\dot{x}_1$, $\dot{z}_1$, $\dot{\phi}_1$ sont les valeurs de sortie des vecteurs de positions, vitesses, et angle du lancement le long du rail. t_1 est le temps à la fin de la phase 1. Ces valeurs sont données à la fin de la simulation de la phase 1. Lors de la phase 2, l'équation de l'accélération est :

$$\dot{\vec{v}}_2 = \frac{1}{m}(Q \cdot (\vec{F}_t + \vec{F}_n + \vec{F}_D + \vec{F}_g) - \dot{m}\vec{v}_i) \tag{3.36}$$

Et l'équation de l'accélération angulaire ϕ est :

$$\ddot{\phi}_2 = -\frac{1}{I_r} \cdot ((\vec{F}_n + D \cdot sin(\alpha)) \cdot (CP - CM) + sin(\phi) \cdot \vec{F}_t \cdot (L_{fusée} - CM) + \dot{I}_r \cdot \dot{\phi}) \tag{3.37}$$

L'équation d'état est :

$$\begin{cases} \dot{x} = V_x & (3.38) \\ \dot{z} = V_z & (3.39) \\ \ddot{x} = \dot{\vec{v}}_2 \cdot \vec{e_x} & (3.40) \\ \ddot{z} = \dot{\vec{v}}_2 \cdot \vec{e_z} & (3.41) \\ \dot{\phi} = \phi & (3.42) \\ \ddot{\phi} = \ddot{\phi}_2 & (3.43) \end{cases}$$

La résolution de ce système nous permet de trouver la position de la fusée ainsi que l'angle en tout temps.

3.5.3 Phase 3 : Vol sans poussée jusqu'à l'apogée

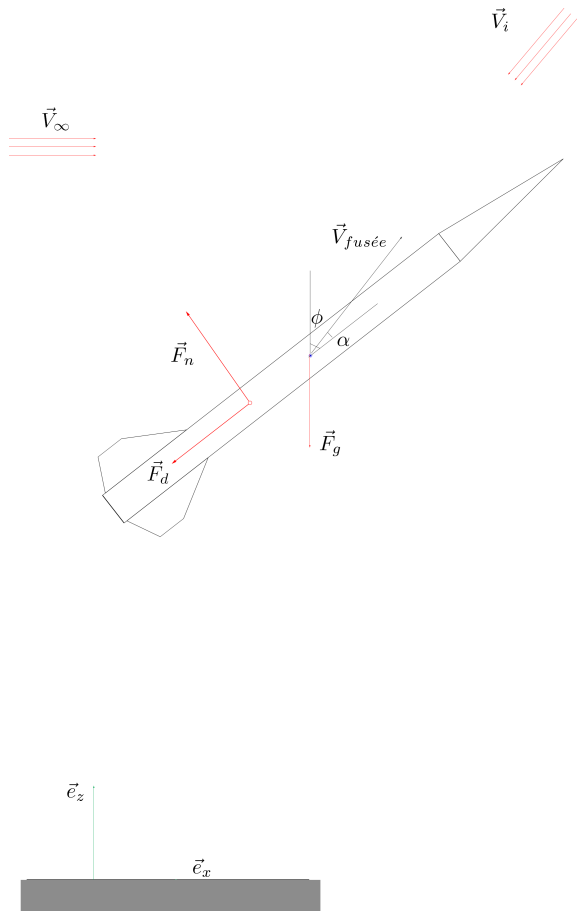


FIGURE 3.5 – Diagramme des forces sur la phase sans poussée jusqu'à l'apogée

Les conditions initiales sont :

$$\begin{cases} z(t=t_2) = z_2 & (3.44) \\ \dot{z}(t=t_2) = \dot{z}_2 & (3.45) \\ x(t=t_2) = x_2 & (3.46) \\ \dot{x}(t=t_2) = \dot{x}_2 & (3.47) \\ \phi(t=t_2) = \phi_2 & (3.48) \\ \dot{\phi}(t=t_2) = \dot{\phi}_2 & (3.49) \end{cases}$$

Où x_2, z_2, ϕ_2, $\dot{x}_2$, $\dot{z}_2$, $\dot{\phi}_2$ sont les valeurs de sortie des vecteurs de positions, vitesses, et angle du du vol avec poussé par le moteur. Ici t_2 est le temps à la fin de la phase 2. Ces valeurs sont données à la fin de la phase 2 de la simulation.

Lors de la phase 3, l'équation de l'accélération est donnée par

$$\dot{\vec{v}}_3 = \frac{1}{m}(Q \cdot (\vec{F}_n + \vec{F}_D + \vec{F}_g) - \dot{m}\vec{v}_i) \tag{3.50}$$

Et l'équation de l'accélération angulaire ϕ est :

$$\ddot{\phi}_3 = -\frac{1}{I_r}((< \vec{F}_n; \vec{e}_x > + D \cdot sin(\alpha)) \cdot (CP - CM) + \dot{I}_r \cdot \dot{\phi}) \tag{3.51}$$

Rappel : α est l'angle d'attaque. On peut donc écrire l'équation d'état de la troisième phase :

$$\begin{cases} \dot{x} = V_x & (3.52) \\ \dot{z} = V_z & (3.53) \\ \ddot{x} = \dot{\vec{v}}_3 \cdot \vec{e_x} & (3.54) \\ \ddot{z} = \dot{\vec{v}}_3 \cdot \vec{e_z} & (3.55) \\ \dot{\phi} = \dot{\phi} & (3.56) \\ \ddot{\phi} = \ddot{\phi}_3 & (3.57) \end{cases}$$

Nous arrêtons la simulation par des «events» qui l'arrête après certaines conditions que nous avons choisies. Les vols restants suborbitaux, la fin de nos simulations est l'apogée, la phase de descente s'effectuant à vitesse plus ou moins constante à l'aide d'un parachute.

3.6 Structure du code

Les classes et les fonctions principales du code sont présentées ici. Le code est organisé en deux classes et deux fonctions. Celles-ci sont présentées dans le document annexe [5].

Classe : Rocket Définit la géométrie, le positionnement des éléments internes, le moteur et d'autres paramètres de la fusée. Les méthodes permettent surtout d'estimer les forces aérodynamiques et l'inertie de la fusée.

Classe : Simulation Définit les paramètres de simulation et les résultats à afficher après la simulation.

Fonction : Simulate Prend en entrée un objet *Rocket* et un objet *Simulation* et retourne une structure de résultats de la simulation. L'évaluation et l'intégration des équations d'état s'effectuent dans cette fonction.

Fonction : drawGraph Prend les résultats de la simulation et affiche les graphes sélectionnés par l'utilisateur.

La séquence d'utilisation typique de ces éléments est illustrée dans la figure 3.6. Une description rapide des attributs, méthodes et fonctions de la classe *Rocket* est explicitée en figure 3.8. La fonction *Simulate* a une exécution séquentielle, celle-ci se trouve en figure 3.7.

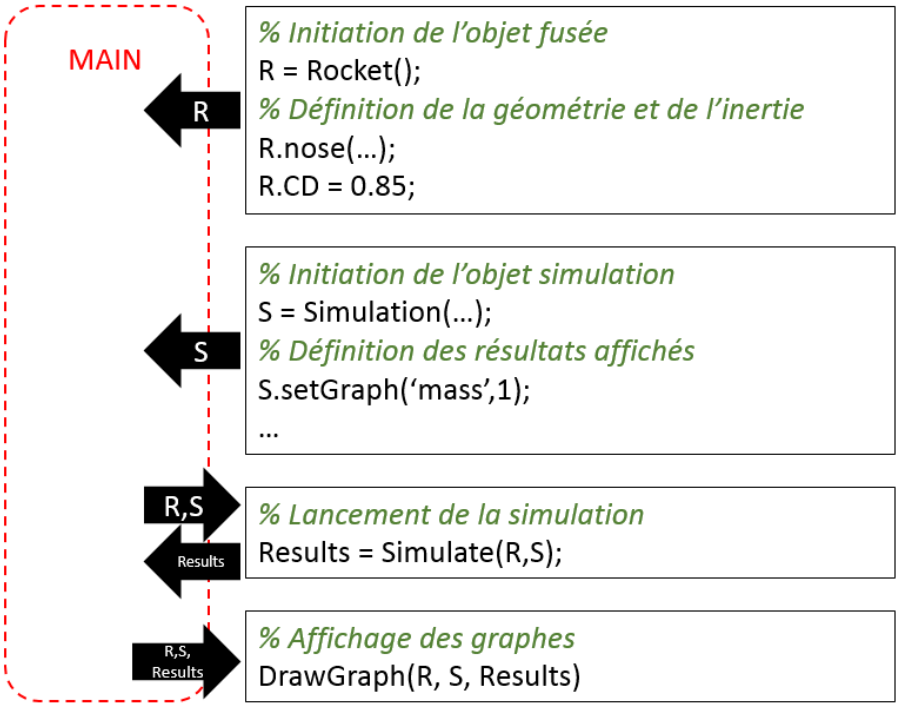


FIGURE 3.6 – Procédure typique d'une simulation.

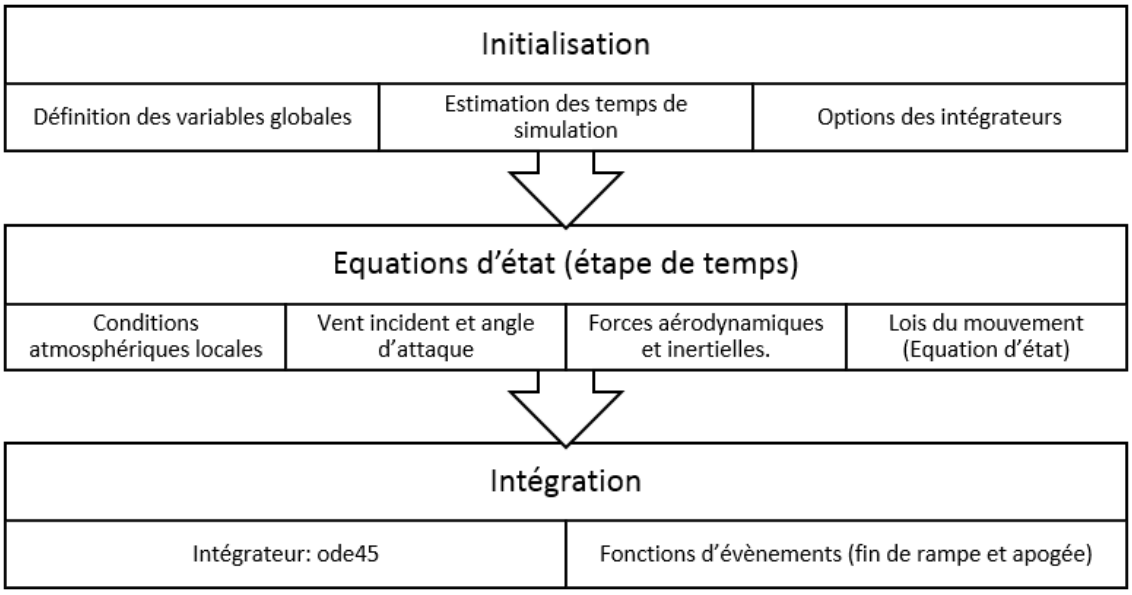


FIGURE 3.7 – Étapes de la simulation.

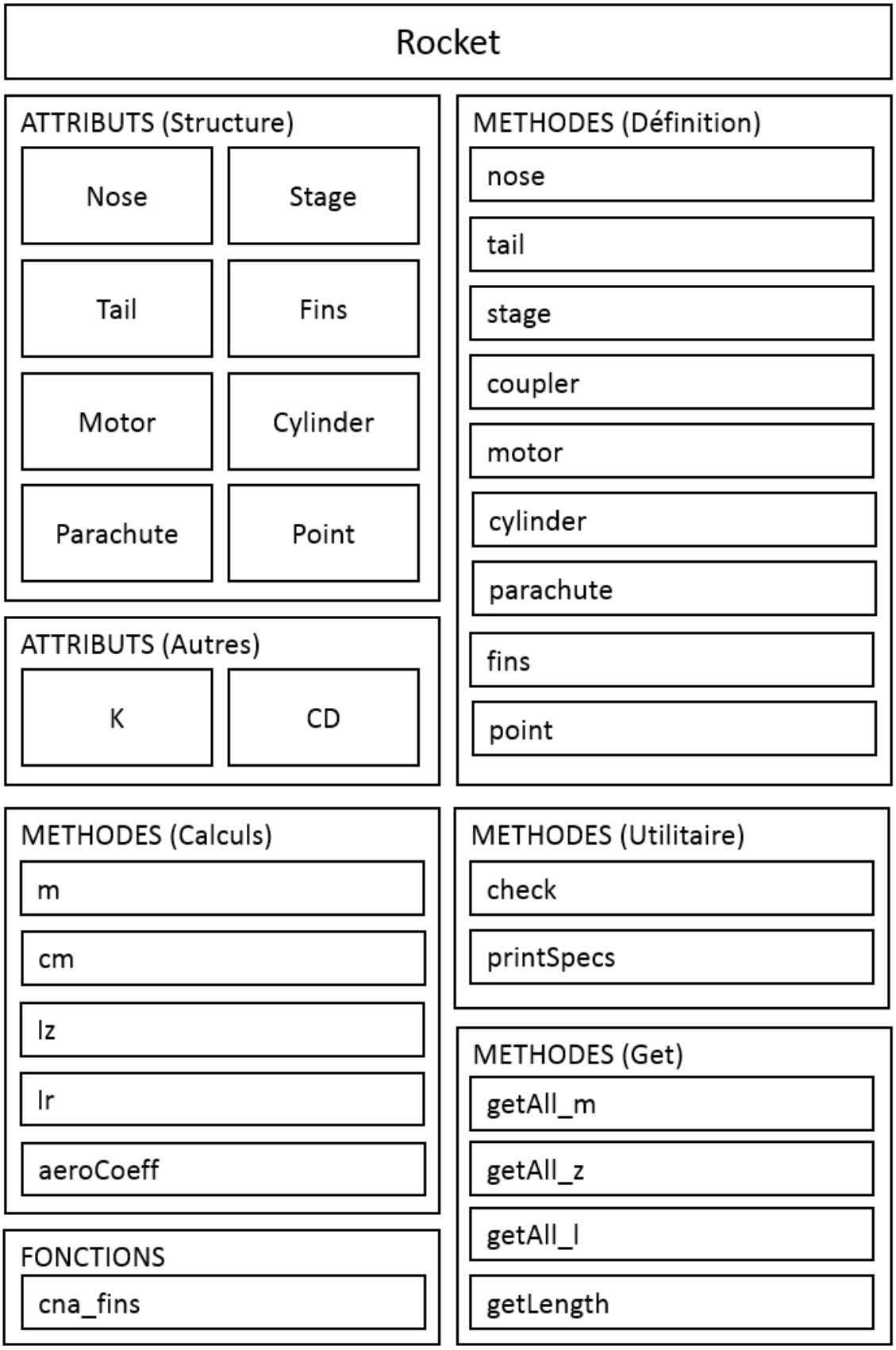


FIGURE 3.8 – Elements de la classe *Rocket*.

Chapitre 4

Validation

Ce chapitre traite de la validation du code MATLAB en le comparant aux mesures expérimentales, au programme de simulation *OpenRocket* et aux résultats du premier code de simulation 1D. Plusieurs cas d'étude seront explorés et les résultats discutés.

4.1 Cas 1 : MadCow Super DX3

La MadCow Super DX3 est la fusée de certification de Julien Rey. Lancée le 01.04.207 à Cernier, NE, en Suisse. La configuration de la fusée est que présentée dans la figure 4.2. Le lancement s'est fait d'une altitude initiale de 820 mètres avec un vent d'environ 10m/s.

Le tableau suivant fait la liste des caractéristiques de base du véhicule.

M_{initiale}	2.294 kg
$M_{\text{combustible}}$	0.148 kg
D_{ref}	4"
C_D	0.51 (*OpenRocket*)
Moteur	*Aerotech* H148-R

Les résultats étudiés ici sont ceux de l'altitude (figure 4.2a) et de l'angle par rapport à la verticale (figure 4.2b) en fonction du temps.

4.1.1 Comparaison avec les mesures

L'altitude prédite par la simulation matlab 2D est comparée aux mesures. L'altitude de la fusée en fonction du temps, mesurée par le système embarqué, est

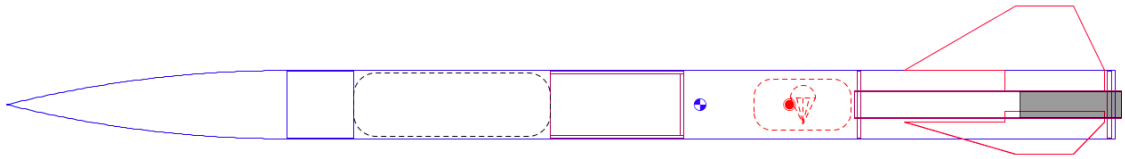
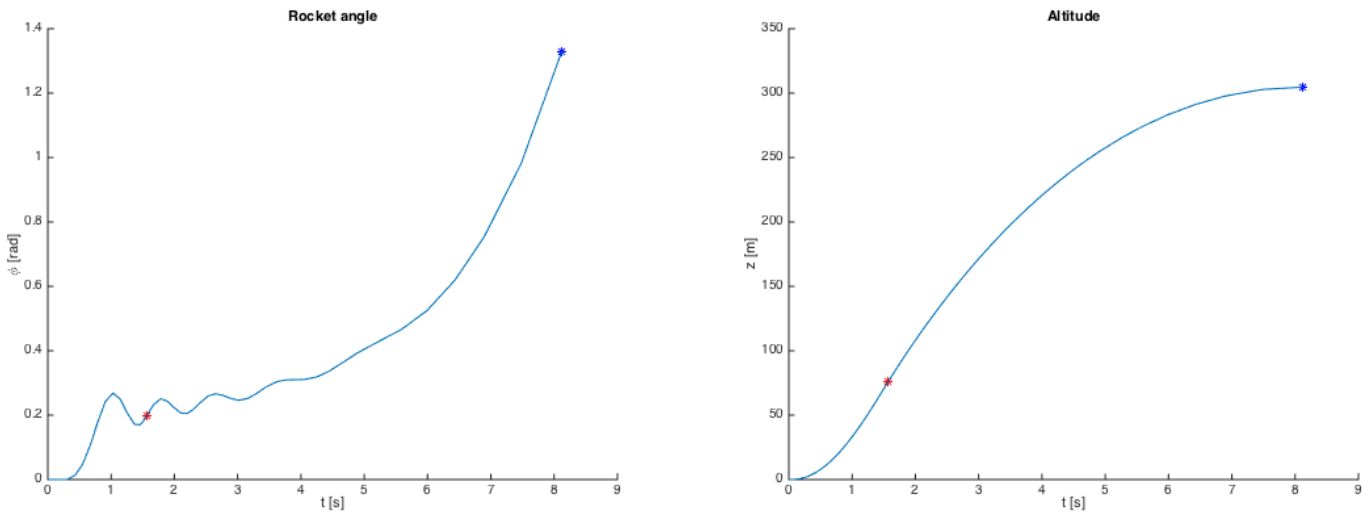


FIGURE 4.1 – Schéma de la MadCow Super DX3.



(a) Angle par rapport à la verticale de la fusée.

(b) Altitude en fonction du temps.

FIGURE 4.2

présentée dans la figure 4.3. Le simulateur trouve une apogée de 304m et les mesures indiquent 298.44m. Nous pouvons également vérifier la dynamique en rotation de la fusée en la comparant à une photo prise lors du lancement (figure 4.4). En prenant la fusée comme référence, nous pouvons estimer son altitude sur la photo. Puis, grâce à la figure 4.2b, on détermine le temps auquel la photo a été prise après le démarrage. Finalement par la figure 4.2a on obtient l'angle par rapport à la verticale prédit par la simulation. On estime ainsi un angle de 6°, tandis que sur la photo on a 14°. Il existe bien une incertitude par rapport à l'angle mesuré sur la photo qui provient de la projection d'un angle tri-dimensionnel sur le plan de l'image. Mais cet effet n'est pas suffisant pour expliquer la différence d'angle. En analysant la dynamique de rotation, on se rend compte qu'elle est très sensible à la répartition de la masse dans la fusée et, de ce point de vue, peu fiable en comparaison avec l'estimation de l'apogée.

4.1.2 Comparaison avec *OpenRocket*

La courbe d'altitude prédite par *OpenRocket*, présentée dans la figure 4.5, montre une apogée de 299m. *OpenRocket* permet également de connaître l'angle azimutal et la marge de stabilité de la fusée, présentés en figure 4.6. On voit que sur la même première oscillation, *OpenRocket* prédit un angle similaire à celui estimé par notre code. De ce fait, on peut se douter que *OpenRocket* est également moins précis dans l'estimation de cette dynamique.

4.1.3 Comparaison avec la Simulation 1D

La simulation 1D prévoit une apogée de 313m. Comme attendu, cette simulation prévoit une altitude plus élevée que la simulation à trois degrés de liberté et les mesures réelles car elle ne tient pas compte des perturbations externes.

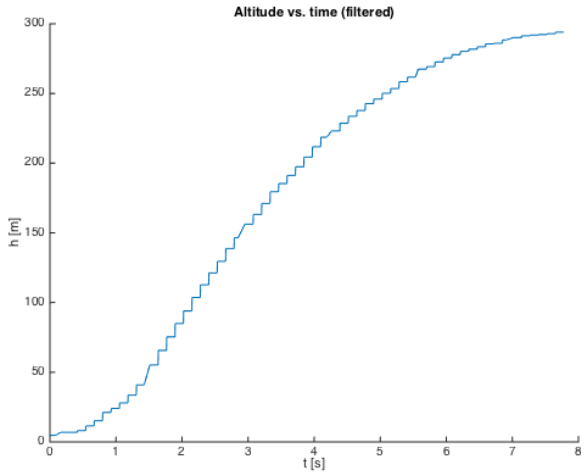
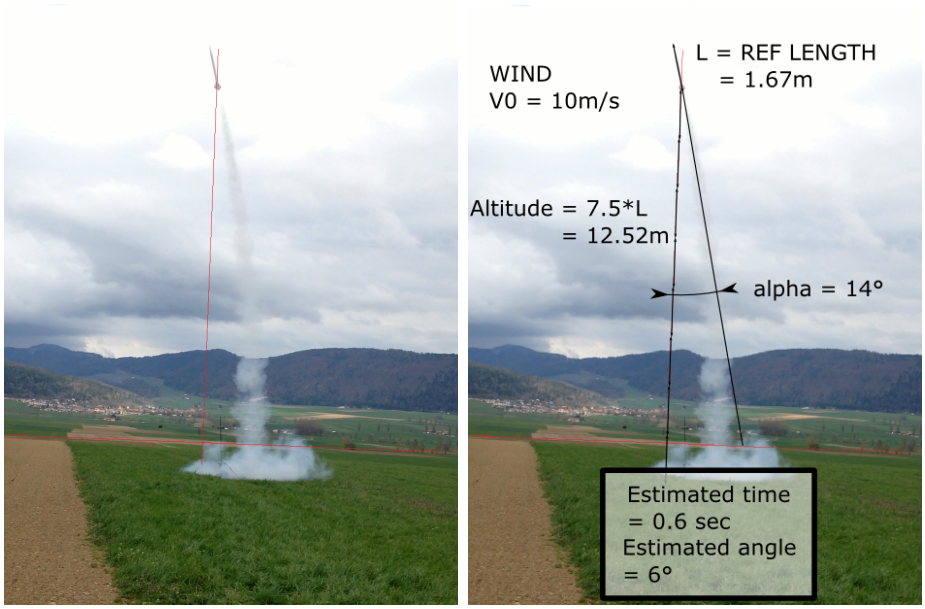


FIGURE 4.3 – Mesures de l'altimètre sur le vol de la DX3.



(a) Photo originale avec guides. (b) Photo annotée.

FIGURE 4.4 – La DX3 dans sa première oscillation.

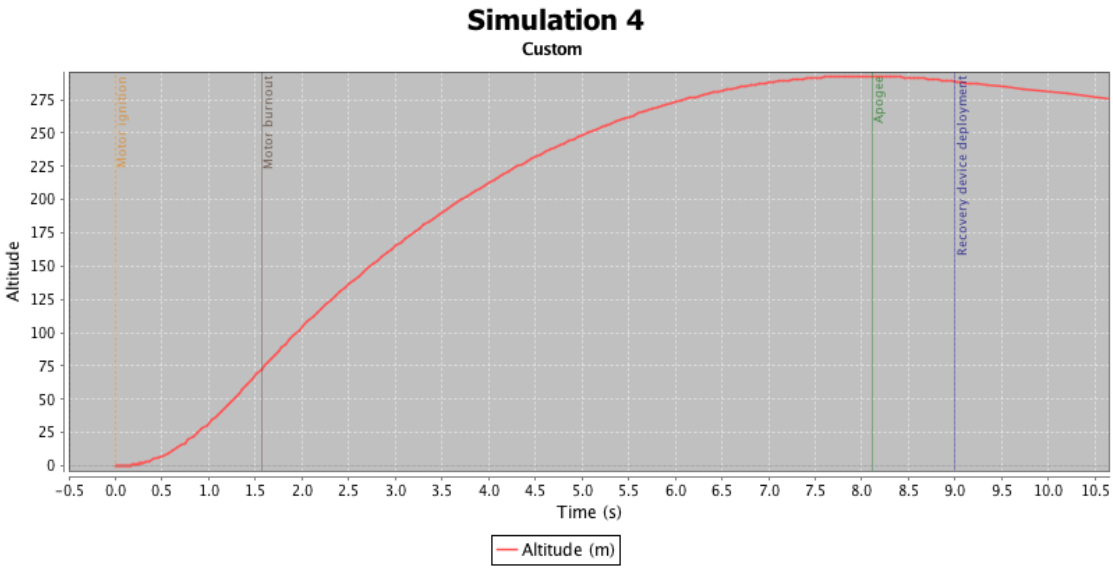


FIGURE 4.5 – Prédiction de l'altitude par *OpenRocket*.

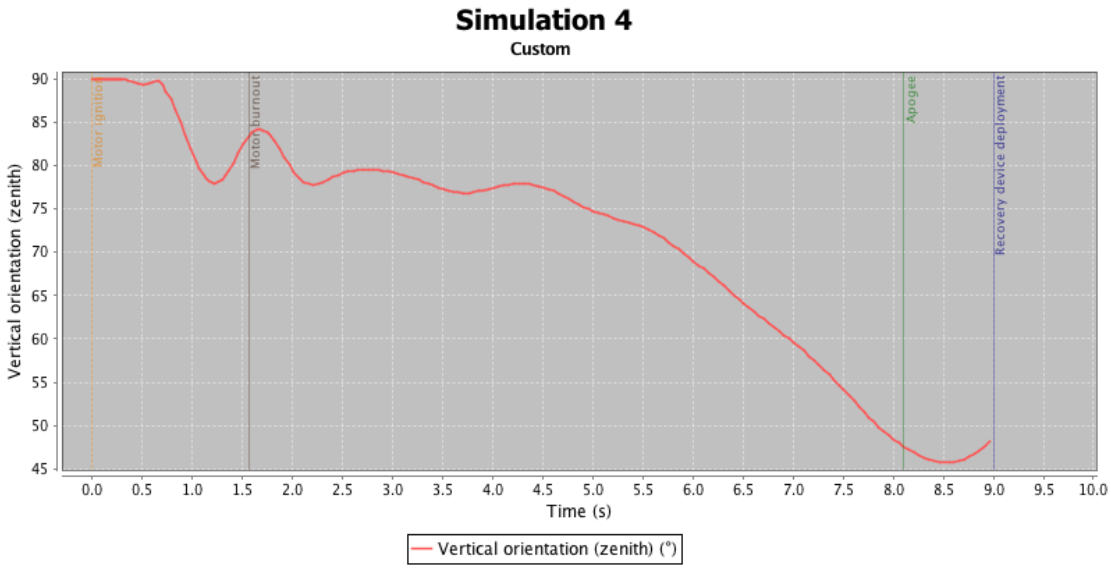


FIGURE 4.6 – Prédictions d'angle azimutal et marge de stabilité par *OpenRocket*.

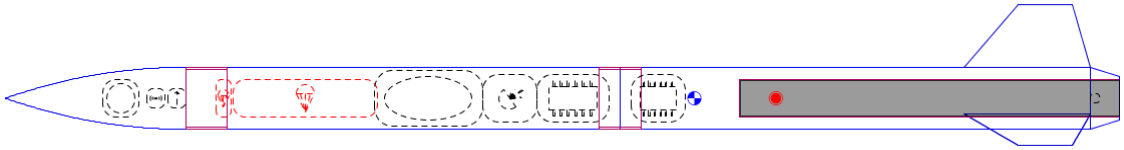


FIGURE 4.7 – Schéma d'une fusée type pour IREC.

4.2 Cas 2 : Fusée type pour IREC

Ici nous allons comparer les prédictions de *OpenRocket* et la simulation 1D par rapport à celles de notre code sur un design de fusée envisagé pour le concours. La fusée considérée est illustrée dans la figure 4.7 a les caractéristiques suivantes :

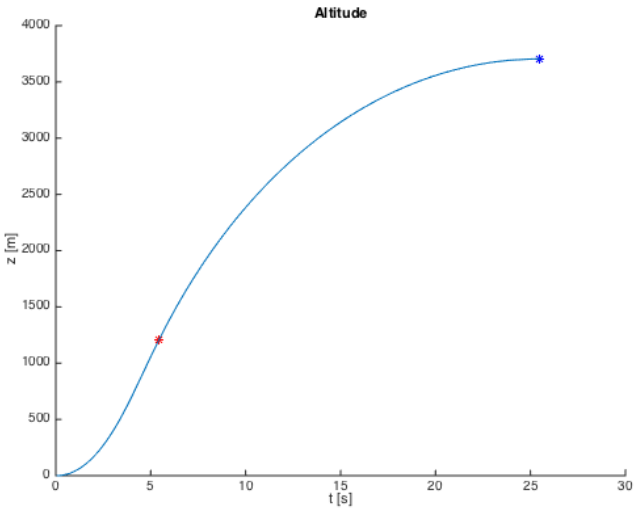
M_{initiale}	27.765 kg
$M_{\text{combustible}}$	7.183 kg
D_{ref}	17cm
C_D	0.55 (*OpenRocket*)
Moteur	*Aerotech* N2220DM-B

Les conditions de vent sont les mêmes que pour le premier cas, 10m/s de vent latéral. Les résultats qui nous intéressent sont ceux de l'altitude pour les trois simulations. Les courbes d'altitudes sont incluses en figure 4.8. Les apogées prédites sont : 3331m selon *OpenRocket*, 3823m selon la simulation 1D et 3705m selon la simulation 2D.

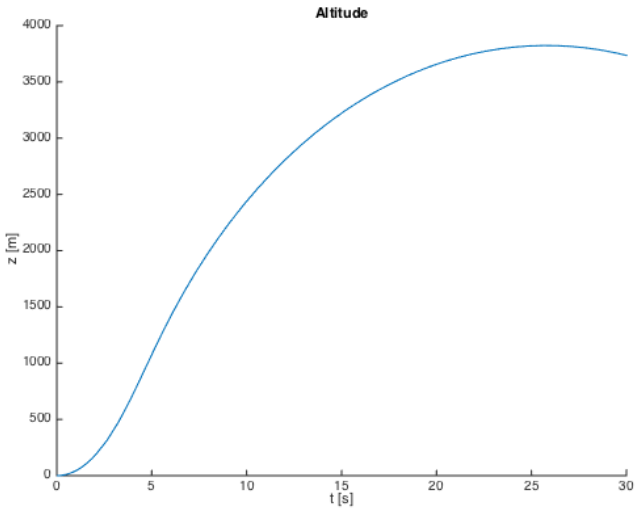
La différence relative d'altitude entre OpenRocket et nos codes de simulation a remarquablement augmenté. L'apogée de la simulation 1D reste supérieure aux deux autres simulations comme attendu. Il est important de remarquer que la simulation 2D prédit des vitesses supersoniques sur une portion du vol, ce qui n'est pas prédit par *OpenRocket*.

4.3 Discussion

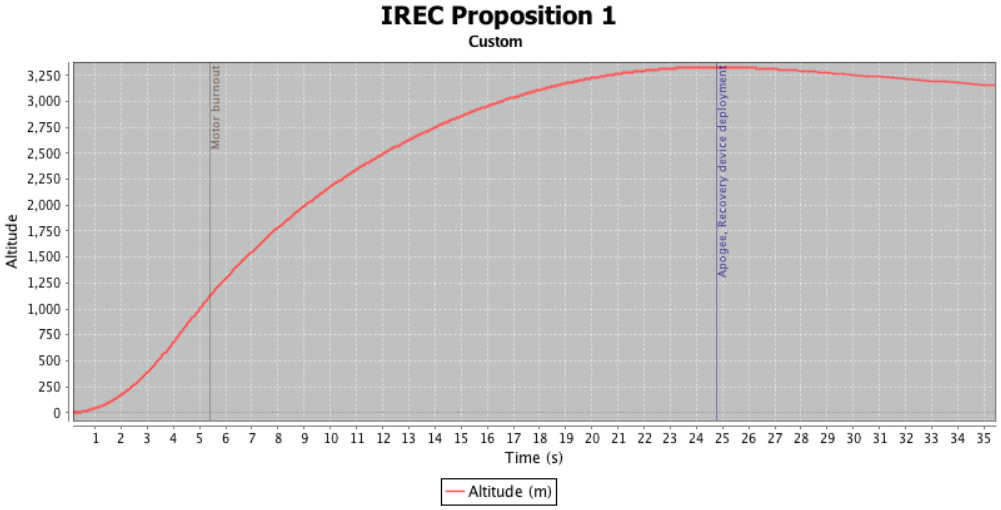
On observe que les simulations donnent de bonnes estimations de l'altitude pour des fusées telles que la DX3 volant sur un moteur de classe H. Pour des altitudes plus élevées, la simulation 2D montre des faiblesses par rapport à *OpenRocket*. Cet effet peu s'expliquer par l'atteinte des limites des modèles utilisés dans la simulation. Effectivement, on rappellera que les vitesses admises par les modèles aérodynamiques sont subsoniques et que le deuxième cas de simulation excède la vitesse du son. On remarque également que la dynamique de l'oscillation ne reflète pas forcément la réalité et qu'il reste important de se tenir dans le domaine prescrit de marges de stabilité (1 à 2 calibres) pour assurer un vol sûr.



(a) Simulation 2D



(b) Simulation 1D



(c) *OpenRocket*

FIGURE 4.8 – Altitudes prédites par les codes de simulation

Troisième partie

Analyse structurelle

Chapitre 1

Cas de charge

1.1 Introduction

L'objectif est de faire un pré-design de la structure en considérant des cas de charge simples et statiques. Ceci afin de déterminer des bornes supérieures pour les contraintes qui risquent d'apparaître durant une mission, celle-ci comprenant le transport, la manutention et le vol. Les applications numériques se basent sur des chiffres couramment rencontrés sur des fusées de compétition étudiante (type IREC) avec un plafond de 10'000 pieds et un payload minimum de 8.8 pounds (charge embarquée).

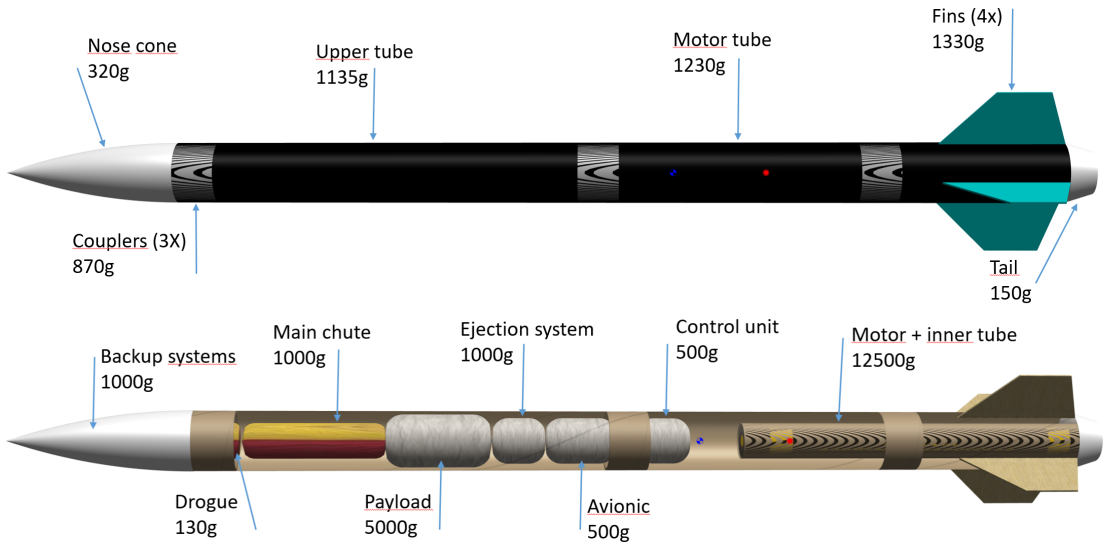


FIGURE 1.1 – Fusée type IREC, répartition des masses

1.2 Flambage dû à la poussée

Comme première analyse nous considérons le flambage d'Euler (Figure 1) du tube lors de l'accélération engendrée par le moteur. Les hypothèses d'un "worst case scenario" sont les suivantes :

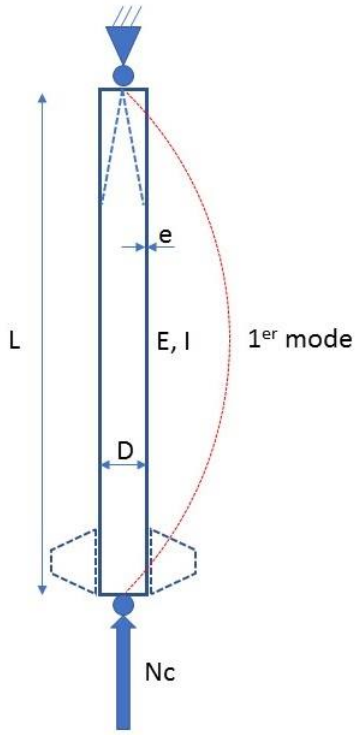


FIGURE 1.2 – 1er mode de flambage d'Euler

- Tube en aluminium isotrope avec $E = 70GPa$
- Longueur $L = 3.5m$
- Diamètre interne $D = 0.15m$
- Masse totale $m = 35kg$
- Poussée Max $Thrust = 3600N$ (correspond à un moteur Aerotech M2100 avec 20% de marge de sécurité)
- Charge critique fondamentale $N_c = \frac{\pi^2 EI}{L^2}$ qui est équivalente à la poussée max du moteur
- $I = \frac{\pi}{64}((D+e)^4 - D^4) \approx \frac{\pi D^3 e}{16}$
- Toute la masse est attachée en haut du tube, la force s'applique donc à son extrémité, le moteur étant considéré comme un pivot à l'opposé.

En utilisant les équations des poutres connues pour N_c et I on obtient facilement la valeur de la charge critique en fonction de l'épaisseur ce qui nous permettra de dimensionner notre tube.

$$N_c = \frac{\pi^3 D^3 eE}{16L^2} = Thrust \tag{1.1}$$

$$e = \frac{16l^2}{\pi^3 D^3 E} N_c \approx 0.1mm \tag{1.2}$$

Il serait possible de raffiner le modèle en considérant des matériaux anisotropes tels que la fibre de carbone dont les propriétés seraient meilleures que celle de l'aluminium. Nous pourrions de plus faire une répartition plus précise des masses ainsi que considérer le cas d'un tube en plusieurs sections avec des coupleurs au milieu. Mais

toutes ces modifications ne feraient que diminuer la valeur trouver pour l'épaisseur e en augmentant E, diminuant la force ou la longueur l.

De plus un rapide calcul montre que pour un rapport D/l inférieur à 1/20, la limite élastique de l'aluminium en compression est atteinte avant même d'avoir appliqué la charge critique nécessaire au flambage.

$$e = \frac{N_c}{\pi D \sigma_{el}} \tag{1.3}$$

On en conclut que le flambage d'Euler n'est pas pertinent dans notre cas d'étude, mais par contre il serait intéressant d'étudier par élément fini la déformation locale de la surface du tube qui pourrait avoir des conséquences pour des cas de charge beaucoup moins élevés. En effet le dimensionnement de l'épaisseur du tube semble être régit par d'autres facteurs que le flambage ou la compression simple (modèles très théoriques), un peu à l'image de la canette de soda, le moindre défaut d'une parois mince dû au matériau, à la fabrication ou même à la manutention peut avoir des conséquences destructrices.

1.3 Déploiement du parachute

Le deuxième cas (Figure 1.3) étudie la force de traction maximale induite par l'ouverture du parachute principal, afin de pouvoir dimensionner les différentes attaches / points d'ancrage. Deux méthodes différentes ont été utilisées afin d'obtenir des bornes supérieures : Une basée sur la conservation de l'énergie et l'autre prenant en compte le drag du parachute à la vitesse maximale. Les hypothèses sont les suivantes :

- Masse de la fusée à vide $m_v = 25kg$
- Vitesse d'ouverture $v_i = 46m/s$
- Vitesse d'atterrissage souhaitée $v_f = 5m/s$
- Masse volumique de l'air constante ρ
- Coefficient de frottement $C_x = 2.2$ (Fruity chutes)
- Longueur de la corde $l = 20m$
- Module de Young de la corde $E_{Nylon} = 3GPa$
- Section de la corde $S = 30mm \cdot 2mm = 60mm^2$

1.3.1 Energie

Dans ce cas on considère que toute l'énergie cinétique dissipée se retrouve dans l'élongation de la corde. En modélisant la corde comme un oscillateur élémentaire il est ensuite possible de calculer la force de rappel maximale que celle-ci exerce sur la fusée.

$$E_{cin} = \frac{1}{2} m_v (v_i^2 - v_f^2) = F \Delta l \tag{1.4}$$

$$\Delta l = l \cdot \epsilon = l \frac{F}{SE} \tag{1.5}$$

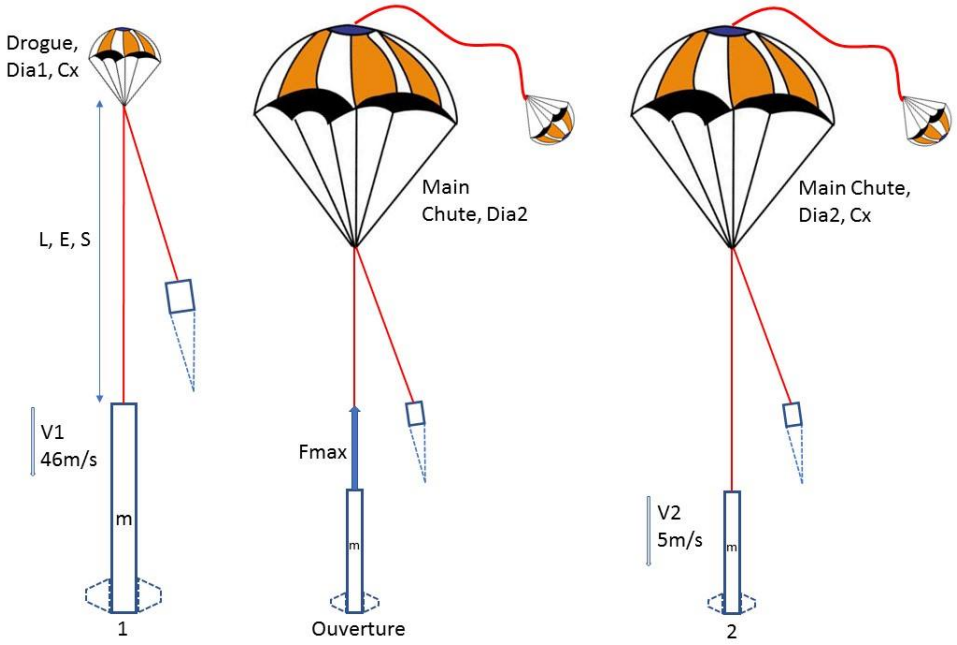


FIGURE 1.3 – Ouverture en 2 phases

$$F = \sqrt{\frac{m(v_i^2 - v_f^2)SE}{2L}} \approx 19.8kN \tag{1.6}$$

1.3.2 Drag force

Nous nous occupons ici uniquement de la force de frottement du parachute qui s'ouvrirait instantanément. La taille du parachute est tout d'abord estimée afin d'obtenir une vitesse d'atterrissage d'environ 5m/s, bien en dessous des règles fixées par la compétition car nous avons envie de préserver au maximum la structure et donc diminuer l'énergie à l'impact est primordial.

$$S = \frac{2m_v g}{\rho C_x v^2} \approx 7,4m^2 \tag{1.7}$$

Ce résultat nous permet donc de sélectionner un parachute chez Fruity chute de 120 inches de diamètre pour une aire de $7,3m^2$.

$$F = \frac{1}{2}\rho S C_x v_i^2 \approx 20.4kN \tag{1.8}$$

Les deux techniques que nous avons employées sont totalement différentes et se basent toutes deux sur des hypothèses qui surestiment grandement la force, néanmoins nous obtenons deux résultats très similaires et élevées ($20kN$) ce qui nous conforte dans l'idée qu'il est important que le premier parachute diminue la vitesse bien en dessous de $46m/s$ afin de limiter les contraintes sur les différentes fixations lors de l'ouverture du parachute principal. La plage fixée par la réglementation autorise une vitesse de descente entre 23 et $46m/s$ pour la première phase.

1.4 Atterrissage

L'objectif de ce cas de charge est d'estimer l'énergie qui devra être dissipée au moment de l'impact de la fusée au sol, pour pouvoir dimensionner une pièce sans importance structurelle qui servirait de "crash box" afin de protéger le reste de la fusée et de préserver les instruments de mesures.

$$E = \frac{1}{2} m_{vide} v^2 \approx 1kJ \tag{1.9}$$

En supposant une vitesse d'arrivée imposée par les règles de la compétition de $v = 9m/s$ l'énergie à dissiper est considérable, c'est pourquoi il nous paraît très important d'assurer une vitesse de descente finale plus lente de l'ordre de $v_f = 5m/s$ ce qui diminuerait d'un facteur 3 l'énergie finale au moment de l'impact.

$$E = \frac{1}{2} m_{vide} v_f^2 \approx 310J \tag{1.10}$$

1.5 Charge latérale

Ce dernier cas de charge vise à étudier la déformation en flexion du tube ainsi que des moments de forces auxquels seront soumises les liaisons entres les différents modules lorsque la fusée subit des accélération latérales importantes. Ces accélérations peuvent avoir deux sources différentes :

- Les corrections de trajectoire aérodynamiques qui arrivent à haute vitesse ou lors de la stabilisation en sortie de rail.
- le transport peut engendrer des accélérations que nous estimons pouvant aller jusqu'à $4g$ (chocs) et nous utiliserons cette valeur comme limite inférieure pour notre dimensionnement si les corrections aérodynamiques se révèlent être plus faibles.

1.5.1 Correction aérodynamique

Il est tout d'abord nécessaire de calculer l'accélération latérale qu'engendre une correction de trajectoire. Pour ce faire nous faisons un certain nombre d'hypothèses :

- Trajectoire circulaire : la fusée décrit un arc de cercle lors de la correction de son angle d'attaque
- Le temps de stabilisation est constant, correspond à une demi période d'oscillations Δt
- Petits angles, $\alpha << 1$

Par de simples considérations géométriques et connaissant la distance verticale y et la déviation horizontale maximale x de la fusée lors d'une oscillation il est possible de trouver le rayon du cercle qui représente la trajectoire de la fusée.

$$r^2 = (r-x)^2 + (y/2)^2 \Leftrightarrow r = \frac{x}{2} + \frac{y^2}{8x} = \Delta tv(\frac{sin(\alpha)}{2} + \frac{cos(\alpha)}{tan(\alpha)} \approx \frac{\Delta tv}{2\alpha} \tag{1.11}$$

$$a_c = \frac{v^2}{r} = \frac{2\alpha v}{\Delta t} \tag{1.12}$$

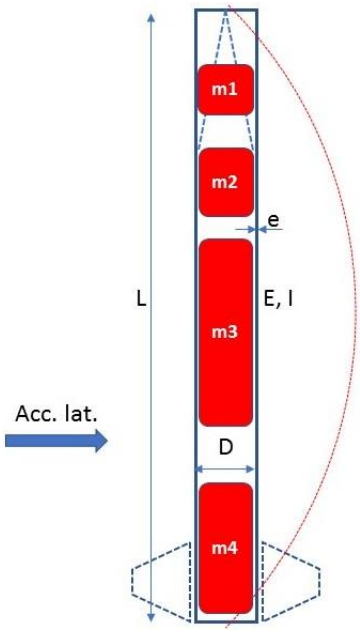


FIGURE 1.4 – Accélération latérale sur masses discrétisées

Ensuite chaque élément de masse est caractérisés par une force répartie qui simule une masse linéique constante et représente sa position dans la fusée.

$$p_i = \frac{m_i a_c}{l_i} \tag{1.13}$$

Par principe de superposition on peut additionner les effets engendrés par les différents éléments. On calcule donc l'effort tranchant et le moment en fonction de x pour chaque élément puis on additionne pour finalement obtenir les contraintes normales et de cisaillement maximales. On sait que la traction maximale apparaît le plus loin possible du centre de la poutre, de ce fait $y_{max} = 75mm$.

$$\sigma_{max} = \frac{y_{max} M_{tot}}{I} \tag{1.14}$$

Le cisaillement max pour un tube mince peut souvent être négligé par rapport à la contrainte de traction mais il est néanmoins donné par la formule suivante ou la surface $S = \pi De$.

$$\tau_{max} = 2\frac{T_{tot}}{S} \tag{1.15}$$

1.5.2 Transport

Durant le transport de nombreux événements indépendants de notre volonté peuvent arriver. En effet, les compagnies de transport ne sont pas réputées pour leur douceur vis à vis de la marchandise, et il est donc fort probable que la fusée se fasse secouer plus que désirer. De plus lors du transport de la fusée totalement assemblée jusqu'au pas de tir, des accélérations latérales non négligeables peuvent apparaître. Ainsi si les accélérations latérales dues aux corrections aérodynamiques

pendant le vol se révèlent être faibles, il est nécessaire de prendre en considération un cas de charge statique avec un point d'appuis à chaque extrémité de la fusée. Ceci couvrant les cas de manutention. Les formules sont équivalentes au point précédent si ce n'est l'accélération latérale fixée arbitrairement à $4g$.

Finalement, pour ce dernier cas de charge, nécessitant la position et les masses de tous les composants internes à la fusée, nous avons décidé de l'intégrer directement à la simulation et de récupérer les valeurs numériques calculées pour les accélérations latérales dues aux corrections aérodynamiques, plutôt que l'approximation faite au point 1.5.1. Nous obtenons donc en figure 1.6 une évolution dynamique des moments de force en fonction de l'accélération latérale que subit la fusée.

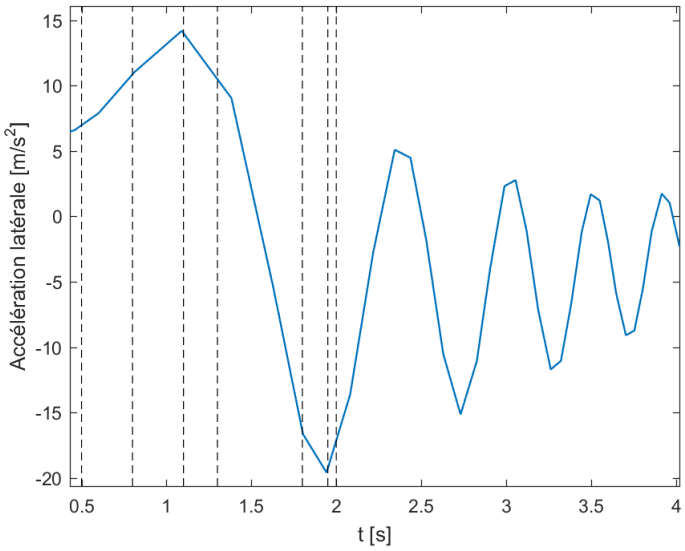


FIGURE 1.5 – Accélérations latérales dues aux corrections aérodynamiques

Ainsi on trouve que le moment maximum auquel notre fusée sera soumise apparaît lors de mauvaise manutention et est équivalent à $\approx 400Nm$. On peut donc calculer les contraintes maximales que cela représente pour le tube pour différentes épaisseurs, voir figure 1.8.

Après analyse des résultats de vols dans la partie expérimentation IV nous avons remarqué qu'il existe une autre situation où l'accélération latérale peut être très importante. Il s'agit de l'ouverture du parachute qui fait décrire un arc de cercle aux parties de la fusée au bout des cordes. Nous avons ainsi mesuré des accélérations jusqu'à $6g$, ce qui dépasse nos estimations maximales. Néanmoins le dispositif de mesure étant situé dans le nez de notre fusée, il est difficile de tirer des conclusions, mais il serait intéressant d'approfondir le sujet lors de lancements ultérieurs.

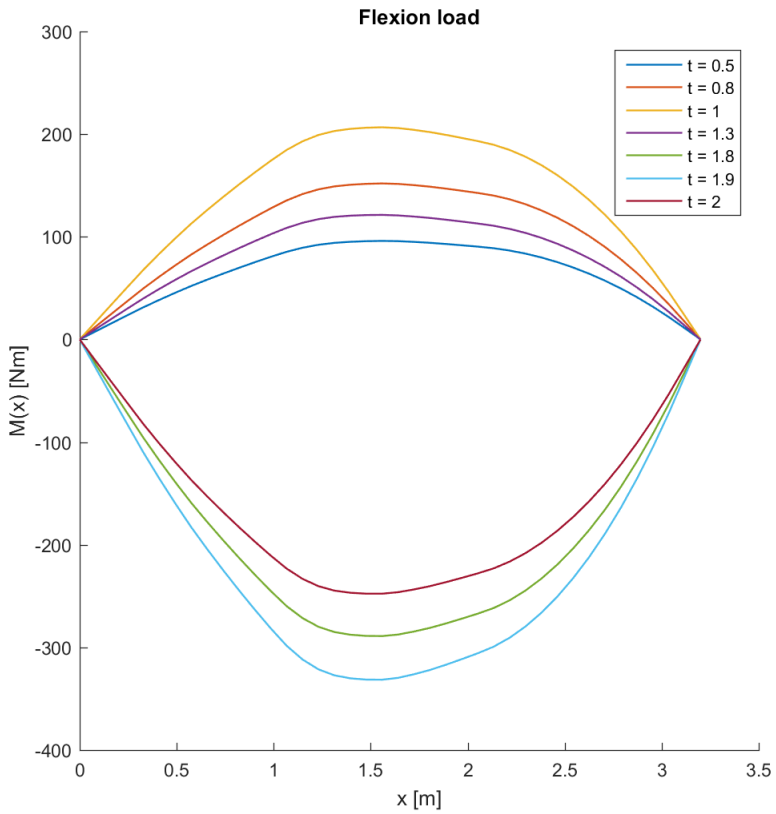


FIGURE 1.6 – Moment induit par les corrections aérodynamiques

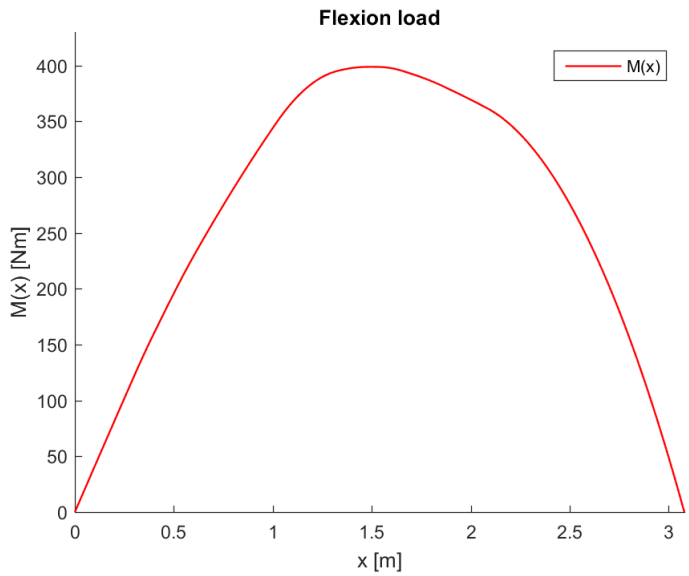


FIGURE 1.7 – Moment de force induit par une accélération latérale de $4g$

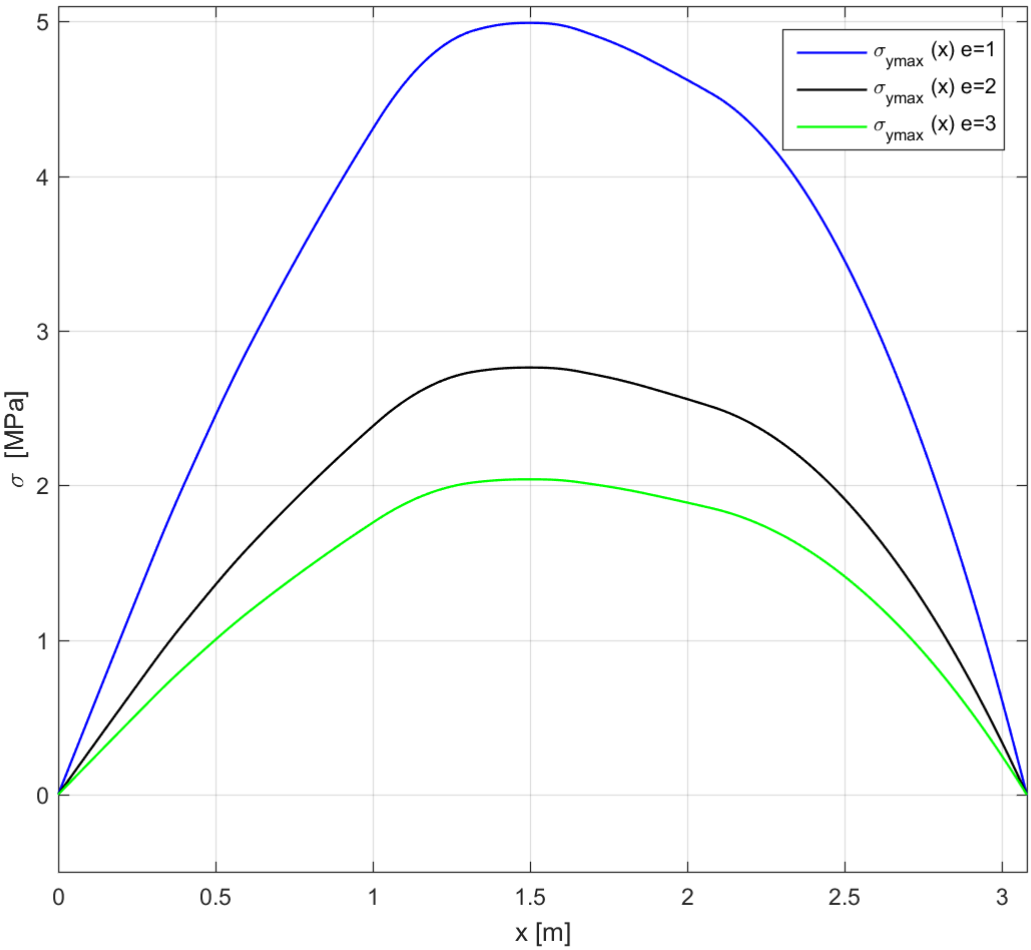


FIGURE 1.8 – Contraintes maximales pour un moment de 400Nm

Chapitre 2

Analyse par éléments finis

2.1 Simulation du tube

2.1.1 Présentation

Dans cette section, nous présentons un outil de simulation pour le tube. Nous considérons uniquement le cas de charge axiale lors du décollage. Pour modéliser au mieux l'analyse et être le plus en adéquation avec les éléments déjà présentés dans ce rapport, voici les hypothèses de modélisation :

— Avec la forme des coupleurs définie et analysée en section 2.2, on remarque que les modules doivent être plus petit que le diamètre du tube pour pouvoir passer dans le coupleur.
  Les modules sont joints au tube extérieur par les bagues de couplage. Il y a une bague à chaque extrémité des module. Ainsi l'effort transmis d'un module au tube se fait par le cisaillement du collage. On peut observer l'idée à la figure 2.1. Chaque zone grisée correspond à l'effet d'une des deux bagues d'un module.
— La force que chaque module exerce sur les tubes correspond à la masse multipliée par l'accélération. Elle est ensuite modélisée comme un effort tranchant sur la surface interne du tube.
— La plaque de poussée située sous le tube correspond à la référence et sera considérée comme encastrement.
— Le moteur est attaché à la plaque de poussée, il n'y a par conséquent aucun effort sur le tube.
— Il peut y avoir un réservoir, dans ce cas il sera placé sous les modules comme schématisé à la figure 2.1. Celui-ci est considéré comme collé au tube.

Créer une simulation plutôt qu'un seul cas de charge en exemple permet une certaine flexibilité sur le design et une itération rapide en fonction des résultats. Avec un outil, il est possible de réitérer l'analyse jusqu'à obtenir un résultat satisfaisant. Pour encore plus de simplicité, cette simulation est modélisée comme une fonction à ajouter (add-in) au code MATLAB présenté dans la partie II.

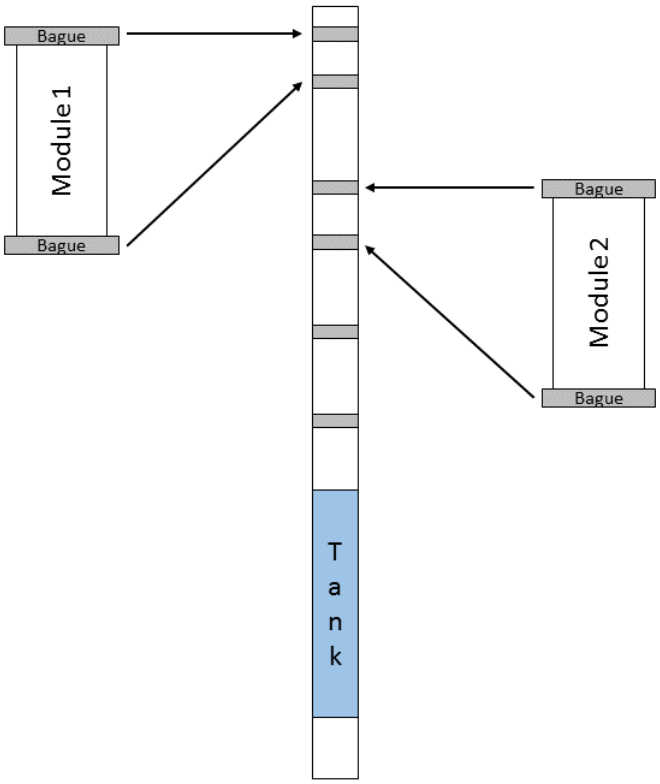


FIGURE 2.1 – Modélisation des modules dans le tube

2.1.2 Outils

Pour procéder à la mise en place de l'analyse, la fonction nécessite plusieurs éléments émanant du code de simulation. Dans cette simulation, il est possible de concevoir la géométrie d'une fusée avec des objets.

La fonction MATLAB que l'on utilise pour la mise en place de l'analyse réutilise un modèle décrit de fusée. Elle génère ensuite un script python représentant la géométrie de la fusée afin de la recréer dans un programme implémentant un solveur d'éléments finis.

Pour l'analyse en éléments finis nous utilisons Abaqus. Celui-ci permet l'utilisation de script python afin de modéliser la géométrie, appliquer les efforts, créer une première proposition de maillage et soumettre une première analyse.

2.1.3 Protocole

Pour correctement transmettre les dimensions à Abaqus et puisqu'il s'agit d'un pré-script dans lequel il faut ajouter des valeurs, il faut définir un protocole. En réalité une fusée est déjà cachée dans ce dernier. Il faut libérer les valeurs que l'on souhaite ajouter. Par exemple, si on ajoute un module, il faut laisser les plans apparaître puis laisser la coupe du tube par ces plans pour ensuite appliquer des conditions de bords sur la surface générée. Le protocole des étapes à respecter est le suivant :

Étape	Condition	Vient ensuite
Génération du Tube	ø et L_{Tube}	Modélisation Module 1
Modélisation Module 1	z_{m1}, m_{m1} & l_{m1}	Modélisation Module 2
...	...	...
Modélisation Module 6	z_{m6}, m_{m6} & l_{m6}	Maillage
Maillage	Taille approximative	Analyse

Comme il s'agit d'un code pré-écrit avec une fusée déjà modélisée, il y a une certaine limitations du nombre de modules que l'on peut rajouter. Le script limite à 6 le nombre d'éléments que contient une fusée. Dans tous les cas, MATLAB génère 6 modules. Si la fusée en contient moins, la fonction génère des modules de masse nulle.

Pour générer un module dans Abaqus, il faut absolument remplir un objet de type "Cylinder" dans la définition de la fusée. Les autres objets, hormis le moteur, ne sont pas pris en considération.

Une condition if dans le code permet de savoir si un module est de masse nulle ou non. Ainsi la condition fait disparaître toutes les commandes Abaqus liées à un module si sa masse est nulle et celui-ci n'apparaît jamais dans la géométrie.

Comme précisé dans les souhaits de ce programme, on remarque que le moteur n'applique pas de force sur le tube. Or, s'il s'agit de moteur hybride ou liquide, il peut y avoir un réservoir. Celui-ci est considéré collé dans le tube, la surface de contact est alors tout le long du réservoir.

2.1.4 Paramètres utilisateurs

Mise à part le fonctionnement autonome du programme, il faut faire appel à la fonction dans le code principal. Lors de l'appel de cette fonction, il faut préciser quelques paramètres. La fonction se présente sous la forme :

```
Abaqus_Analysis(eBague,h,ply_Thick,a,R);
```

Les paramètres en entrées sont :

Variable	Définition
eBague	Épaisseur de la bague de maintient entre un module et le tube
h	Taille approximative du maillage
ply_Thick	Épaisseur d'une couche de fibre de carbone
a	Accélération que l'on souhaite appliquer
R	Fusée définie dans simulation

2.1.5 Limitations

Même si le protocole est respecté, il peut y avoir des erreurs que le code pas abouti ne gère pas. Il faut donc respecter quelques éléments supplémentaires.

1. Le réservoir doit se situer en-dessous de tous les modules.
2. L'ordre de déclaration des modules dans l'objet "Cylinder" doit être respecté comme suit : Le module le plus haut doit être déclaré en premier et le plus bas en dernier.

3. Les modules ou le réservoir ne doivent pas se couper.
4. Il ne faut pas dépasser 6 modules et 1 réservoir.
5. Pour le moment, le réservoir est caché dans l'objet moteur.
6. Ce programme propose un première géométrie, il est possible de la changer une fois dans Abaqus si elle ne plaît pas.
7. Le seul composite proposé est un $[45°, 0°, -45°]$. Il est possible de le changer à vôtre guise une fois Abaqus lancé.

Ne pas respecter un de ces points peut mener à une géométrie non complète. En réalité le script s'arrêtera là où il rencontre un problème. Le script, dans l'état actuel, est présenté en annexe D.

2.1.6 Éléments à ajouter

Pour améliorer la génération de la géométrie Abaqus, il serait intéressant de compléter le script MATLAB pour gérer les erreurs mentionnées plus tôt. Il pourrait par exemple réarranger les éléments s'ils ne sont pas définis dans l'ordre. Le programme pourrait aussi indiquer les erreurs non solvables et s'arrêter.

Concernant le script python, il est difficile de générer des formes uniquement avec des commandes. Il faut alors d'abord construire la pièce dans l'environnement Abaqus puis retravailler le fichier .jnl. Il serait intéressant d'observer des récurrences dans le code pour pouvoir générer une infinité de modules.

Chaque effort est défini par une direction. Celle-ci est décrite par un vecteur. La possibilité de le changer permettrait d'étudier plus de cas de charges. Notamment le fait de porter la fusée par ses deux extrémités. Alors on s'intéresserait à la résistance en flexion de la fusée.

2.1.7 Conclusion & Exemple

Cet outil est encore en création. Il est possible que la forme de la fusée finale ne corresponde pas au hypothèses émises. Dans ce cas il faudra changer la pré-modélisation disponible en script. Celle-ci doit être réalisée sur Abaqus.

Pour terminer cette section portant sur la simulation du tube, nous souhaitons présenter un exemple. Celui-ci porte sur une première proposition non définitive d'un design de fusée par l'EPFL Rocket Team. Cette fusée est présentée en figure 1.1. Les modules nécessaires sont un payload, un système de récupération, un module d'électronique principal et un second redondant. Le moteur se situe à la fin du tube. La modélisation réalisée par le script python est présentée en figure 2.2. Toutes les surfaces hachurées sont soumises à des efforts. On reconnaît les paires de bagues.

Un effort est appliqué pour chaque module et donc pour chaque paire de bagues, comme illustré à la figure 2.3. Pour représenter la plaque de poussée, on modélise le bas du tube par un encastrement, comme présenté en figure 2.4.

Finalement, avec une première proposition de maillage et de matériau, Abaqus nous présente un résultat d'effort sur le tube. Généralement, l'effort maximal se situera au bas de la fusée, au niveau de la plaque de poussée. Un résultat typique est proposé à la figure 2.5.

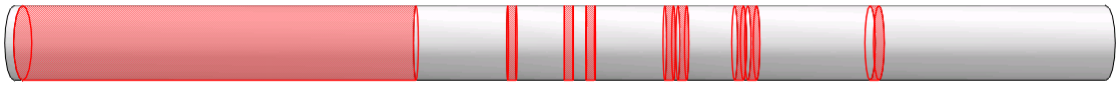


FIGURE 2.2 – Tube modélisé dans Abaqus

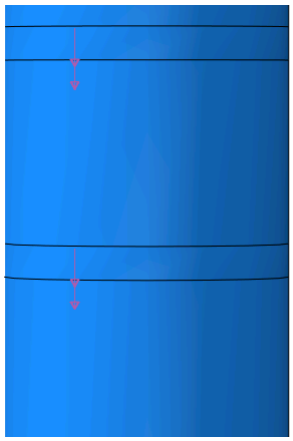


FIGURE 2.3 – Application des efforts pour chaque module

FIGURE 2.4 – Condition de bord par la plaque de poussée

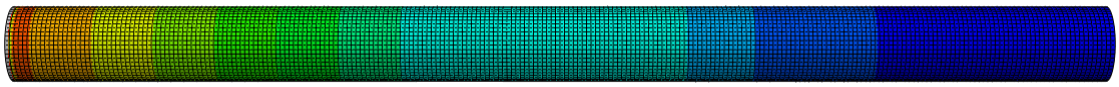


FIGURE 2.5 – Idée de résultats proposé par la simulation

2.2 Analyse des coupleurs

Contrairement au reste du projet qui se voulait d'ordre général, nous présentons ici un design spécifique pour les coupleurs liant les différents tubes et modules de la fusée. En effet comme nous avons plus le voir au chapitre 1, le cas de charge qui paraît le pu contraignant d'un point de vue de la structure externe de la fusée correspond à l'ouverture du parachute principal qui pourrait engendrer une force de traction d'environ $20kN$ sur les points d'attache dans le pire des cas pour une fusée de type IREC. Nous nous sommes donc penchés sur le design de pièces de liaison pouvant supporter une telle charge.

2.2.1 Présentation

Le coupleur est constitué de 4 pièces différentes :

1. Deux anneaux que l'on colle dans chacune des sections tubulaires à relier
2. Une bague interne avec filetages pour les vis
3. Une bague externe séparée en 12 secteurs
4. Des vis M4x20-8.8

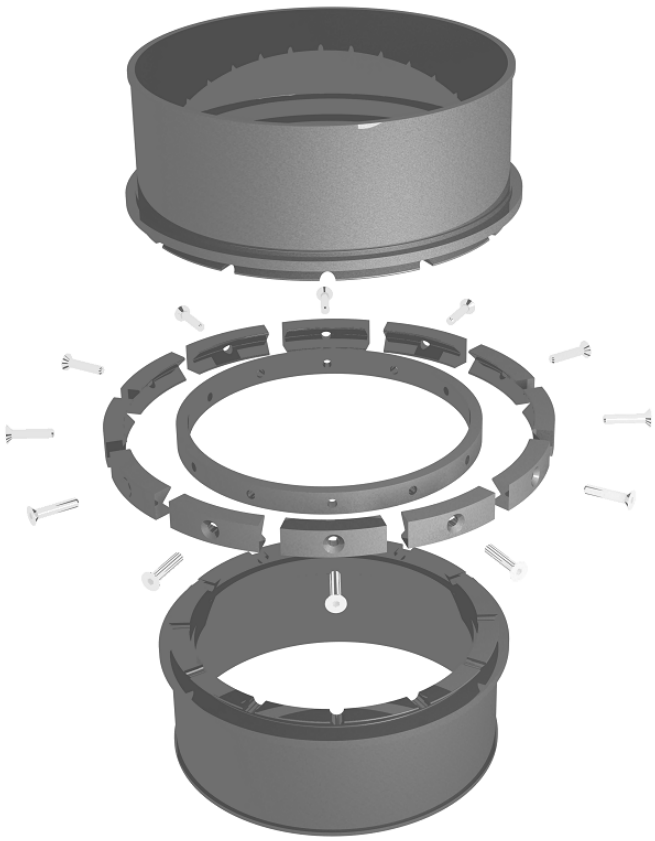


FIGURE 2.6 – Vue exlposée du coupleur

FIGURE 2.7 – Design de clampe existant

L'objectif est de lier les deux anneaux ensemble à la manière d'une clampe (ou V-band) en serrant progressivement les 12 quartiers autour de l'assemblage. Ce design à l'avantage d'avoir une surface externe complètement lisse une fois le montage complètement réalisé, contrairement aux produits déjà existants qui ont une surface externe non-travaillée car non fonctionnelle, et une vis de serrage saillante, voir figure 2.7. [1] En effet, ces types de fixation ont tous une vis qui permet un serrage tangentiel, mais pour une fusée ce design n'est pas souhaitable, une telle protubérance dans le flux d'air devant être évitée à tout prix pour limiter les effets transsoniques locaux.

Afin d'éviter cette vis latérale pour serrer la clampe nous avons donc décidé de séparer la bague externe en plusieurs quartiers et de les fixer indépendamment les uns des autres avec une vis radiale. Néanmoins cette technique à le désavantage de découper la bague externe ce qui lui permet de s'ouvrir légèrement sous la traction et modifie la répartition des efforts. C'est pour cette raison que notre design est constitué de 12 parties, en effet en diminuant la longueur des éléments on diminue d'autant plus le déplacement des extrémités.

2.2.2 Optimisation de la taille

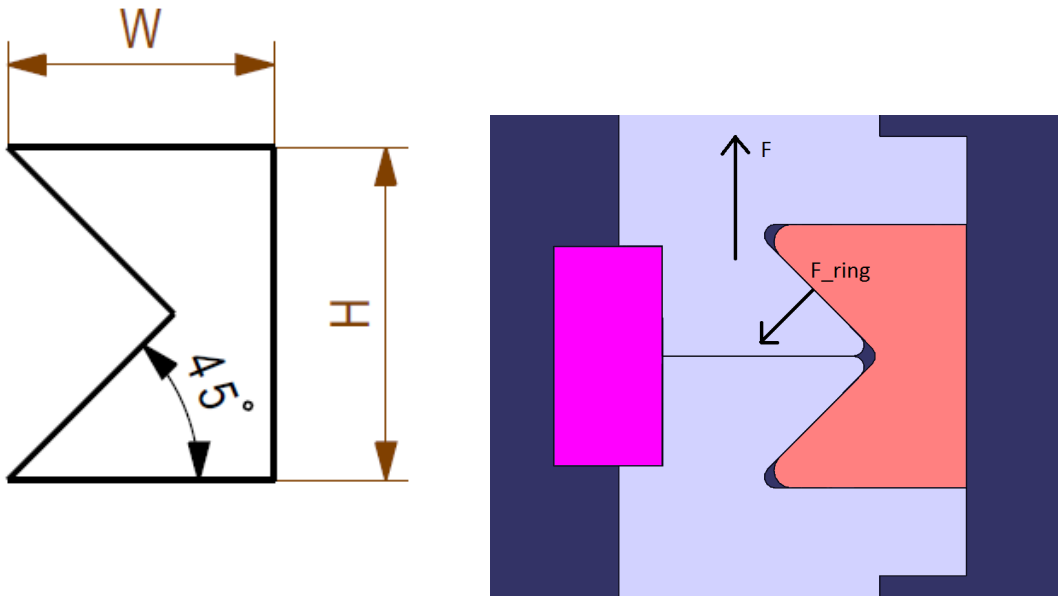
Dans le but de réduire la masse totale et le coût du coupleur, ainsi que de maximiser le diamètre de passage à travers celui-ci, nous avons procédé à une optimisation de notre design par la méthode des éléments finis.

Analyse axisymétrique

Une analyse axisymétrique d'une bague du coupleur seule a permis de faire les premières optimisations géométriques, et d'obtenir un ordre de grandeur pour les différentes parties de cette bague comme l'épaisseur du tube, la hauteur de la partie conique, la dimension et le nombre de vis. Nous avons pris pour variable la largeur et la hauteur de la bague tel qu'illustré dans la figure 2.8a. Le cas de charge considéré est une répartition homogène de la force de $20kN$ sur la surface interne oblique de l'anneau de couplage. La formule pour calculer la surface d'appuis est,

$$S = \frac{1}{\sqrt{2}}\pi H(D - 2W + \frac{1}{2}H). \tag{2.1}$$

1. http ://www.teconnex.com/product-aerospace/



(a) Dimensions optimisées dans la bague. (b) Chargement considéré.

FIGURE 2.8

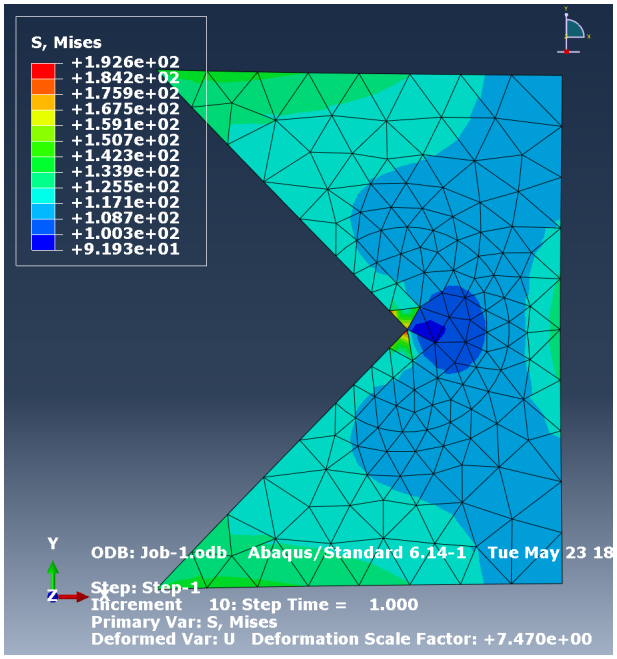
Ainsi, on peut estimer la pression de contact à l'interaction entre le coupleur et la bague,

$$\sigma = \frac{F}{S\cos(45°)}, \tag{2.2}$$

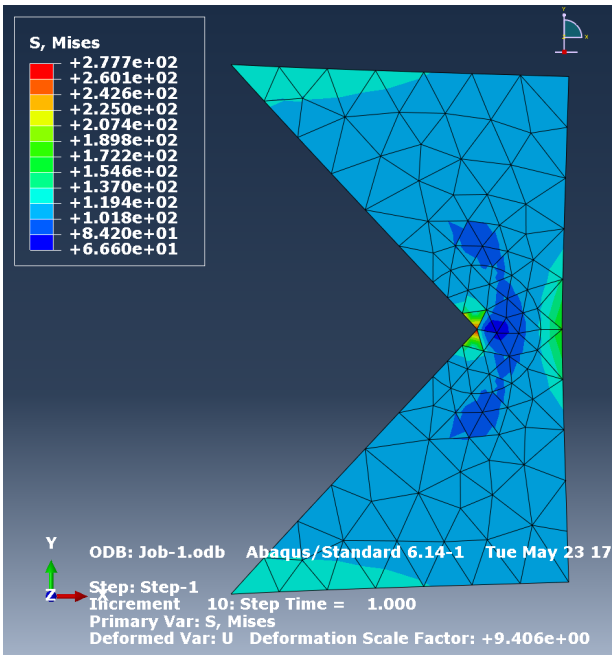
où $F = 20kN$ la force de traction sur le coupleur.

Suite à quoi différentes tailles ont été testées dans le but d'arriver à une dimension qui atteigne la limite élastique de la bague moyennant un facteur de sécurité de 2. Notons que l'analyse axisymétrique a tendance à améliorer la répartition des efforts et donc à réduire les contraintes. Un extrait des cas testés est présenté en figure 2.9.

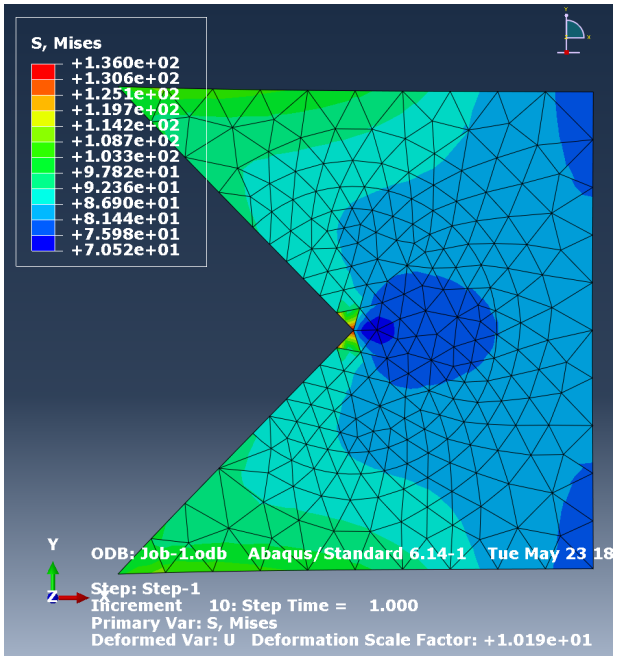
Des résultats plus détaillés sur cette analyse sont disponible dans le rapport de projet [3].



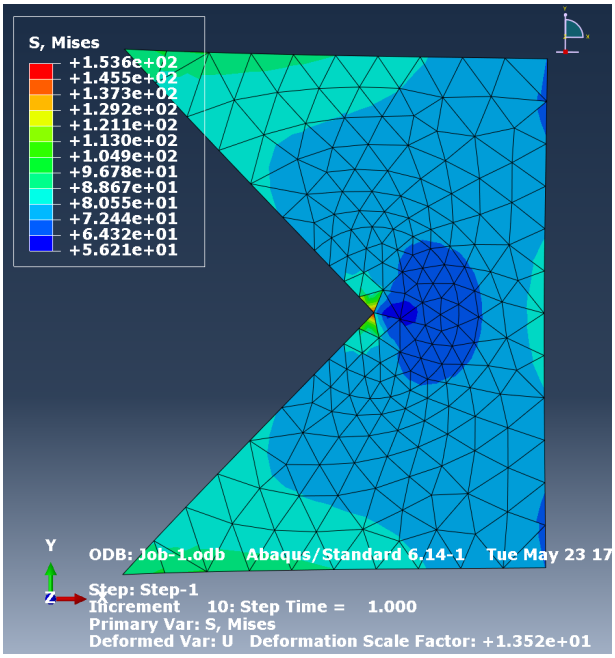
(a) W = 8, H = 10.



(b) W = 8, H = 12.



(c) W = 10 , H = 10.



(d) W = 10, H = 12.

FIGURE 2.9 – Résultats de l'optimisation des éléments de bague.

2.2.3 Assemblage

Après avoir effectué l'étude itérative d'optimisation sur la bague externe en axisymétrique, nous avons voulu vérifier l'assemblage complet en appliquant les contraintes réelles. En effet l'étude axisymétrique néglige plusieurs paramètres, ce qui a pour effet d'augmenter grandement la rigidité de l'assemblage. Elle ne prend pas en compte les trous de passage pour les vis, qui agissent comme concentrateurs de contrainte, ni les effets de décollement qui changent la répartition des contraintes à mesure que les quartiers de la bague externe s'ouvrent. Nous avons donc procédé à une analyse tridimensionnelle de l'assemblage avec des contraintes de contact non linéaire afin de modéliser au mieux la réalité et permettre la séparation des différentes surfaces.

Analyse 3D

Afin de modéliser au mieux la réalité, une première précontrainte à été appliquée sur la vis à l'aide d'une déformation thermique comme on peut le voir en figure 2.12a. L'objectif de cette précontrainte est d'assurer que les deux anneaux restent en contact tant que le chargement est limité à moins de 10kN de traction, ce qui représente la majorité du vol. Son effet est difficile à évaluer car elle a aussi comme effet négatif d'augmenter la contrainte finale dans les matériaux, mais cela garantit un meilleur plaquage des éléments entre eux et permet une meilleure répartition des efforts sur les surfaces en contact ce qui tendrait à faire diminuer la concentration de contrainte en certains points.

Ensuite nous appliquons une force de traction de $20kN$ sur les faces des anneaux qui seront collées avec les sections tubulaires et nous observons comme résultat final les contraintes équivalentes de Von Mises dans l'aluminium et l'acier en figure 2.11, ainsi que l'ouverture radiale de la bague externe, voir figure 2.10.

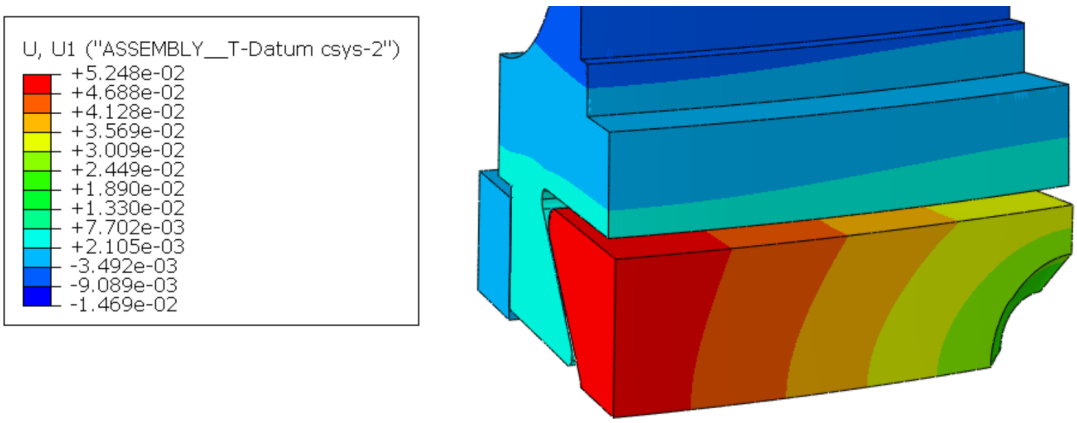
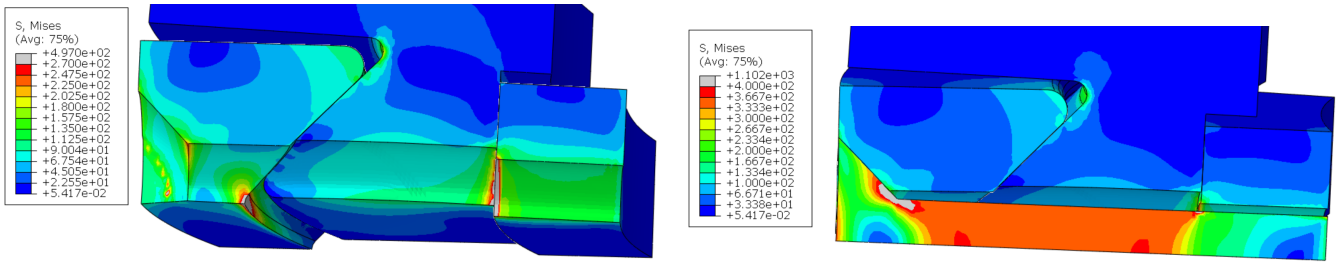


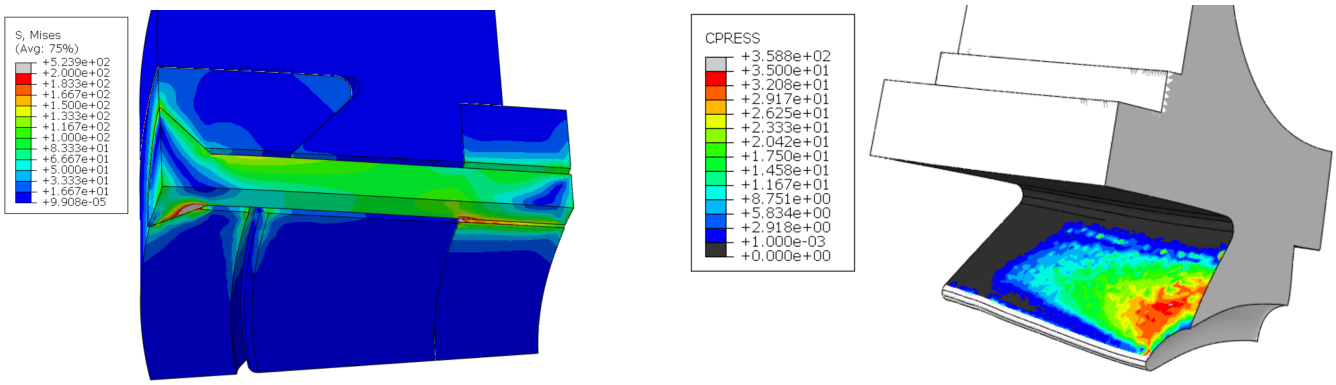
FIGURE 2.10 – Ouverture radiale de la Bague externe



(a) Contraintes dans l'aluminium

(b) Contraintes dans la vis en acier

FIGURE 2.11 – Contraintes dans l'assemblage



(a) Précontrainte thermique

(b) Pression de contact entre l'anneau et la bague externe

FIGURE 2.12 – Résultats supplémentaires

2.2.4 Résultats et conclusions

Pour l'analyse des pièces en aluminium, une limite artificielle égale à la limite élastique de $270MPa$ a été fixée sur la figure 2.11a. A la vue de cette image on remarque rapidement que les contraintes globales sont largement en dessous des limites fixées. Ainsi hormis quelques concentrateurs de contraintes sur les angles vifs et congés d'arête proches du trou de passage où l'aluminium risque de plastifier un petit peu, le reste de l'assemblage ne subit aucun dommage suite à ce cas de charge.

Pour ce qui est de la contrainte dans la vis sur la figure 2.11b, la limite à été fixée à $400MPa$, c'est à dire bien en dessous de la limite élastique de $640MPa$ d'une vis de qualité 8.8. Malgré cela on remarque que la contrainte ne dépasse cette valeur qu'en quelques points, là où les caractéristiques géométriques concentrent les efforts, ainsi qu'à l'endroit où la condition aux limites modélisant le filetage induit un effort concentré. Celle-ci restant tout de même inférieure à la limite élastique dans le deuxième cas. Comme la valeur maximale augmente très rapidement à proximité de la base de la tête de vis, il est difficile de tirer des conclusions sur la convergence de cette contrainte, mais une probable plastification pourrait apparaître dans cette région et il ne sera probablement pas superflu de changer les vis après chaque vol.

Finalement la dernière valeur qui nous intéresse est celle de la déformation radiale des quartiers de la bague externe, présentée à la figure 2.10. On observe que l'ouverture attendue des bagues est bien présente mais reste très faible, de l'ordre de $50\mu m$, ce qui implique que les efforts sont relativement bien transmis sur la surface

entre la bague et les anneaux. En effet, on remarque sur la figure 2.12b que la zone de pression de contact reste très importante, avec une augmentation graduelle de la valeur à mesure que l'on se rapproche de la vis.

Pour conclure, cette étude nous a donc permis de réduire la taille et l'encombrement du coupleur, tout en démontrant que notre système est fiable, mais il ne serait pas inutile de forcer une vitesse d'ouverture du parachute inférieure au $46m/s$ fixés par les règles de la compétition IREC, ceci afin de grandement réduire la traction maximale engendrée.

Des résultats plus détaillés sur cette analyse sont disponible dans le rapport de projet [4].

Quatrième partie

Expérimentation

Chapitre 1

Introduction

1.1 Pourquoi

Dans un projet tel que celui-ci, beaucoup de temps et d'efforts sont consacrés à la mise en place d'outils d'analyse et de dimensionnement. Ces outils reposent bien entendu sur une série d'hypothèses indispensables à une modélisation simple. Ces hypothèses permettent la mise en équations des phénomènes physiques régissant, dans notre cas, le vol et l'intégrité d'une fusée. Très couramment, celles-ci simplifient le problème au dépend de la validité du résultat. Il en existe différentes natures :

1. Hypothèses sur la géométrie du modèle.
2. Hypothèses sur la physique du modèle
3. Hypothèses sur la résolution des équations

A chaque fois que l'une d'entre elles est introduite, une partie de l'exactitude du résultat disparaît. Il faut donc se poser la question de la validation des résultats, la solution finales est-elle convenable ou complètement erronée ?

L'expérimentation existe pour répondre à ces questions. La meilleure méthode pour vérifier et valider des résultats provenant de simulations est l'essai et la mesure. Dans le cadre de ce projet, nous avons procédé à l'acquisition de données sur six décollages. L'ordre de grandeur pour la taille des fusées utilisées est de 2 $[m]$ et pour leur masse de 2 $[kg]$. Il s'agit de fusées de certification niveau 1, illustrées par la figure 1.1.

1.2 Objectifs

La réussite d'une expérience n'est pas chose facile, elle requiert de la minutie dans la préparation et l'exécution. La précaution et l'organisation sont primordiales. Il est important de préciser les objectifs et les attentes de cette expérience. Ceux-ci permettent la mise en place correct du dispositif de mesures. Nous avons pris en considération le cahier des charges suivant :

1. Résistance mécanique lors du vol et de la récupération
2. Autonome
3. Programmes/code accessibles

FIGURE 1.1 – Fusée pour certification niveau 1

4. Calibration valide des capteurs
5. Valeurs fiables
6. Prix raisonnable
7. Compacte, géométrie pour fusée d'un diamètre de 3 [*inch*]
8. Réalisation rapide

Il est important de spécifier ce que nous souhaitons mesurer. Pour pouvoir valider les modèles, les grandeurs physiques nécessaires sont listés dans le tableau suivant :

Grandeur	Utilisation dérivée
Temps de vol	
Accélération en X, Y et Z	Efforts
Rotation autour de X, Y et Z	
Vecteur champs magnétique	Orientation de la fusée
Pression	Altitude
Température	

Nous verrons que tous ces attentes n'ont pas toujours été validées. Cependant, le module réalisé fourni bien assez de mesures pour discuter de la validité des simulations.

Chapitre 2

Matériel & set-up

2.1 Capteurs

Il existe une large gamme de capteurs capable de mesurer ce que nous souhaitons. Il est même possible de trouver, dans le domaine des drones et des avions radio commandés, des circuits complets mesurant tout ce que nous souhaitons avec récupération des données. Il est évident qu'il est aisé de choisir cette option qui n'est pas la plus mauvaise en termes de qualité de données. Cependant, il nous intéressait de concevoir un module complet pour la beauté de l'exercice. Nous avons finalement décidé de trouver les capteurs qui offraient de bonnes possibilités tout en étant abordables pour des novices. Notre choix s'est porté sur la gamme proposée par Adafruit Industry. Ces capteurs ont l'avantage d'être utilisables par un micro-contrôleur et bon nombre de librairies sont disponibles.

2.1.1 Accéléromètre & Boussole

Le LSM303LDHC est un micro chip dans lequel est intégré à la fois l'accéléromètre et la boussole. Ils permettent pour l'un la mesure du vecteur accélération et pour l'autre le vecteur champs magnétique. Tous deux sont compatibles avec le protocole de communication I^2C présenté à la section 2.3.1. De ce fait, ils sont tous deux très faciles à utiliser. Ces deux instruments partagent la même carte électronique, mais il est cependant possible, via l'utilisation de libraire, de décomposer la lecture de leur signaux respectifs. Le circuit imprimé est illustré à la figure 2.1 [1].

Les caractéristiques sont présentées dans la tableau suivant :

Paramètres	
Accéléromètre	$\pm$ 16 $[g]$
Boussole	$\pm$ 1.3 $[gauss]$
Fréquence max	400 $[Hz]$
Température nominale	-40 à 80 $[°C]$

Notons qu'en réalité, les capteurs étant connectés en série via un bus de données, le protocole demande un temps d'adressage. Par conséquent, lorsque plusieurs d'entre

1. https ://www.adafruit.com/product/1714

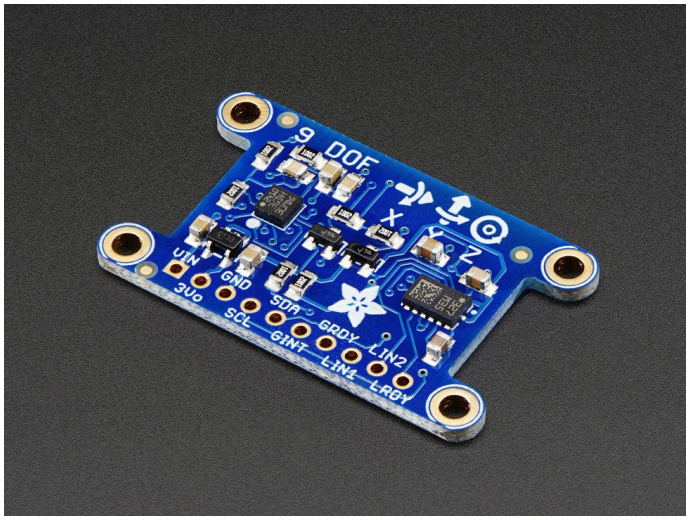


FIGURE 2.1 – Circuit imprimé avec LSM303LDHC & L3DG20H

eux sont lus, la fréquence d'échantillonnage réelle peut apparaître plus faible que la fréquence théorique.

2.1.2 Gyroscope

Le L3DG20H est le second composant électronique présent sur le circuit imprimé, présenté à la figure 2.1. Sa fonction est de mesuré une vitesse angulaire autour des 3 axes principaux. Une fois de plus, ce senseur est compatible avec le protocole I^2C. Il existe tout autant de librairies et l'utilisation est simple. Ses caractéristiques sont les suivantes :

Paramètre	
Vitesse angulaire	$\pm$ 250 $[°/s]$
Fréquence max	400 $[Hz]$
Température nominale	-40 à 80 $[°C]$

Évidemment, l'argument de fréquence présenté en section 2.1.1 est tout autant valable. Toutefois, même à basse fréquence, la quantité de mesures est suffisante pour la validation de nos simulations.

2.1.3 Baromètre

Le BMP180 est un second circuit imprimé qui sert de thermomètre et baromètre, il est présenté à la figure 2.2 [2]. Avec une pression mesurée, il est possible de déterminer une altitude absolue si l'on fournit la pression équivalente au niveau de la mer. Cependant, il est important de s'intéresser à la différence d'altitude lors d'un vol.

La température est quant-à-elle pas nécessaire pour des vols en basses altitude. Pour l'instant elle nous apporte juste un renseignement supplémentaire. Mais pour

2. https ://www.adafruit.com/product/1603

des vols à haute altitude, notamment lorsque la future fusée atteindra des apogées de 3000 $[m]$, il sera intéressant d'avoir ces données, notamment pour les calculs d'aéro, les propriétés de l'air changeant avec la température. Les caractéristiques du capteur sont les suivantes :

Paramètre	
Précision Pression Absolue	$\pm$ 1 $[hPa]$
Fréquence max	182 $[Hz]$
Température nominale	-40 à 85 $[°C]$

Ce capteur se branche en série sur le bus I^{2}C. Sa présence ralentit légèrement la fréquence d'échantillonnage de toutes les valeurs.

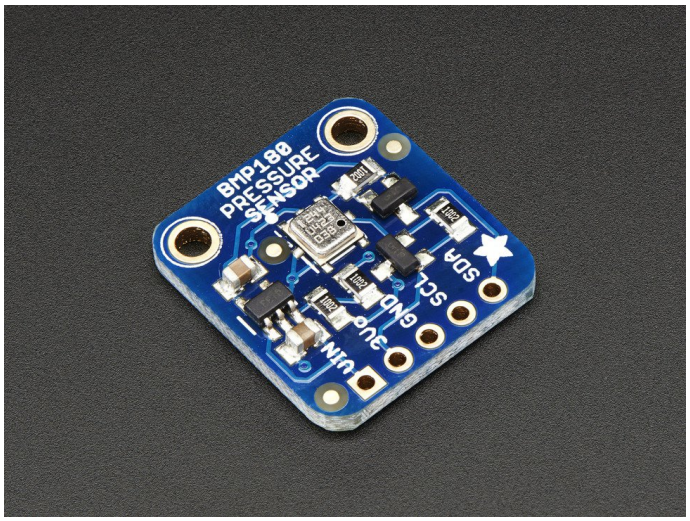


FIGURE 2.2 – Circuit imprimé avec BMP180

2.2 Microcontrôleur

Le microcontrôleur est le coeur du système. De son point de vue les capteurs sont des esclaves. Les valeurs ne sont fournies que si quelqu'un les demande. Et ce quelqu'un est un Teensy 3.5, présenté à la figure 2.3 [3]. Il s'agit d'un cousin du célèbre Arduino, plus puissant et avec plus de possibilités, notamment par ses dimensions. Ses caractéristiques principales sont listées dans le tableau suivant :

Paramètre	
Coeur	120 $[MHz]$ ARM Cortex-M4
Flash	512 $[KB]$
Ram	192 $[KB]$
EEPROM	4 $[KB]$
I/O	62 pins
Mirco SD slot	Oui
Dimensions	62.3 x 18 $[mm]$x$[mm]$.

3. https ://www.adafruit.com/product/3267

Mais pourquoi le Teensy 3.5 ? Après avoir étudié les possibilités, il s'avère que les Arduinos sont très présent dans le domaine des débutants. L'environnement de travail qu'ils offrent est un avantage, la compilation et le code sont très faciles. Cependant, les performances de ce micro-contrôleur sont basses. Pour obtenir un processeur 32 bits, il faut s'intéresser à l'Atmega 2560, qui est de trop grande taille pour notre utilisation.

C'est là que le Teensy 3.5 intervient. Sa petite taille et sa puissance de calcul font de lui un candidat favorable à notre utilisation. Ses dimensions sont parfaites pour lui permettre de prendre place dans l'un de tubes de nos fusées de certification de niveau 1. Ajouté à cela, la board Teensy 3.5 est disponible sur l'environnement Arduino, environnement très discuté sur les forums, avec une grande aide en ligne. Avec tout ceci, le Teensy permet une réalisation rapide d'un module de mesure.

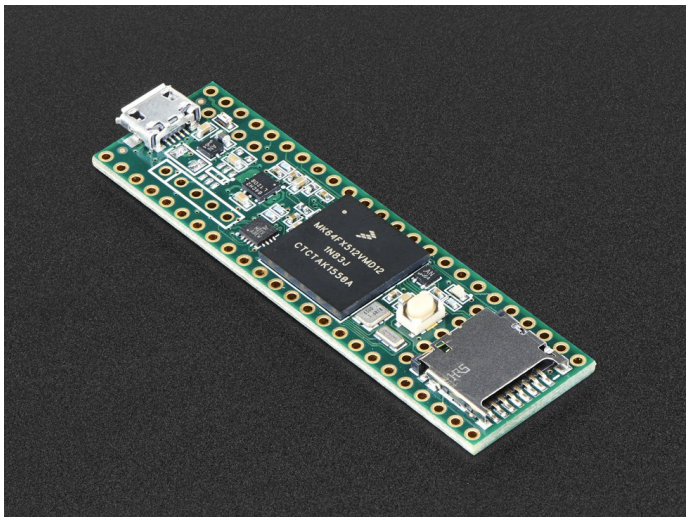


Figure 2.3 – Microchip Teensy 3.5

2.3 Circuit

Posséder les capteurs et le Teensy est une chose, les faire communiquer en est une autre. L'objectif est de les lier au mieux pour gagner du temps lors des mesures. Il faut créer la circulation. Comme mentionné plusieurs fois, nous utilisons le protocole I^2C, détaillé en section 2.3.1. Le set-up et la réalisation du module sont présentés en section 2.3.2.

2.3.1 Protocole de communication

La manière la plus efficace de communiquer est d'établir un protocole d'envoi et de réception entre les différents composant du bus de données. Un bus est simplement un jeu de connexions qui lient tous les éléments participants à l'échange de données.

Pour le module de mesures, nous avons choisi le protocole I^2C, protocole très réputé dans le monde des microcontrôleurs. Son fonctionnement est assez simple et les librairies Adafruit proposent suffisamment de fonctions pour la réception des

valeurs. Ainsi il est possible de demander à recevoir un paramètre sans comprendre comment l'échange s'opère. Prenons quand même le temps d'expliciter son fonctionnement.

Ce protocole est basé sur la présence d'un maître et d'un ou plusieurs esclaves. On comprend par là que le maître demande lorsqu'il souhaite obtenir une valeur et l'esclave lui envoie sans tarder. Le lien est fait par une paire de lignes. La première se dénomme SDA, Serial Data Line et la seconde SCL, Serial Clock Line, illustrées dans la figure 2.4 [4]. La ligne SCL est une horloge qui impose le rythme d'écriture et qui permet de synchroniser le lecture d'un bit au moment souhaité. Deux séquences de bits reconnaissables signalent le début et la fin d'une transmission, Start & Stop, visible sur la figure 2.5 [5]. L'adresse de l'intéressé est au préalable envoyé sur la ligne, il s'agit d'un autre canevas.

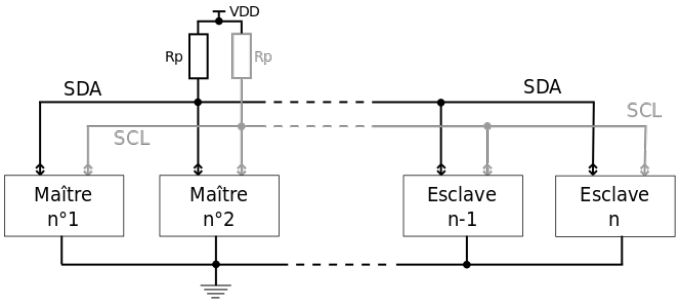


FIGURE 2.4 – Lignes qui composent un bus I^{2}C

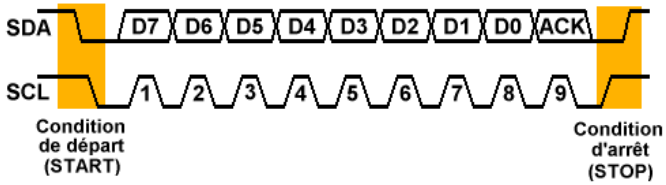


FIGURE 2.5 – Séquence typique d'envoi d'un byte.

Finalement, la manière dont sont physiquement reliés les chips et les séquences qui sont utilisées composent l'ensemble d'un protocole.

2.3.2 Branchements

Pour réaliser le module, nous savons que les senseurs et le microcontrôleur sont compatibles et peuvent communiquer par un bus SDA & SCL. Nous présentons le schéma de branchement à la figure 2.6. On reconnaît la tension d'alimentation de 5 $[V]$ (Rouge) fournie par le régulateur et la terre (noir). Ces deux connexions sont nécessaires et doivent être les mêmes pour tous les éléments du montage. Sur la figure 2.6, on observe deux lignes de plus. Il s'agit des lignes de bus SDA et SCL décrites en section 2.3.1. Finalement, tous les chips, maîtres et esclaves, sont branchés en

4. *https : //upload.wikimedia.org/wikipedia/commons/thumb/a/a0/I2C$_A$rchitecture$_2$.svg/666px*
5. *https : //www.aurel32.net/elec/i2c2.png*

parallèle. La ligne SCL, représentée en orange, provient du pin 19 du Teensy et la ligne SDA, représentée en verte, est issue du pin 18. Une carte informant la position des pins et leur fonction est présentée en figure 2.7 [6]. On remarque la tension d'entrée V_{in} et la terre GND.

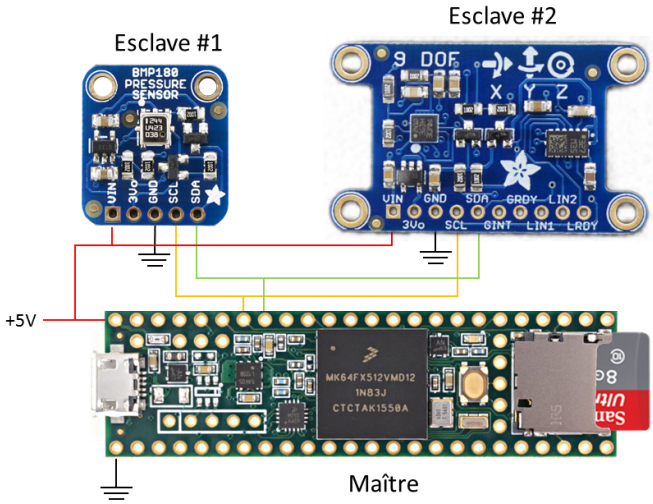


FIGURE 2.6 – Schéma de connexion

Dans le module embarqué, nous avons rajouté un bouton, monté avec une résistance pull-up. Celui-ci nous permet de démarrer la mesure et d'arrêter celle-ci de manière à ce qu'on puisse retirer la carte micro SD sans que par malchance on soit en train d'écrire dessus. Les risques d'endommager le fichier sont ainsi fortement réduits. La photo 2.8 [7] illustre le montage final du module. Il est conçu pour prendre place dans une fusée d'un diamètre de 3 $[inch]$ ($\sim$ 80 $[mm]$). Une version adaptable existe pour les fusées d'un plus gros diamètre. On remarque les deux capteurs juste en dessous du régulateur de tension. Le bouton est l'élément jaune positionné sur le bas à gauche. Juste au-dessus se trouve le Teensy avec sa carte micro SD pour la sauvegarde des données. Finalement, une batterie se situe à l'arrière de la plaque à trous sur une seconde tôle en alu, rigidifiant l'entier de la structure.

2.4 Programmation

Acheter les bons capteurs, brancher correctement ces capteurs et mettre sous tension ne suffisent pas à lire des données. Il faut à présent programmer le module pour qu'il remplisse correctement sa tâche. Pour s'y prendre, il y a deux phases. La première permet de contrôler le Teensy, on code les instructions nécessaires à la récolte de données. La seconde permet de vérifier les mesures et dans le cas échéant, de faire la calibration. Il est notamment obligatoire de contrôler avant chaque vol le champ magnétique car des champs ambiants existent partout et sont tous différents.

6. https ://www.pjrc.com/teensy/pinout.html
7. Jonathan Burkhard Photography

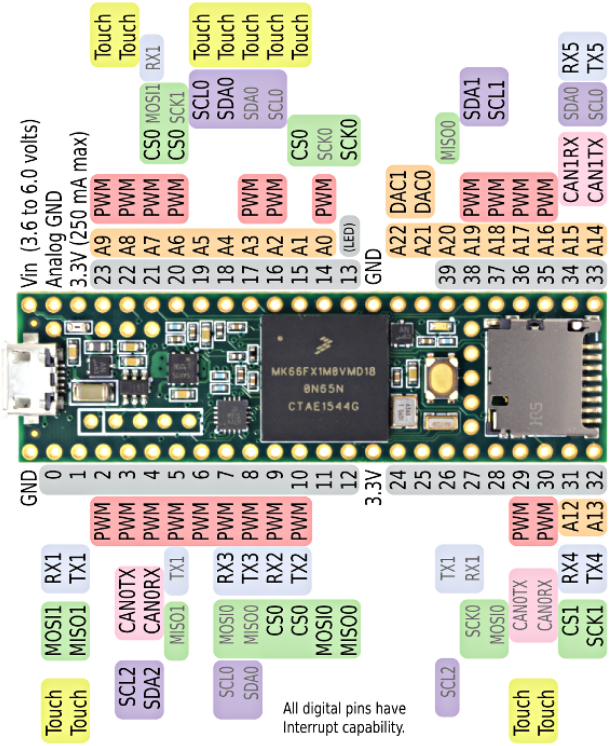


FIGURE 2.7 – Carte des pins I/O du Teensy 3.5

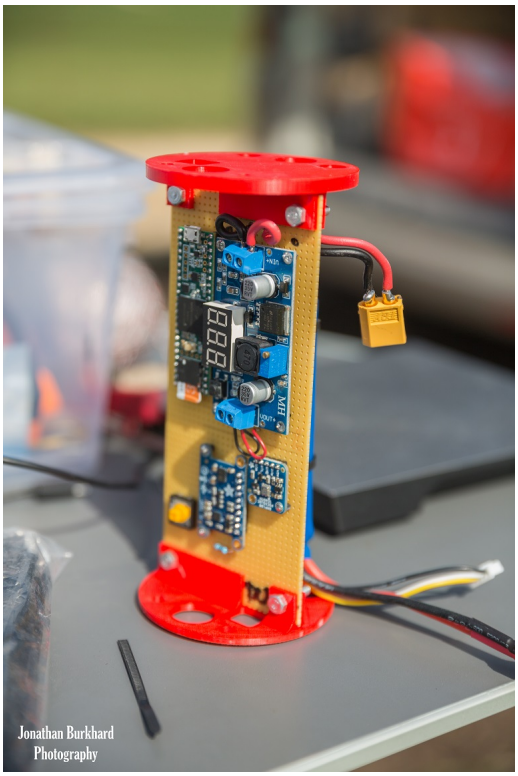


FIGURE 2.8 – Module de mesure opérationnel

2.4.1 Arduino & Teensyduino IDE

Pour compiler nos instructions sur le Teensy, nous utilisons l'environnement créé par Arduino. Celui-ci présente une interface graphique compréhensible et facile d'utilisation. Un second avantage réside dans la présence d'add-ons pour notre microcontrôleur. On peut alors se concentrer uniquement sur le code sans se soucier de la compilation et de l'accès aux ports du Teensy, qui peut s'avérer difficile puisque chaque marque et chaque modèle portent des noms de registres différents. Cet ajout s'appelle le Teensyduino. Celui-ci apporte toute la configuration nécessaire à l'environnement pour la compilation sur cette carte, qui s'est avéré très utile pour réaliser rapidement le module.

Les capteurs Adafruit, présentés en section 2.1, possèdent tous une libraire compatible Arduino et donc par défaut Teensyduino. Nous n'allons pas présenter celles-ci, dans la mesure où elles sont complexes et leur utilisation se restreint à un appel de valeurs. Il est tout de même important de préciser que certain registres ont étés changés pour sélectionner les fréquences, plages de mesures et précisions souhaitées, de telles manière à obtenir les spécifications proposées dans les caractéristiques des capteurs, présentées en sections 2.1.

Du point de vue du microcontrôleur, il n'existe aucun programme qui réalise l'entier de ce que nous souhaitons. Nous avons crée l'ensemble du programme, tout en se basant sur les exemples proposés avec les librairies. L'idée est de partir sur une machine d'état qui balaie les différentes étape en boucle. On est obligé de le faire de cette manière plutôt que d'utiliser une boucle infinie car on souhaite rencontrer un état de départ et un état d'arrivée. Ce dernier protège la carte micro SD lors de son retrait. Les valeurs mesurées sont sauvées dans un buffer d'une longueur compatible avec l'écriture en bloc typique des cartes micro SD. Le programme traverse les différentes étapes suivantes :

Etat	Suivant	Condition	Caractéristique
Attente	Mesures	Push Button	Pente montante
Mesures	Écriture	Buffer Plein	Baromètre utilisé 1/30
Écriture	Mesure ou Attente	Fin écriture ou Push Button	

Le code est présenté en annexe C. La machine d'état est déclarée et initialisée aux lignes 86 à 89. Elle est ensuite utilisée dans la boucle infinie, lignes 108 à 174. Dans le cas pratique, comme la fréquence du baromètre est plus faible que celle de l'accéléromètre, gyroscope et boussole, il bloque la vitesse d'acquisition. Pour contourner ce problème, un compteur est démarré lors de l'initialisation. Celui-ci permet de compter et indiquer lorsqu'on souhaite rafraîchir le baromètre. Il suffit d'indiquer un rapport voulu entre la nouvelle valeur et la copie. On peut régler ce paramètre lors des définitions à la ligne 27, définie par Bmp_Measure.

Comme nous le savons déjà, les valeurs sont enregistrées sur la carte micro SD. Pour éviter la perte de temps dû à la procédure d'écriture sur carte et ainsi optimiser la cadence de mesure, nous regroupons une grosse quantité de données avant de les enregistrer. Il s'agit des états Mesures et Écriture. Le groupe de données est un tableau à 1 dimension dans lequel sont stockées les valeurs les unes après les autres. Pour permettre une bonne relecture, il faut préciser un protocole d'écriture.

Comme il y a 12 paramètres à récolter, une mesure se retrouve toutes les 12 cases du tableau. La boucle for, lignes 111-146, incrémente une variable qui balaie le tableau le temps de parcourir toute la boucle. Multipliée par le nombre de paramètres, Nb_Param = 12, on crée un saut de bloc composé de 12 positions à chaque itération. C'est à l'intérieur de cette boucle for que l'on dégrossit le bloc. Chaque paramètre est affilié à une position relative au bloc, par exemple la mesure du gyroscope autour de x est à la position 7 [8]. La procédure est illustrée dans la figure 2.9.

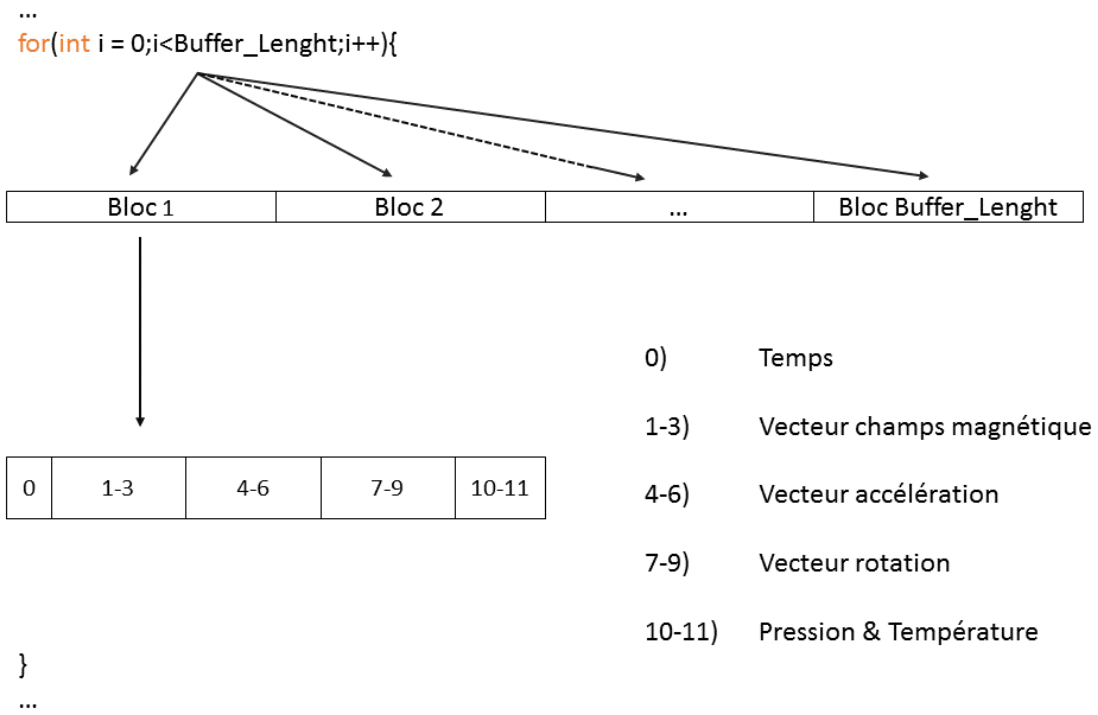


FIGURE 2.9 – Décomposition du buffer de données

Une fois que le tableau a atteint la valeur limite de 16 $[KB]$, il est mis en mémoire bit par bit, ligne 164. Chaque valeur mesurée est un float de 4 bytes. Le fichier résultant ne contient pas d'extension et doit être lu bit par bit. Cette partie est réalisée à l'aide de MATLAB.

2.4.2 MATLAB

Avec l'aide de MATLAB, on peut calibrer les capteurs. Les valeurs ne sont pas encore exactes, il n'est tout du moins pas possible de le savoir. Cependant, il faut pouvoir lire les données. Le protocole présenté en section 2.4 est compatible avec MATLAB. Le bout de code suivant permet de lire notre fichier :

```
1 n = 12;                                  % Nb de parametres mesures
2 fid = fopen('VOL1','rb');                % Fichier que l'on souhaite lire (dossier courant)
3 data = fread(fid,[n inf],'real*4');      % Enregistree dans matrice n*Nombreacquisition ([n inf])
4 fclose(fid);                             % Ferme le fichier
```

8. En C, les positions commencent à 0.

Une fois le code lancé, il y a à disposition une matrice contenant un paramètre différent sur chaque ligne. Le temps entre chaque mesure est déterminée par le temps d'acquisition des données. Il peut différer par moment car on n'accède pas toujours aux résultats du baromètre.

Les calibrations qui nous intéressent sont celles de la boussole et de l'accéléromètre. Elles sont faciles car les vecteurs terrestres de ces deux quantités sont fixes. Comme la référence est connue, on peut essayer de les autoréférencer ; pour chacune d'entre elle, on fait l'hypothèse que pour chaque composante il y a un offset et un gain de correction. Dans la majorité des cas, la quantité de mesures est suffisante pour déterminer toutes les inconnues.

Pour la boussole, on utilise une fonction MATLAB qui optimise les paramètres d'entrées de manière à représenter au mieux les données acquises avec une forme souhaitée. Dans notre cas, nous mettons en forme une sphère d'un rayon R et de centre (x(1),x(2),x(3)). Si on réfléchit, l'amplitude du vecteur terrestre est constante. Cela veut dire que si on bouge le module, le vecteur bouge relativement au capteur. Or il est possible qu'un champ magnétique soit créer par le circuit imprimé. Dans ce cas on observerait une sphère décentrée, ce qui correspond bien à notre hypothèse. La portion de code suivante procède à la calibration :

```
MAG = [data(2,:)' data(3,:)' data(4,:) '];     % Isole le champs magnetique du set de donnees
Fun = @(x) errorNorm(x,MAG);                   % Appelle la fonction qui décrit la sphere et ses parametres
[C,RESNORM,RESIDUAL,EXITFLAG,OUTPUT,LAMBDA,JACOBIAN] = lsqnonlin(Fun,[0,0,0,1]); % Fonction d'optimisation des parametres
```

Où la fonction errorNorm est déclarée comme suit :

```
function err=errorNorm(x,data)
    err=zeros(size(data));
    C=[x(1),x(2),x(3)];
    R=x(4);
    for i=1:size(data,1)
      err(i)=(norm(data(i,:)-C)-R)/R;
    end
end
```

Ce bout de code retourne les paramètres discutés plus haut. Pour corriger la mesure, on introduit un offset sur chaque composante du champs magnétique comme on peut le remarquer dans l'annexe C aux lignes 31 à 33. Pour vérifier une seconde fois que la calibration effectuée est bonne et que la sphère apparaît dorénavant à l'origine, on procède à une seconde séquence de mesures. Les nouvelles composantes du champs magnétiques sont regroupée dans un graphe. On vérifie leur validité avec une sphère, seul le rayon est à déterminer. Un exemple de résultat est présenté à la figure 2.10. La calibration est correcte si la majorité des points sont sur la surface sphérique.

Pour l'accéléromètre, il existe une manière plus simple. L'idée consiste à calibrer un seul axe à la fois. Un axe doit être aligné tour à tour avec la verticale, ou plutôt la gravité. On peut alors mesurer la moyenne des valeurs et trouver le gain. Le bout de code qui suit retourne les gains. Attention, il faut procéder à 3 mesures distinctes et ne jamais bouger le module pendant celle-ci. Une fois que les gains sont déterminés, il faut les intégrer au code du Teensy. C'est fait au lignes 34 à 36.

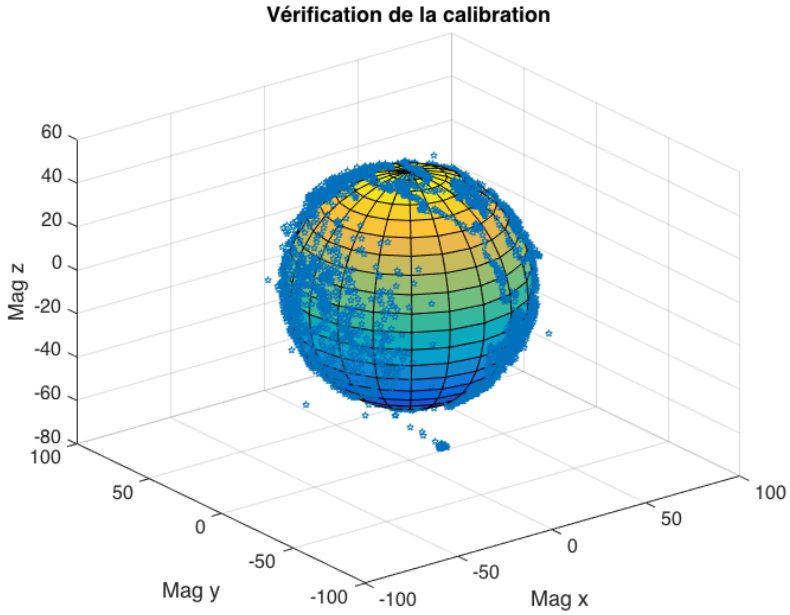


FIGURE 2.10 – Vérification de la calibration du champs magnétique

```
xaccel = data(5,:);          % Selectionne les accelerations en x
XX = mean(xaccel);           % Moyenne
Gx = 9.81/XX;                % Calcul le gain
yaccel = data(6,:);          % Selectionne les accelerations en y
YY = mean(yaccel);           % Moyenne
Gy = 9.81/YY;                % Calcul le gain
zaccel = data(7,:);          % Selectionne les accelerations en z
ZZ = mean(zaccel);           % Moyenne
Gz = 9.81/ZZ;                % Calcul le gain
display(sprintf('Gain acceleration x:\t%d', Gx)); % Affichage
display(sprintf('Gain acceleration y:\t%d', Gy)); % Affichage
display(sprintf('Gain acceleration z:\t%d', Gz)); % Affichage
```

Chapitre 3

Résultats

Dans le cadre de ce projet, nous avons procédé à deux journées de lancement. Lors des 2 journées, les fusées concernées étaient de type certification niveau 1. Le module présenté au chapitre 2 a volé les deux jours jusqu'à sa destruction, lors d'une récupération mal négociée. Voici les observations que nous avons pu faire.

3.1 Lancement 1, 1 Avril 2017

3.1.1 Fusées

Au total, le module de mesure a pris place dans 4 fusées. Elles sont toutes construites par l'équipe et proviennent de kit de construction à faire soi-même.

Nom	Calibre $[mm]$	Longueur $[mm]$	Moteur
D-Region Tomahawk, phenolic	80	1800	H123W
D-Region Tomahawk, PVC	80	1800	H123W
Miranda, PVC	80	1600	H148R
Super DX3, phenolic	101.6	1680	H148R

3.1.2 Moteurs

Toutes les fusées présentées peuvent accueillir un moteur de 38 $[mm]$. Pour ces vols de certification, nous avons sélectionné les moteurs suivants :

Nom	Compagnie	Poids consommable $[g]$	Impulsion totale $[Ns]$
H123W	AeroTech	129	223.6
H148R	AeroTech	122	206.0

Leurs spécifications sont très semblables et leurs courbes de poussée sont présentées dans les figures 3.1 [1] et 3.2 [2].

1. http ://www.thrustcurve.org/simfilesearch.jsp ?id=9
2. http ://www.thrustcurve.org/simfilesearch.jsp ?id=482

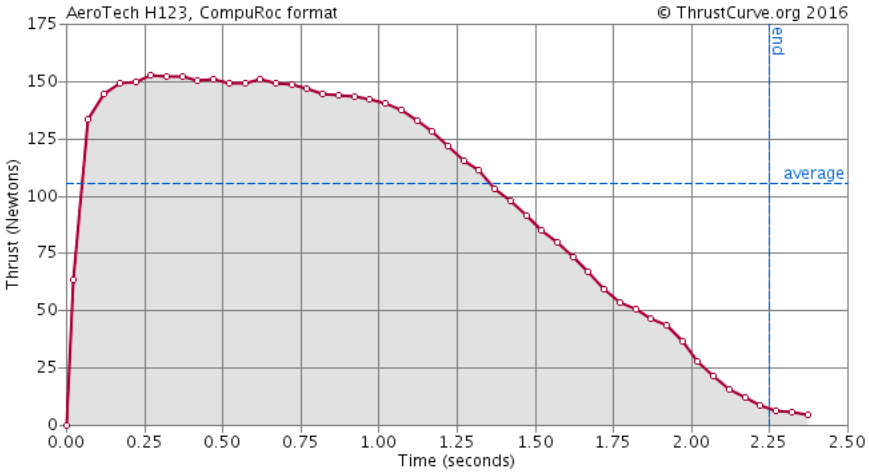


FIGURE 3.1 – Courbe de poussée, H123W

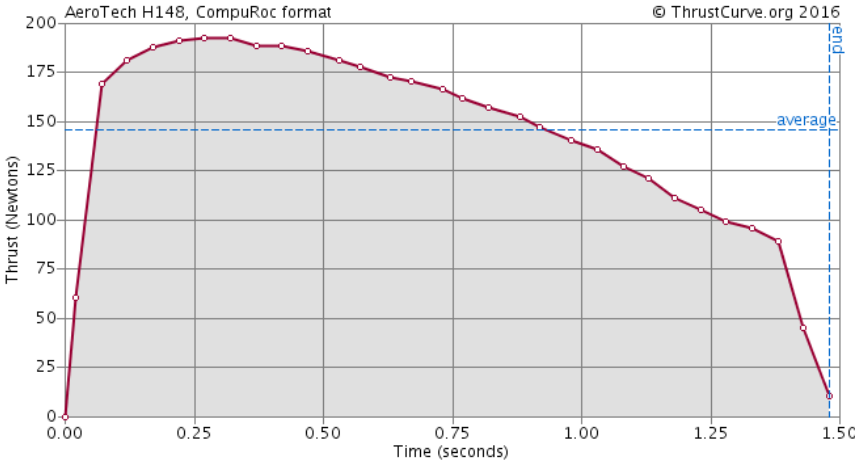


FIGURE 3.2 – Courbe de poussée, H148R

3.1.3 Altitude

Lors du premier jour de lancement, le module a mesuré instantanément l'altitude en se basant sur les mesures de pression actuelle et au niveau de la mer. Cette fonction qui est directement intégrée au code et a été retirée à cause de son temps d'exécution trop long et remplacée par du post-traitement. Un bug, non résolu, affichait périodiquement des zéros. Les courbes qui seront présentées ont subi un traitement pour retirer ce défaut sans aucune dégradation des valeurs. Pour valider nos mesures, nous allons corréler nos résultats avec un simulateur connu, OpenRocket. Voici les valeurs obtenues lors de ces simulations :

Fusée	Estimation d'apogée $[m]$
D-Region Tomahawk, phenolic	380
D-Region Tomahawk, PVC	390
Miranda, PVC	380
Super DX3, phenolic	299

Nous pouvons comparer avec les apogées mesurées. Les courbes d'altitudes sont

présentées dans la figure 3.3. A première vue, toutes les courbes sont normales. C'est-à-dire qu'elles présentent une première phase vol balistique puis une seconde phase avec l'ouverture du parachute et une descente linéaire.

On remarque en figure 3.3a un pic lors de l'ouverture du parachute. Il s'explique par la mauvaise attache du cône au corps de la fusée. Lors de l'explosion, l'expansion des gaz d'éjection augmente un court instant la pression, faisant artificiellement baisser l'altitude. Le cône est par conséquent poussé hors de son logement jusqu'à tension de la corde qui le retient. Pendant ce temps la pression chute et l'altitude semble augmenter. On peut conclure que ces hypothèses sont vraies car la D-Region Tomahawk en phénolique était la seule fusée avec un cône à moitié sorti lors de la récupération.

Finalement, les apogées mesurées sont très proche de la simulation. On note une erreur globale inférieure à 2% si les simulations pouvaient être considérées comme exactes. Les conditions de vol réelles ne sont pas idéales, vent, humidité, pression ... peuvent fortement différer. On conclut que les mesures d'altitude peuvent être considérées comme valides. Voici les résultats :

Fusée	Apogée Mesurée $[m]$
D-Region Tomahawk, phenolic	372
D-Region Tomahawk, PVC	392
Miranda, PVC	392
Super DX3, phenolic	295

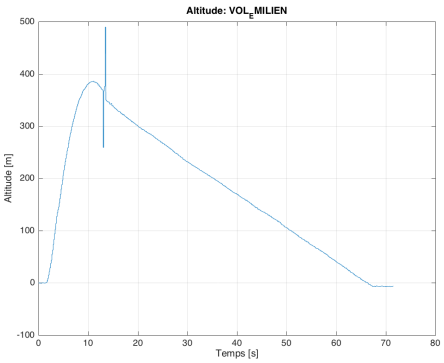
3.1.4 Accélération

Les mesures d'accélération sont très enrichissantes dans le sens où elles renseignent sur les efforts mécaniques que peut subir la structure. Pour bien comprendre les graphes, il faut définir un système de coordonnées. Ce n'est pas un système absolu car il est fixe avec la fusée qui bouge et tourne. Le repère présenté en figure 3.4 est considéré comme notre système de référence. L'axe x indique la direction de vol quant aux deux autres, ils sont radiaux. En observant la symétrie d'une fusée, on peut s'attendre à observer le même genre de résutat dans ces deux directions.

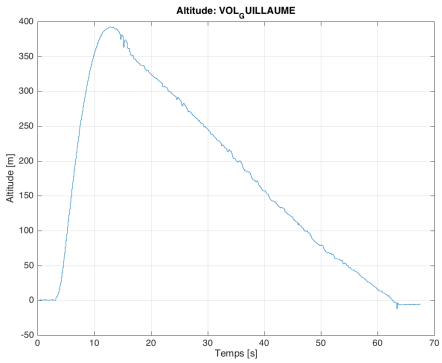
Pour supprimer le bruit présent dans nos données, nous avons recours à un filtre de Butterworth. Il permet de lisser les courbes sans pour autant affecter l'information nécessaire à l'interprétation des mesures. Il ne faut toutefois pas abuser de l'effet lisseur.

Aux figures 3.5, 3.6 et 3.7 sont présenté les 3 composantes du vecteurs d'accélération. Il s'agit de la fusée D-Region Tomahawk en phénolique. Seuls les moments critiques du vol sont reportés. Un vol balistique sans poussée ni parachute n'a pas d'intérêt. Ces moments critiques sont la phase de poussée, l'ouverture du parachute et le choc au sol.

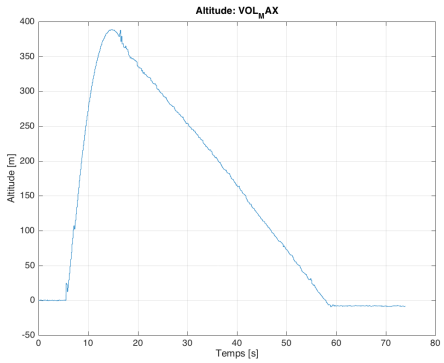
On peut observer que l'accéléromètre sature à 2 $[g]$ et ceci dans toutes les directions. La saturation est bien visible lors de la poussée initiale, sensée être proche des 7 $[g]$. Il s'agit d'une erreur de programmation. Un registre doit être changé pour accéder à la plage complète de ± 16 $[g]$. Malgré les résultats erronés, on observe des



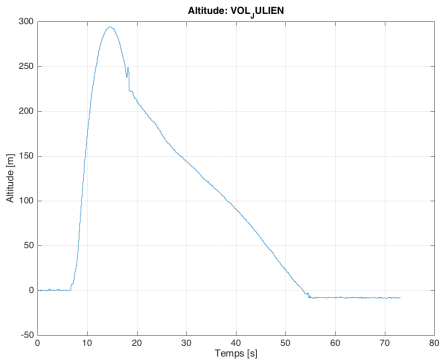
(a) D-Region Tomahawk, phenolic



(b) D-Region Tomahawk, PVC



(c) Miranda, PVC



(d) SUPER DX3, phenolic

FIGURE 3.3 – Courbes d'altitude mesurées

efforts radiaux importants lors de l'ouverture du parachute. Il est consigné qu'il sera intéressant de vérifier lors du second jour de vol si les accélérations dépassent 2 $[g]$.

Il est encore important de remarquer qu'en phase de poussée l'accélération est quasi unidirectionnelle. On peut toutefois observer une excitation en z après une seconde de poussée. Cet instant coïncide avec la sortie du rail de guidage. Lors de cette journée de lancement, le vent était constant et intense. On peut donc conclure que les efforts induits par une correction de trajectoire due au vent sont non négligeables. Cependant, ils restent inférieurs aux efforts produits lors de la décélération à l'ouverture du parachute.

3.1.5 Champs magnétiques

Le champs magnétique terrestre est constant dans une parcelle raisonnable du globe. La boussole mesure en continu le vecteur magnétique. On utilise MATLAB pour redresser le vecteur initiale dans le référentiel de la fusée. Aussitôt que l'angle varie entre la référence et la nouvelle mesure il est possible d'orienter la fusée dans l'espace. Avec l'aide de MATLAB, nous avons créé une vidéo qui retrace l'orientation de la fusée Super DX3 en fonction de l'altitude. La vidéo est disponible sur la plateforme YouTube, le lien est donné à la figure 3.8 [3].

3. https ://www.youtube.com/watch ?v=kzFYA-GEzfA

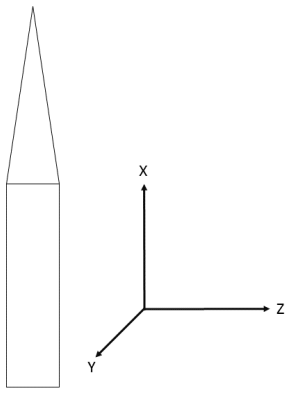


FIGURE 3.4 – Axes de références pour les mesures

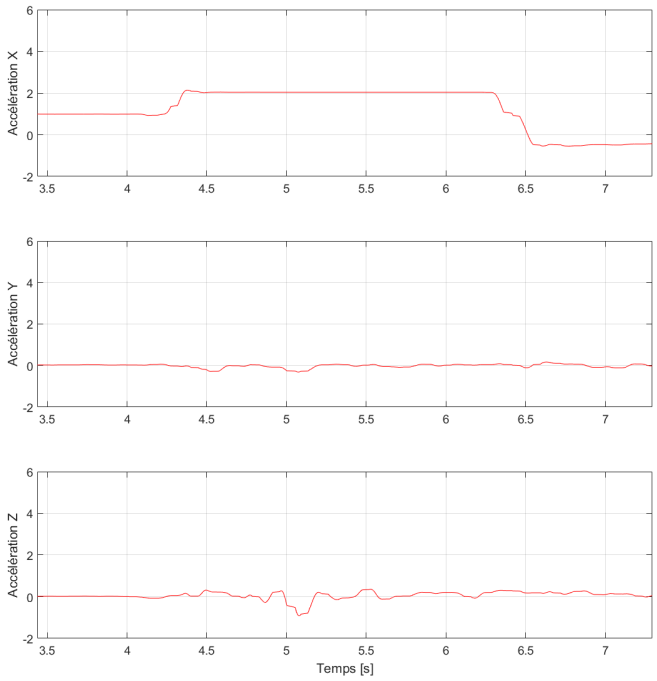


FIGURE 3.5 – Accélération en phase de poussée, D-Region Tomahawk

3.2 Lancement 2, 20 Mai 2017

3.2.1 Fusées

Lors du deuxième jour de lancement, seulement deux fusées ont pu embarquer le module de mesure. Au final, uniquement une seule d'entre elle propose tous les résultats escomptés. La seconde s'étant malencontreusement écrasée avec l'électronique. Nous avons choisi de présenter en section 3.1.4 la fusée D-Region Tomahawk phenolic pour pouvoir la comparer à nouveau avec les données de la seconde journée.

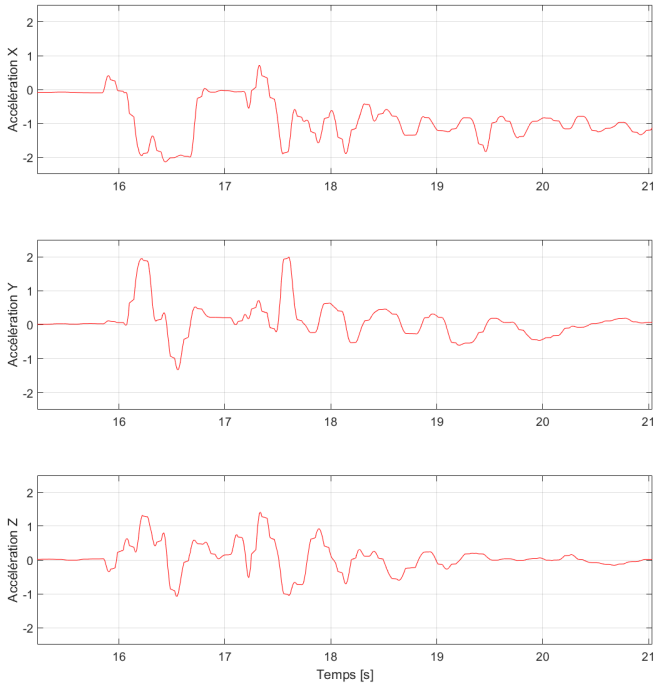


FIGURE 3.6 – Accélération à l'ouverture du parachute, D-Region Tomahawk

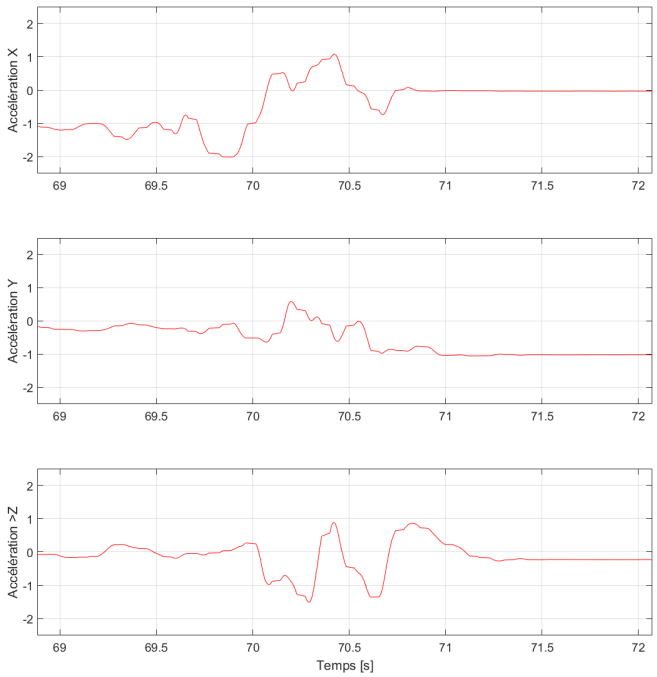


FIGURE 3.7 – Accélération au contact du sol, D-Region Tomahawk

FIGURE 3.8 – Lien vers les résultats en vidéo.

Nom	Calibre $[mm]$	Longueur $[mm]$
D-Region Tomahawk, phenolic	80	1800
Miranda, PVC	80	1600

3.2.2 Moteurs

Il s'agit des mêmes moteurs.

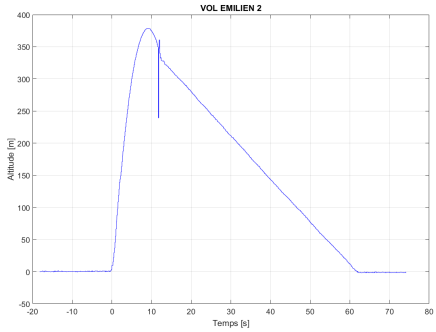
Nom	Compagnie	Poids consommable $[g]$	Impulsion totale $[Ns]$
H123W	AeroTech	129	223.6
H148R	AeroTech	122	206.0

3.2.3 Altitude

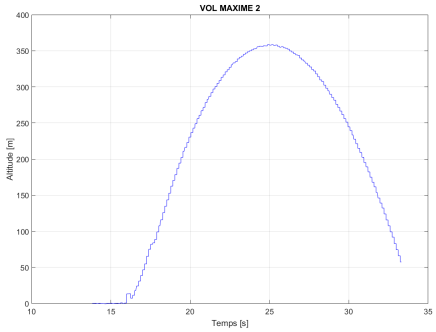
Les fusées et les moteurs sont respectivement les mêmes, il est évident que les apogées estimées par OpenRocket sont identiques au premier vol (même simulation). Les mesures d'altitudes sont ensuite présentées à la figure 3.9. On reporte l'estimation et la mesure de l'apogée dans le tableau suivant :

Fusée	Estimation d'apogée $[m]$	Mesure d'apogée $[m]$
D-Region Tomahawk, phenolic	380	379
Miranda, PVC	380	360

Tout d'abord, on remarque une erreur relative élevée pour la fusée Miranda. Il n'y a pas eu d'événement marquant pouvant expliquer ce résultat. Il s'agit de la pire mesure si l'on en croit la simulation d'OpenRocket, avec une erreur de 5.2 %. A l'opposé, les résultats de la D-Region Tomahawk présentent la meilleure précision des six vols. Dans l'ensemble, les évaluations de l'apogée peuvent être considérées comme valables.



(a) D-Region Tomahawk, phenolic



(b) Miranda, PVC

FIGURE 3.9 – Courbes d'altitude mesurées

3.2.4 Accélération

Après les corrections apportées au module de mesure, il dorénavant possible de mesurer des composantes d'accélération pouvant atteindre ± 16 $[g]$. Plusieurs tests ont été menés pour s'assurer de la validité des mesures. Nous nous proposons de commencer par l'accélération axiale. Proposée en figure 3.10, la composante axiale de l'accélération présente une forme reconnaissable. En effet, les courbes de poussées illustrées en section 3.1.2 présentent une forme identique. Le graphe 3.10 contient la courbe de poussée du moteur correspondant et sa marge de $\pm$ 20%. On peut observer que les deux formes sont quasi identiques.

Cela est possible uniquement si la masse de la fusée ne varie pas trop durant la phase d'accélération. Cette hypothèse est vérifiée. La masse perdue est de l'ordre de 0.1 $[kg]$ alors que la fusée pèse 2 $[kg]$. La différence de masse est négligeable. Ainsi les possibles distorsions n'ont pas lieu d'être.

Il s'agit d'un résultat intéressant. Celui-ci permet une validation de la mesure d'accélération dans le sens où les bonnes mesures sont effectuées au bon moment. Il est temps de s'intéresser aux accélérations axiales. Nous avons vu en section 3.1.4 que nos mesures saturaient à 2 $[g]$, regardons ce qu'il en est avec une plage de mesure étendue à 16 $[g]$.

Par argument de symétrie, nous traiterons les accélérations en Y et Z en même temps, celle-ci doivent théoriquement présenter des comportements identiques. Nous souhaitons nous concentrer sur l'ouverture du parachute, dernier moment critique de la phase de vol. Les mesures sont présentées dans la figure 3.11. On observe un pic élevé d'accélération au moment de l'ouverture du parachute. La valeur est aussi importante que lors de l'accélération initiale. C'est un résultat étonnant qu'il faut prendre avec des pincettes.
Lors de l'expérience, le module se trouvait dans la partie supérieure de la fusée, partie la plus légère composée uniquement du cône et d'un tube vide. Il est fort possible que ce morceau balance plus aisément que l'autre et que l'accélération soit plus élevée que pour le reste de la structure. Comme on s'intéresse en réalité à l'accélération du tube de la fusée il faudrait reproduire l'expérience avec un capteur dans la partie inférieure de la fusée.

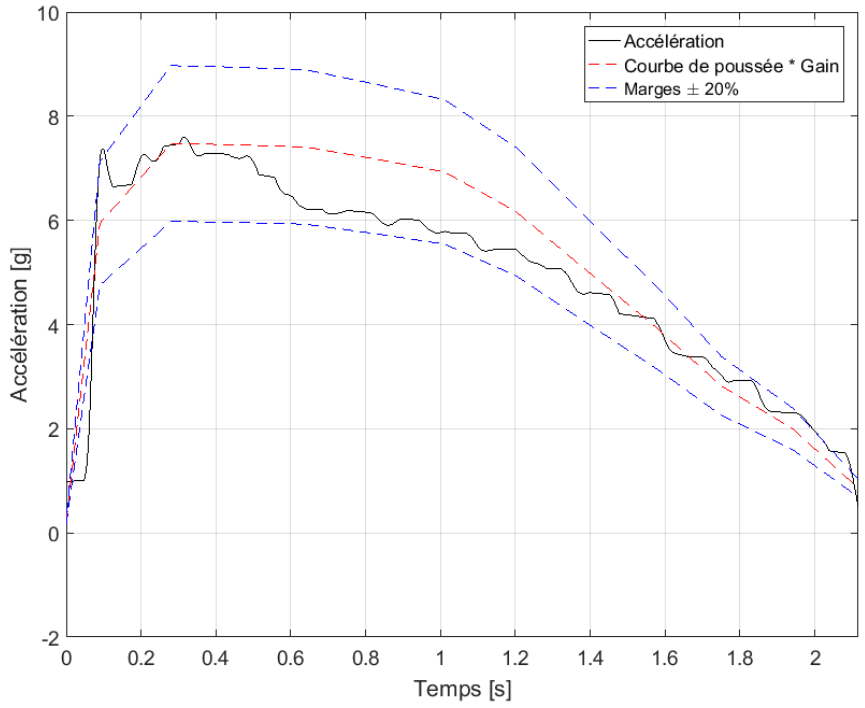


FIGURE 3.10 – Accélération et courbe de poussée.

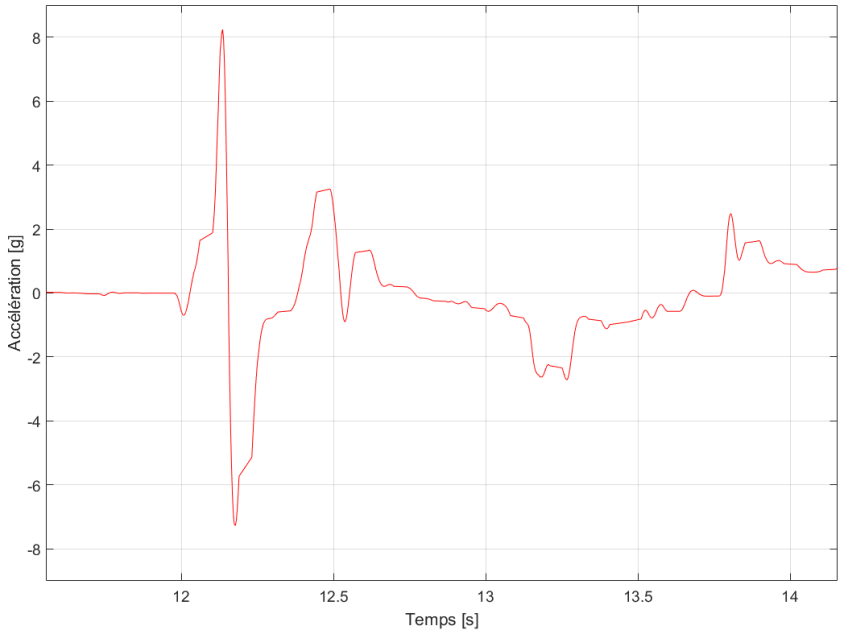


FIGURE 3.11 – Accélération radiale à l'ouverture du parachute

3.3 Conclusion module de mesure

Dans l'ensemble, le module de mesure à rempli sa tâche. Il nous a fourni bon nombre de données. Ces dernières sont loin d'être toutes analysées. Seul quelques

échantillons ont été présentés dans ce rapport. On retient qu'il est important de vérifier ses outils par l'expérimentation pour s'assurer qu'on se repose sur des hypothèses valables. Aussitôt que les mesures sont fiables, celles-ci renseignent sur le comportement de la fusée en vol, ce qui est très précieux car le vol représente le seul moment où l'humain ne peut pas observer véritablement la fusée.

Les tests statiques peuvent être réalisés au sol. En revanche, il est plus difficile de tester la dynamique de vol. C'est pourquoi il est important de développer un bon système embarqué. Cette partie démontre que des résultats exploitables peuvent être obtenus sans dépenser une somme conséquente.

Cinquième partie

Divers et conclusion

Chapitre 1

Activités annexes

1.1 Journées de lancement

Après un début très théorique dans ce projet, surtout de la lecture, de la recherche et autres simulations, l'équipe motivée avait envie d'un peu de mise en pratique. D'abord pour vérifier certains résultats théoriques de simulation mais aussi pour donner la chance à tous les membres de faire un peu de travaux manuels au labo. La façon la plus simple de pratiquer de la fusée dans le monde réel s'est rapidement avérée être les clubs de fusées amateurs : il existe en Suisse une association (ARGOS-http ://www.argoshpr.ch) qui organise des journées de lancement dans le respect des règles établies par une des associations faîtières de la fusée amateur (TRA - Tripoli Rocketry association - http ://www.tripoli.org/).

Les personnes qui désirent lancer des fusées dans le cadre de ces journées doivent impérativement être certifiées ou en train d'obtenir leur certification. La certification est en 3 phases et va pas à pas vers de plus en plus grosses fusées. Les 5 membres de l'équipe ont donc décidé de préparer leur 1er niveau de certification. Une fois la partie administrative réglée (inscriptions et paiements des cotisations et assurance au près des différent organismes impliqués), la construction a pu commencer. Quelques photos de construction sont en annexe G.

Une fois la partie découpage et collage terminée il a fallu préparer la journée de lancement, organisation des transports, négociation d'un budget, s'assurer de l'approvisionnement en moteurs par le responsable des certifications en Suisse, etc.

Une première journée à eu lieu à Cernier (NE) le 1er avril 2017.
Une deuxième journée à eu lieu à Kaltbrunn (ZH) le 20 mai 2017.

Lors de ces deux journées, tous les membres ont pu lancer une ou plusieurs fois leur fusée, et ainsi acquérir de l'expérience. Les vols avec électroniques embarquée ont permis de récolter de précieuses données (traitées dans la partie résultat). Un lancement lors de la journée du 20 mai a permis une certification de niveau 2, quelques règles et idée pour la préparation est la réalisation de ce vol sont en annexe H.

1.2 Meeting Ruag

Faire partie d'un projet de fusée ouvre certaines portes, dans le cadre de ce projet nous avons eu la possibilité de non seulement fréquenter les membres de l'EPFL rocket team participant au concours international IREC en été 2017 mais aussi d'être invités à leurs PDRs (preliminary design reviews) organisée par leur sponsor principal, RUAG AG, à Zurich. Ainsi nous avons pu rencontrer mais surtout écouter des professionnels du spatial décortiquer le design de nos confrères. Nous en avons retiré pas mal d'enseignements ainsi que des conseils, des mises en garde, des contacts... Deux membres de l'équipe ont pu participer à la première de ces journées de revue alors que quatre membres ont pu participer à la deuxième organisée dans les locaux de l'ETH de Zurich. Quelques points importants et quelques pense-bêtes issus de ces journée sont consignés en annexe E.

1.3 Discussion aéro-freins

Durant ce semestre, 4 autres collègues ont réalisé un autre travail de bachelor sur les fusées. Leur travail, plus axé sur le contrôle les a amenés à contacter notre équipe pour partager des connaissances et des conseils. Ainsi durant le semestre, une étroite collaboration s'est établie entre les deux équipes. Après discussions, le contrôle d'altitude est devenu le but à atteindre pour leur groupe. Intéressés de pouvoir réaliser un test réel de leur système, le groupe de contrôle a dans un premier temps demander à ce que l'on valide une stratégie de freinage de la fusée. Puis demander de l'aide et notre aval pour leur design mécanique contrôlé par leur avionique. Le but étant que leur système vole avec une de nos fusées. Finalement nous avons tous pris part à la dernière journée de vol organisée le 20 mai ou le système d'aérofreins a pu être testé avec succès. Une partie de ces discussions inter-équipes sont résumées en annexe F.

Chapitre 2

Conclusion

La simulation que nous avons écrite correspond aux objectifs que nous nous sommes fixés et nous permet d'évaluer les différents paramètres auxquels nous sommes intéressés pour décrire le vol d'une fusée. Les résultats que nous obtenons ont pu être comparés et validés avec les données expérimentales recueillies lors des vols.

L'analyse structurelle nous a permis de déterminer les cas de charges dominants lors d'une mission complète. Nous avons ainsi remarqué qu'il est primordial de dimensionner les différentes liaisons en fonction de la force générée par l'ouverture du parachute, et qu'il serait par conséquent intéressant de diminuer la vitesse de chute à cet instant. Nous avons de plus vérifié que notre design de coupleur était fonctionnel et sûr.

Pour la partie expérimentale nous avons tout d'abord construit chacun notre propre fusée ce qui nous a grandement appris d'un point de vue pratique. Puis toutes les mesures désirées ont pu être effectuées à bord des différents prototypes. L'expérience se conclut avec des résultats qui correspondent pour la plupart aux valeurs attendues.

Sixième partie

Annexes

Annexe A

Formules aéro

A.1 Introduction

This document is a synthetic summary of the master thesis written by Sampo Niskanen [8], presenting the most important formula used to describe the aerodynamic behaviour of a rocket during the ascending part of it's flight. It only covers the subsonic flow behaviour, as it was the part that we were interested in for the simulation.

The forces are normalised by the current dynamic pressure

$$q = \frac{1}{2}\rho v_0^2$$

and a characteristic area A_{ref} to get non-dimensional coefficients. For moments we add a characteristic length d.

The following results follow Barrowman's method :
-Split the rocket into separate components
-Calculate CP and CN_α separately for each component
-Sum all the contribution to get final values for the whole rocket

A.2 Variables and abreviations

Éléments	Abréviation	Formules
A_{fin}	Fin area	–
A_{plan}	Reference area	–
$Æ$	Aspect ratio of a fin	$\frac{2s^2}{A_{fin}}$
c	Speed of sound	–
$\bar{c}$	Mean aerodynamic chord length fin	–
$c(y)$	Chord length of a fin at spanwise position y	–
C_a	Axial drag coefficient	–
C_D	Drag force coefficient	$\frac{D}{\frac{1}{2}\rho v_0^2 A_{ref}}$
C_f	Skin friction drag coefficient	–
C_{ld}	Roll moment coefficient	$\frac{l}{\frac{1}{2}\rho v_0^2 A_{ref}}$

C_{lf}	Roll damping moment coefficient	–
C_m	Pitch moment coefficient	$\frac{m}{\frac{1}{2}\rho v_0^2 A_{ref} d}$
C_{m_α}	Pitch moment coefficient derivative	$\frac{\partial C_m}{\partial \alpha} \approx \alpha C_m$
C_N	Normal force coefficient	$\frac{N}{\frac{1}{2}\rho v_0^2 A_{ref}}$
C_{N_α}	Normal force coefficient derivative	$\frac{\partial C_N}{\partial \alpha} \approx \alpha C_N$
d	Reference length, rocket diameter, caliber	–
D	Drag force	–
f_b	Rocket fineness ratio	$\frac{L}{d}$
L	Rocket length	–
m	Pitch moment around the tip of the nose cone	–
M	mach number	–
N	normal force / number of fins	–
p	Air pressure	–
$r(x)$	Body radius at position x	–
R	Reynolds number	$\frac{v_0 L}{\nu}$
s	Spanwise length of a fin	–
T	Air temperature	–
V	Volume	–
v_0	Free-stream velocity	–
X	Position along the rocket centerline	–
y	Spanwise position	–
α	Angle of attack	–
β	-	$\sqrt{\lvert M^2 - 1 \rvert}$
γ	Specific heat ratio, for air $\gamma = 1,4$	–
Γ_c	Fin midchord sweep angle	–
δ	Fin cant angle	–
η	Airflow inclination angle over a fin	–
θ	Roll angle	–
Λ	Dihedral angle between a fin and the direction of airflow	–
ν	Kinematic viscosity of air	–
ξ	Distance from rotation axis	–
ρ	Density of air	–
ω	Angular velocity	–
CG	Center of gravity	–
CP	Center of pressure	–
LE	leading edge	–
MAC	Mean aerodynamic chor	–
RK4	Runge-Kutta 4 iteration method	–

A.3 Subsonic flow, basic revolution solids

A.3.1 C_{N_α} of body components

The normal force for an axially symmetric body at position x in a subsonic flow is given in the Barrowman's report by

$$N(x) = \rho v_0 \frac{\partial}{\partial x}\big[A(x)w(x)\big] \tag{A.1}$$

where $w(x) = v_0 sin(\alpha)$.

Assuming the derivative $\frac{dA(x)}{dx}$ is well definite and using the definition of the normal force coefficient C_N we can integrate over the component length

$$C_N(x) = \frac{2sin(\alpha)}{A_{ref}} \int_0^l \frac{dA(x)}{dx} dx = \frac{2sin(\alpha)}{A_{ref}}\big[A(l) - A(0)\big] \tag{A.2}$$

We then have

$$C_{N_\alpha}(x) = \frac{2}{A_{ref}}\big[A(l) - A(0)\big]\frac{sin(\alpha)}{\alpha} \tag{A.3}$$

Thus we see that as long as the component shape changes smoothly, the normal force coefficient derivative does not depends on the component shape. Due to the potential flow theory, a cylindrical body tube has no effect on the normal force coefficient or CP location.

A.3.2 C_{m_α} of body components

A normal force $N(x)$ at position x produces a pitching moment $m(x) = xN(x)$ at the nose cone tip. Therefore using the pitching moment definition and by integration over the body length we get

$$\begin{aligned} C_m &= \frac{2sin(\alpha)}{A_{ref}d} \int_0^l x\left(\frac{dA(x)}{dx}\right)dx \frac{2sin(\alpha)}{A_{ref}d}\left[lA(l) - \int_0^l A(x)dx\right] = \\ &= \frac{2sin(\alpha)}{A_{ref}d}\big[lA(l) - V\big] \end{aligned} \tag{A.4}$$

and

$$C_{m_\alpha} = \frac{2}{A_{ref}d}\big[lA(l) - V\big]\frac{sin(\alpha)}{\alpha} \tag{A.5}$$

A.3.3 Effect of body lift

Thus far we neglected the effect of body lift, due to our description of the flow around the rocket, and taking in account the d'Alembert paradox. Robert Galejs suggest however to add a correction term to the body component C_{m_α} to account for body lift. Thus the normal force exerted on a cylindrical body at an angle of attack α is

$$C_N = K\frac{A_{plan}}{A_{ref}} sin^2(\alpha) \tag{A.6}$$

where $A_{plan} = d \cdot l$ is the planform area of the cylinder and K is a constant $K \approx 1.1$. The CP location is assumed to be the center of the planform area

$$X_{lift} = \frac{\int_0^l x2r(x)dx}{A_{plan}} \tag{A.7}$$

A.3.4 Center of pressure of body components

The CP location of the body components can be calculated

$$X_B = \frac{(C_{m_\alpha})_B}{(C_{N_\alpha})_B} d = \frac{lA(l) - V}{A(l) - A(0)} \tag{A.8}$$

The total CP location can be calculated from the separate CP location X_i and normal force coefficient derivative $(C_{N_\alpha})_i$ by the moment sum

$$X = \frac{\sum_{i=1}^n X_i((C_{N_\alpha})_i)}{\sum_{i=1}^n (C_{N_\alpha})_i} \tag{A.9}$$

A.3.5 Planar fins

Barrowman argued that the CP of a regular trapezoidal fin is located at the intersection of its mean aerodynamic chord (MAC) and its quarter chord for low subsonic speed. Extension to freeform fins

Length of th MAC : $\bar{c} = \frac{1}{A_{fin}} \int_0^s c(y)^2 dy$
Spanwise location of MAC : $y_{MAC} = \frac{1}{A_{fin}} \int_0^s y \cdot c(y) dy$
Effective leading edge location $x_{MAC,LE} = \frac{1}{A_{fin}} \int_0^s x_{LE}(y) \cdot c(y) dy$
cf doc "Fins geometry pages A27-A30".

When theses equations are applied to trapezoidal fins and the lengthwise position of the CP is selected at the quarter chord, $X_f = x_{MAC,LE} + 0.25\bar{c}$ and we recovers Barrowman's results. Cf. Openrocket documentation pages 26-27.

Barrowman's method assumes that the lengthwise position of the CP stays at a constant 25% of the MAC at subsonic speed, however the position starts moving rearward above approximately Mach 0.5. For $M > 2$ the relative lengthwise position of the CP is given by an empirical formula.

$$\frac{X_f}{\bar{c}} = \frac{Æ\beta - 0.67}{2Æ\beta - 1} \tag{A.10}$$

Between Mach 0.5 and Mach 2 the lengthwise position of the CP is interpolated with a fifth order polynomial $p(M)$ with the constraints :
$p(0.5) = 0.25$, $p'(0.5) = 0$
$p(2) = f(2)$, $p'(2) = f'(2)$, $p''(2) = 0$, $p'''(2) = 0$

Single fin C_{N_α} at subsonic speed

Barrowman's derivation of Diederich semi-empirical method.

$$(C_{N_\alpha})_1 = \frac{C_{N_\alpha 0} F_D \left(\frac{A_{fin}}{A_{ref}}\right) cos(\Gamma_c)}{2 + F_D \sqrt{1 + \frac{4}{F_D^2}}} \tag{A.11}$$

Where :
The normal force coefficient derivative for a 2D aifoil in theory of potential flow corrected for compressible airflow and $M < 1$

$$C_{N_\alpha 0} = \frac{2\pi}{\beta} \tag{A.12}$$

Diedrich's planform correlation parameter

$$F_D = \frac{Æ}{\frac{1}{2\pi} C_{N_\alpha 0} cos(\Gamma_c)} \tag{A.13}$$

By substitution of $Æ$ we then get

$$(C_{N_\alpha})_1 = \frac{2\pi \frac{s^2}{A_{ref}}}{1 + \sqrt{1 + \left(\frac{\beta s^2}{A_{fin} cos(\Gamma_c)}\right)^2}} \tag{A.14}$$

which is the normal force coefficient derivative for one fin, where the angle of attack is between the airflow and fin surface.

Multiple fin C_{N_α}

The lift force of a fin is perpendicular to the fin and originate from its CP. Therefore a single fin may cause a rolling and yawing moment in addition to a pitching moment. In this case all of the forces and moments must be computed from the geometry. If there are two ore more fins placed symmetrically around the body then the yawing moments cancel and if there is no fin cant then the total rolling moment is also zero, and thus doesn't need to be computed. The dihedral angle between each of the fins and the airflow direction is denoted Λ_i. The fin i encounters a local angle of attack $\alpha_i = \alpha sin(\Lambda_i)$ for which the normal force component (parallel to the lateral airflow) is $(C_{N_\alpha})_{\Lambda_i} = (C_{N_\alpha})_1 sin^2(\Lambda_i)$. The sum of the coefficients then yields

$$\sum_{k=1}^{N} (C_{N_\alpha})_{\Lambda_i} = (C_{N_\alpha})_1 \sum_{k=1}^{N} sin^2(\Lambda_i) \tag{A.15}$$

Fin-body interference

The normal force coefficient must still be corrected for fin-body interference which increase the overall produced normal force. The effect of the extra fin lift is taken into account using a correction term :

$$(C_{N_\alpha})_{T(B)} = K_{T(B)} (C_{N_\alpha})_N \tag{A.16}$$

Where $(C_{N_\alpha})_{T(B)}$ is the normal force coefficient derivative of the tail in the presence of the body. The term $K_{T(B)}$ can be approximated by

$$K_T(B) = 1 + \frac{r_t}{s + r_t} \tag{A.17}$$

Where s is the fin span from root to tip and r_t is the body radius at the fin position. The value $(C_{N_\alpha})_{T(B)}$ is then used as the final force coefficient derivative for the fins.

A.3.6 Roll dynamics

When the fins of the rocket are canted at some angle $\delta > 0$, the fins induce a rolling moment on the rocket. ON the other hand when a rocket has a specific roll velocity the furthest part of the fins encounter a notable tangential velocity witch oppose the roll, thus resulting in a steady state roll velocity depending on the current velocity of the rocket.
The effect of roll on a fin can be examined by dividing the fin into narrow streamwise strips and integrating over the strips. A strip i at distance ξ_i from the rocket centerline encounter a radial velocity $u_i = \omega\xi_i$ where ω is the angular roll velocity. The radial velocity induces an angle of attack

$$\eta_i = tan^{-1}\left(\frac{u_i}{v_0}\right) = tan^{-1}\left(\frac{\omega\xi_i}{v_0}\right) \approx \frac{\omega\xi_i}{v_0} \tag{A.18}$$

When a fin is canted at an angle δ, the total inclination of the strip relative to the airflow is $\alpha_i = \delta - \eta_i$. Assuming that the forces produced by a strip is directly proportional to the local angle of attack, the force on strip i is for some k_i

$$F_i = k_i\alpha_i = k_i(\delta - \eta_i) \tag{A.19}$$

The total moment produced by th fin is then

$$l = \sum_i \xi_i F_i = \sum_i \xi_i k_i(\delta - \eta_i) = \sum_i \xi_i k_i(\delta) - \sum_i \xi_i k_i(\eta_i) \tag{A.20}$$

The effect of roll can be split into two components : the first term is the roll moment induced by a canted fin, while the second term is the opposing moment generated by uncanted fin when flying at a roll rate $\omega \neq 0$

Roll forcing coefficient

When flying at zero roll rate ($\omega = 0$) the angle δ acts simply as an angle of attack for each fins. The lift force originate from the mean aerodynamic chord and thus the roll forcing coefficient for N fins is

$$C_{lf} = \frac{N(y_{MAX} + r_t)(C_{N_\alpha})_1\delta}{d} \tag{A.21}$$

where r_t is the radius of the body tube at the fin position.

Roll damping coefficient

At subsonic speed the force generated by strip i is equal to

$$F_i = C_{N_{\alpha 0}} \frac{1}{2} \rho v_0^2 c_i \Delta\xi_i \eta_i \tag{A.22}$$

where $c_i \Delta\xi_i$ is the area of the strip. The roll damping moment generated by the strip is then

$$(C_{ld})_i = \frac{F_i \xi_i}{\frac{1}{2}\rho v_0^2 A_{ref} d} = \frac{C_{N_{\alpha 0}}}{A_{ref} d} \xi_i c_i \Delta\xi_i \eta_i \tag{A.23}$$

Replacing η_i and summing the airfoil strip, we get the total roll damping for N fins

$$C_{ld} = N \sum_i (C_{ld})_i = \frac{N C_{N_{\alpha 0}} \omega}{A_{ref} d v_0} \sum_i \xi_i^2 c_i \Delta\xi_i \tag{A.24}$$

Equilibrium roll frequency

Steady state roll frequency that canted fins induce on a rocket flying at a specific velocity. This is obtained by equating the roll forcing moment and roll damping moment and solving for the roll rate ω. For subsonic speed

$$f_{eq} = \frac{\omega_{eq}}{2\pi} = \frac{A_{ref} \beta v_0 y_{MAC} (C_{N_\alpha})_1 \delta}{4\pi^2 \sum_i \xi_i^2 c_i \Delta\xi_i} \tag{A.25}$$

A.3.7 Drag forces

Drag forces arise from two basic mechanisms, the air pressure distribution around the rocket and skin friction. Two different drag coefficient can be defined : C_D the total drag coefficient and C_A the axial drag coefficient. At zero angle of attack those two coincide, $C_{D_0} = C_{A_0}$

For subsonic flight the main source of drag is due to skin friction drag. As the purpose of this project is to remain in the subsonic region for simulation, the formula presented here will only cover this case and further information on body pressure drag can be found in the original document [8]

Boundary layer

At the front of a streamlined body the air flows smoothly around the body in layers, each having a different velocity. This stack of layers is called the boundary layer and inside it, the velocity is zero at the surface of the object and increase gradually : the further it is from the object the closest to the free stream velocity. If the flow is smooth enough it is said to be a laminar boundary layer which height increase with the distance from the leading edge. At some point the flow becomes turbulent in the boundary layer which height increase rapidly while the skin friction become notably larger. The point at which the flow becomes turbulent is the point that has a local critical Reynolds number :

$$R_{crit} = \frac{v_0 x}{\nu} \tag{A.26}$$

where v_0 is the free-stream air velocity, x is the distance along the body from the nose cone tip and $\nu = 1.5 \cdot 10^{-5}\ m^2/s$ is the kinematic viscosity of air. The critical Reynolds number is approximately $R_{crit} = 5 \cdot 10^5$.

Due to the relatively high speed of flying rocket, the critical point is near to the tip of the nose cone and thus the laminar part of the boundary layer can often be neglected.

Skin friction drag

The skin friction coefficient C_f is defined as the drag coefficient due to friction with the reference area being the total wetted area of the rocket, that is, the body and fins in contact with the airflow :

$$C_f = \frac{D_{friction}}{\frac{1}{2}\rho v_0^2 A_{wet}} \tag{A.27}$$

The values for C_f are given by different formulae depending on the Reynolds number. For fully turbulent flow and assuming that the surface is smooth the coefficient is given by :

$$C_f = \frac{1}{(1.50 \cdot ln(R) - 5.6)^2} \tag{A.28}$$

But at sufficient speed even slight roughness may have an effect on the skin friction. The critical Reynolds number corresponding to an approximate roughness height of the surface is given by :

$$R_{crit} = 51 \left(\frac{R_s}{L} \right)^{-1.039} \tag{A.29}$$

For Reynolds number above the critical value, the skin friction coefficient can be considered independent of Reynolds number thus we get :

$$C_f = 0.032 \left(\frac{R_s}{L} \right)^{0.2} \tag{A.30}$$

For very low Reynolds number, lower than $\approx 10^4$ we might consider a skin friction coefficient constant equal as for $R = 10^4$. This correspond typically to velocities below 1m/s and thus has negligible effect on calculations. Thus we have a complete definition of the coefficient for all Reynolds number :

$$C_f = \begin{cases} 1.48 \cdot 10^{-2} & \text{for } R < 10^4 \\ \text{Eq. } A.28 & \text{for } 10^4 \leqslant R \leqslant R_{crit} \\ \text{Eq. } A.30 & \text{for } R > R_{crit} \end{cases} \tag{A.31}$$

Even if we stay in subsonic speed during flight, some effect happens due to compressibility at Mach numbers close to 1. Thus we introduce a correction factor which decreases a little bit the friction skin coefficient :

$$C_{f_c} = C_f(1 - 0.1M^2) \tag{A.32}$$

Finally after correcting C_f we can convert it to the actual drag coefficient by scaling it to the correct reference area. The body wetter area is corrected for its cylindrical geometry and the fins for their finite thickness.

$$(C_D)_{friction} = C_{f_c} \frac{\left(1 + \frac{1}{2f_b}\right) \cdot A_{wet,body} + \left(1 + \frac{2t}{\bar{c}}\right) \cdot A_{wet,fins}}{A_r ef} \tag{A.33}$$

where the wetted area of the fins $A_{wet,fins}$ includes both sides of the fins, f_b is the fineness ration of the rocket, t the thickness of the fins and $\bar{c}$ mean aerodynamic chord length of the fins.

For further precision on body pressure drag which we may use to enhance precision of our model as well as precise fin drag which depends on the shape, we refer to the master Thesis of Sampo Niskanen [8].

Base drag

Base drag is caused by a low-pressure area created at the base of the rocket on in any place where the body radius diminishes rapidly enough. The magnitude of the base drag can be estimated using empirical formula :

$$(C_D)_{base} = \frac{A_{component}}{A_{ref}} \cdot \begin{cases} 0.12 + 0,13M^2 & \text{if } M < 1 \\ 0.25/M & \text{if } M > 1 \end{cases} \tag{A.34}$$

The base drag is disrupted when a motor exhaust into the area. Thus a good approximation is to subtract the area of the thrusting motors from the base reference area, reducing then the base drag during the first part of the flight.

Annexe B

MATLAB (classes et méthodes principales)

Les classes présentés sont qu'une partie de l'ensemble des fichiers.

B.1 MainTest.m

```
    % Main test for simulation of any rocket and mission

close all;
clear all;
clc;

% Creation des deux objets de la simulation
R = RocketIREC();
S = SimulationIREC();

% Simulate part
results = Simulate( R, S);

% DrawGraphs
DrawGraph(R, S, results);

% Abaqus_Analysis(20,10,0.2,max(abs(results.accel.Vz_dot)),R);
```

B.2 Rocket.m

```
classdef Rocket < handle

    properties (SetAccess = private, GetAccess = public)
        Nose
        Stage
        Tail
        Fins
        Motor
        Cylinder
        Coupler
        Parachute
```

```
        Point

        d = 0; % dimetre de reference a la base du cone
    end

    properties (SetAccess = public, GetAccess = public)
        k = 1.1; % coefficient de correction des corps de r?volution portants
        CD = 0.8; % coefficient de frottement aerodynamique
    end

    methods
        function obj = Rocket()
            % Rocket
            % Constructor of Rocket, empty for now
        end
    end

    methods
        function nose(obj, L, D, e, rho)
            % nose
            % Calcule la masse, le centre de masse, le centre de pression,
            % le coefficient aerodynamique, les moments d'inertie
            % INPUTS
            %   - L     :   longueur du cone
            %   - D     :   Diam?tre de la base du cone
            %   - e     :   epaisseur du cone
            %   - rho   :   densite
            %   - type  :   cone ou ogive

            % Assignation des proprietes
            obj.Nose.L = L;
            obj.Nose.D = D;
            obj.d = D;
            obj.Nose.e = e;
            obj.Nose.rho = rho;

            % Calcule intermediaire
            V = e*pi*sqrt(1+D^2/(4*L^2))*L*D/2;%volume

            % Calcule des proprietes de masse
            obj.Nose.m = rho*V;
            obj.Nose.cm = L*2/3;
            obj.Nose.Iz = rho/8*V*D^2;
            obj.Nose.Ir = rho*V/4*(D^2/4+2*L^2);

            % Calcule des proprietes aerodynamiques
            obj.Nose.CNa = @(alpha) 2*(sin(alpha)/(alpha*(alpha~=0)+(alpha==0))*(alpha~=0)+(alpha==0)); % d?riv?e du coefficient aerodynamique normal
            obj.Nose.zCP = 2/3*L; % position du centre de pression relatif au haut du cone
            Aplan = L*D/2;
            Aref = pi*D^2/4;
            obj.Nose.CNaBL = @(alpha, K) K*Aplan/Aref*(abs(sin(alpha)^2/(alpha*(alpha~=0)+(alpha==0)))*(alpha~=0));
        end

        function tail(obj, D1, D2, L, e, z, rho)
```

```
        % INPUTS
        % D1    :   diametre du haut
        % D2    :   diametre du bas
        % L     :   longueur du tail
        % e     :   epaisseur de paroie
        % z     :   postion par rapport au haut de la fusee du haut du tail
        % rho   :   densite

        % Assignation des proprietes
        obj.Tail.D1 = D1;
        obj.Tail.D2 = D2;
        obj.Tail.L = L;
        obj.Tail.e = e;
        obj.Tail.z = z;
        obj.Tail.rho = rho;

        % Calcule des proprietes de masse
        %on utilise un cone - un plus petit cone
        V = e*pi*sqrt(1+(D2-D1)^2/4/L^2)*L*(D1+D2)/2;
        obj.Tail.m = rho*V;
        obj.Tail.cm = L*(2*D2+D1)/(3*(D2+D1));

        R1 = D1/2;
        R2 = D2/2;
        m = (R2-R1)/L;
        obj.Tail.Iz = pi*rho/(10*m)*((R2^5-R1^5)-((R2-e)^5-(R1-e)^5));
        %obj.Tail.Ir = obj.Tail.Iz/2 + pi*rho*((m^2*L^5/5+1/2*m*L^4*R1+R1^2*L^3/3)-(m^2*L^5/5+1/2*m*L^4*(R1-e)+(R1-e)^2*L^3/3));
        obj.Tail.Ir = obj.Tail.Iz/2 + pi*rho*((1/2*m*L^4*e+(R1^2-(R1-e)^2)*L^3/3));


        % Calcule des proprietes aerodynamiques
        obj.Tail.CNa = @(alpha) 2/obj.d^2*(D2^2-D1^2)*(sin(alpha)/(alpha*(alpha~=0)+(alpha==0))*(alpha~=0)+(alpha==0)); % d?riv?e du coefficient aerodynamique normal
        obj.Tail.zCP = L/3*(1+(1-D1/D2)/(1-(D1/D2)^2)); % position du centre de pression relatif
        Aplan = (D1+D2)/2*L;
        Aref = pi*obj.d^2/4;
        obj.Tail.CNaBL = @(alpha, K) K*Aplan/Aref*(abs(sin(alpha)^2/(alpha*(alpha~=0)+(alpha==0)))*(alpha~=0));
    end

    function stage(obj, id, z, L, Dout, e, rho)
        % stage
        % Calcule la masse, le centre de masse, le centre de pression,
        % le coefficient aerodynamique, les moments d'inertie
        % INPUTS
        %   - id    :   id du stage (string)
        %   - z     :   position du haut de la payload
        %   - m     :   masse
        %   - L     :   longueur de l'etage
        %   - Dout  :   diametre externe
        %   - e     :   epaisseur de la coque
        %   - rho   :   densite

        % Assignation des proprietes
```

```
            stage.id = id; %
            stage.z = z;
            stage.L = L;
            stage.Dout = Dout;
            stage.e = e;
            stage.rho = rho;

            %Calcul intermediaire
            Din = Dout - 2*e;%Si besoin

            % Calcule des proprietes de masse
            %probleme au niveau de l atribution aux differents etages
            stage.m = rho*pi*L*((Dout/2)^2-(Din/2)^2);
            stage.cm = L/2;
            stage.Iz = pi*rho*L*1/2*((Dout/2)^4-(Din/2)^4);
            stage.Ir = pi*rho*L*1/12*(3*((Dout/2)^4-(Din/2)^4)+L^2*((Dout/2)^2-(Din/2)
    ^2));

            % Calcule des proprietes aerodynamiques
            Aplan = Dout*L;
            Aref = obj.d^2*pi/4;
            stage.CNaBL = @(alpha, K) K*Aplan/Aref*(abs(sin(alpha)^2/(alpha*(alpha~=0)
    +(alpha==0)))*(alpha~=0)); % coefficient aerodynamique normal
            stage.zCPBL = L/2;

            obj.Stage = [obj.Stage stage];
        end

        function coupler(obj, id, z, L, Dout, e, rho)
            % Coupleur (piece d'assemblage sans proprietes aerodynamiques
            % Calcule la masse, le centre de masse, le centre de pression,
            % les moments d'inertie
            % INPUTS
            %   - id    :   id du coupleur (string)
            %   - z     :   position du haut de la payload
            %   - m     :   masse
            %   - L     :   longueur de l'etage
            %   - Dout  :   diametre externe
            %   - e     :   epaisseur de la coque
            %   - rho   :   densite

            % Assignation des proprietes
            coupler.id = id; %
            coupler.z = z;
            coupler.L = L;
            coupler.Dout = Dout;
            coupler.e = e;
            coupler.rho = rho;

            %Calcul intermediaire
            Din = Dout - 2*e;

            % Calcule des proprietes de masse
            coupler.m = rho*pi*L*((Dout/2)^2-(Din/2)^2);
            coupler.cm = L/2;
            coupler.Iz = pi*rho*L*1/2*((Dout/2)^4-(Din/2)^4);
```

```
            coupler.Ir = pi*rho*L*1/12*(3*((Dout/2)^4-(Din/2)^4)+L^2*((Dout/2)^2-(Din/2)^2));

            obj.Coupler = [obj.Coupler coupler];
        end

        function motor(obj, id, z, D, L, e, m, mp, rho, thrustCurve, bt)
            % payload
            % Calcule la masse, le centre de masse, le centre de pression,
            % le coefficient aerodynamique, les moments d'inertie
            % INPUTS
            %   - id      :   id du moteur
            %   - z       :   position du haut du moteur
            %   - D       :   Diametre externe du moteur
            %   - L       :   longueur du moteur
            %   - e       :   epaisseur de la paroi
            %   - m       :   masse initiale
            %   - mp      :   masse du combustible
            %   - rho     :   densite du combustible
            %   - thrustCurve
            %             :   Courbe de poussee (N vs. s)
            %   - bt      :   temps de combustion (s)

            motor.id = id;
            motor.z = z;
            motor.D = D;
            motor.L = L;
            motor.e = e;
            motor.m = @(t)(m - mp/bt*t)*(t<=bt)+(m-mp)*(t>bt);
            motor.mp = mp;
            motor.ThrustCurve = thrustCurve;
            motor.bt = bt;

            % Calcule des proprietes de masse

            rho_shell = (m-mp)/(L*pi/4*(D^2-(D-2*e)^2));
            rho_prop = rho;

            if strcmp(id,'motor')
                % le moteur se consume radialement

                motor.cm = @(t) L/2;
                Din = @(t) sqrt((D-e)^2-4*motor.m(t)/L/rho/pi);
                % Check inner diameter
                if ~isreal(Din(0))
                    error(['Le volume du combustible depasse la place disponible. Solution: reduire la masse de combustible ou augmenter sa densite.',...
                            'Le Diametre interne depasse de ', num2str(imag(Din(0))) ' m']);
                end
                motor.Iz = @(t) pi*rho_shell*L/2*((D/2)^4-(D/2-e)^4)...
                                + pi*rho_prop*L/2*((D/2-e)^4-(Din(t))^4);
                motor.Ir = @(t) pi*rho_shell*L/12*(3*((D/2)^4-(D/2-e)^4)...
                                + L^2*((D/2)^2-(D/2-e)^2))...
                                + pi*rho_prop*L/12*(3*((D/2-e)^4-(Din(t)/2)^4)...
                                + L^2*((D/2-e)^2-(Din(t)/2)^2));

```

```
            elseif strcmp(id,'tank')
                % le reservoir se vide de haut en bas.

                h = @(t) 4*(motor.m(t)-m+mp)/rho/pi/(D-2*e)^2;
                cm_shell = L/2;
                cm_prop  = @(t) L - h(t)/2*(motor.m(t)-m+mp)/mp;
                motor.cm = @(t) (cm_shell*(m-mp) + cm_prop(t)*(motor.m(t)-m+mp))/motor.m(t);
                motor.Iz = @(t) pi*rho_shell*L/2*((D/2)^4-(D/2-e)^4)...
                                + pi*rho_prop*h(t)/2*((D/2-e)^4);
                Ir_shell  = pi*rho_shell*L/12*(3*((D/2)^4-(D/2-e)^4)...
                            + L^2*((D/2)^2-(D/2-e)^2));
                Ir_prop   = @(t) (motor.m(t)-m+mp)/12*(3*(D/2-e)^2+h(t)^2);
                motor.Ir = @(t) Ir_prop(t) + (motor.cm(t)-cm_prop(t))^2*(motor.m(t)-m+mp)...
                                + Ir_shell + (motor.cm(t)-cm_shell)^2*(m-mp);

            else

                error('Motor element must either be a motor or a tank.')

            end


            obj.Motor = [obj.Motor motor];

        end

        function cylinder(obj, id, z, m, L, D)
            % cylinder
            % Calcule la masse, le centre de masse, les moments d'inertie
            % INPUTS
            %   - id    :   id du payload (string)
            %   - z     :   position du haut de la payload
            %   - m     :   masse
            %   - L     :   longueur de la payload
            %   - D     :   diametre de la paylaod

            % Assignation des proprietes
            cylinder.id = id;
            cylinder.z = z;
            cylinder.m = m;
            cylinder.L = L;
            cylinder.D = D;

            % Calcule des proprietes de masse
            cylinder.cm = (1/2)*L;
            cylinder.Iz = m*D^2/8;
            cylinder.Ir = m/12*(3*(D/2)^2+L^2)+m*(L/2)^2;

            obj.Cylinder = [obj.Cylinder cylinder];
        end

        function parachute(obj, id, z, L, m, D, Cd)
            % parachutee
            % Calcule la masse, le centre de masse, le centre de pression,
```

```
            % le coefficient aerodynamique, les moments d'inertie
            % INPUTS
            %   - id    :   id du parachute
            %   - z     :   emplacement du parachute
            %   - L     :   longueure du parachute pli?
            %   - m     :   masse du parachute
            %   - D     :   diametre du parachute ouvert
            %   - Cd    :   Coefficient de trainee

            % Assignation des proprietes
            parachute.id = id;
            parachute.z = z;
            parachute.L = L;
            parachute.m = m;
            parachute.D = D;
            parachute.Cd = Cd;

            % Calcule des proprietes de masse
            parachute.cm = 0;%car masse ponctuelle

            obj.Parachute = [obj.Parachute parachute];

        end

        function fins(obj, z, N, Ct, Cr, xt, S, r, e, rho)
            % fins
            % Calcule la masse, le centre de masse, le centre de pression,
            % le coefficient aerodynamique, les moments d'inertie
            % INPUTS
            %   - z     :   position du haut de la fins
            %   - N     :   nombre de fins
            %   - Ct    :   longueur de la pointe de l'aileron
            %   - Cr    :   longueur de la base de l'aileron
            %   - xt    :   port?e axiale du bord d'attaque
            %   - S     :   envergure de l'aileron
            %   - r     :   distance de l'aileron par rapport ? l'axe
            %   vertical
            %   - e     :   epaisseur de la fins
            %   - rho   :   densite

            % Assignation des proprietes
            obj.Fins.z = z;
            obj.Fins.N = N;
            obj.Fins.Ct = Ct;
            obj.Fins.Cr = Cr;
            obj.Fins.xt = xt;
            obj.Fins.S = S;
            obj.Fins.r = r;
            obj.Fins.e = e;
            obj.Fins.rho = rho;

            %Calcules intermediaires
            % densit? surfacique
            lamb = rho * e;
            % Calcule du premier triangle (Surface 1)
            Iz1 = lamb*xt*S^3/36;
```

```
            Ir1  = lamb*xt^3*S/36;
            A1   = xt*S/2;
            cmz1 = 2*xt/3;
            cmr1 = S/3;
            % Calcule du rectangle (Surface 2)
            Iz2  = lamb*Ct*S^3/3;
            Ir2  = lamb*Ct^3*S/3;
            A2   = Ct*S;
            cmz2 = Ct/2;
            cmr2 = S/2;
            % Calcule du deuxi?me triangle (Surface 3)
            xe   = Cr-Ct-xt;
            Iz3  = lamb*xe*S^3/36;
            Ir3  = lamb*xe^3*S/36;
            A3   = xe*S/2;
            cmz3 = xe/3;
            cmr3 = S/3;

            % Calcule des proprietes de masse
            Atot     = (Cr + Ct)/2*S;
            obj.Fins.m        = lamb*Atot;
            obj.Fins.cmz      = (A1*cmz1+A2*(xt+cmz2)+A3*(xt+Ct+cmz3))/Atot;
            obj.Fins.cmr      = (A1*cmr1+A2*cmr2+A3*cmr3)/Atot;
            obj.Fins.Iz       = Iz1 + Iz2 + Iz3 + lamb*(A1*cmr1^2+A2*cmr2^2+A3*cmr3^2);
            obj.Fins.Ir       = Ir1 + Ir2 + Ir3 + lamb*(A1*cmz1^2+A2*(xt+cmz2)^2+A3*(xt
+Ct+cmz3)^2);
            obj.Fins.Itheta   = obj.Fins.Iz+obj.Fins.Ir;

            % Calcule des proprietes aerodynamiques
            obj.Fins.CNa = @(M, theta) cna_fins(N, r, S, obj.d, Cr, Ct, xt, M, theta); %
coefficient aerodynamique normal
            obj.Fins.zCP = xt/3*(Cr+2*Ct)/(Cr+Ct) + 1/6*((Cr+Ct)-(Cr*Ct)/(Cr+Ct)); %
position du centre de pression relatif SUR LE FINS
        end

        function point(obj, id, z, L, m)
            % point materiel
            % Calcule la masse, le centre de masse, les moments d'inertie
            % INPUTS
            %   - id     :   id de la masse (string)
            %   - z      :   position de la masse
            %   - L      :   longueure r?elle de l'?l?ment de masse
            %   - m      :   masse (i.e. equilibrage, d?placememtn centre de
            %   gravite)

            % Assignation des proprietes
            point.id = id;
            point.z = z;
            point.L = L;
            point.m = m;

            % Calcule des proprietes de masse
            point.cm =0;%en zero car point massique

            obj.Point = [obj.Point point];

```

```
        end

    end

    methods (Access = public)
        % rocket property methods

        function setCoeffAero(obj, k)
            % changer la valeur du coefficient aerodynamique
            obj.k = k;
        end

        function m = m(obj, t)
            % calcule de la masse totale

            % masse statique
            m_stat = obj.Nose.m;
            m_stat = m_stat + sum([obj.Stage.m]);
            m_stat = m_stat + obj.Tail.m;
            m_stat = m_stat + obj.Fins.N*obj.Fins.m;
            m_stat = m_stat + sum([obj.Cylinder.m]);
            m_stat = m_stat + sum([obj.Coupler.m]);
            m_stat = m_stat + sum([obj.Parachute.m]);
            m_stat = m_stat + sum([obj.Point.m]);

            % masse variable (masse du moteur)
            m = (m_stat + sum(cellfun(@(c) c(t), {obj.Motor.m})));

        end

        function cm = cm(obj, t)
            % calcule du centre de mass
            % INPUT:
            %   - t      : temps
            % OUTPUT:
            %   - cm     : position du centre de mass

            cm_stat = obj.Nose.m*obj.Nose.cm;
            cm_stat = cm_stat + sum([obj.Stage.m].*([obj.Stage.z] + [obj.Stage.cm]));
            cm_stat = cm_stat + obj.Tail.m*(obj.Tail.z+obj.Tail.cm);
            cm_stat = cm_stat + obj.Fins.m*obj.Fins.N*(obj.Fins.z+obj.Fins.cmz);
            cm_stat = cm_stat + sum([obj.Cylinder.m].*([obj.Cylinder.z] + [obj.Cylinder.cm]));
            cm_stat = cm_stat + sum([obj.Coupler.m].*([obj.Coupler.z] + [obj.Coupler.cm]));
            cm_stat = cm_stat + sum([obj.Parachute.m].*([obj.Parachute.z] + [obj.Parachute.
 cm]));
            cm_stat = cm_stat + sum([obj.Point.m].*([obj.Point.z] + [obj.Point.cm]));

            Motor_m = cellfun(@(c) c(t), {obj.Motor.m});
            Motor_cm = cellfun(@(c) c(t), {obj.Motor.cm});
            cm = (cm_stat + sum(Motor_m.*([obj.Motor.z]+Motor_cm)))/obj.m(t);
        end

        function Iz = Iz(obj, t)
            % calcule du moment d'inertie axial au centre de masse
            % INPUT:
            %   - t      : temps
```

```
        % OUTPUT:
        %   - Iz     : moment d'inertie axial au centre de masse

        Iz_stat = obj.Nose.Iz;
        Iz_stat = Iz_stat + sum([obj.Stage.Iz]);
        Iz_stat = Iz_stat + obj.Tail.Iz;

        Iz_stat = Iz_stat + obj.Fins.N*(obj.Fins.Iz +...
            obj.Fins.m*(2*obj.Fins.cmr*obj.Fins.r+ obj.Fins.r^2));
        % moment calculation
        Iz_stat = Iz_stat + sum([obj.Cylinder.Iz]);
        Iz_stat = Iz_stat + sum([obj.Coupler.Iz]);
        Iz = Iz_stat + sum(cellfun(@(c) c(t), {obj.Motor.Iz}));
    end

    function Ir = Ir(obj, t)
        % calcule du moment d'inertie perpendiculaire a l'axe de la
        % fusee au centre de masse
        % INPUT:
        %   - t      : temps
        % OUTPUT:
        %   - Ir     : moment d'inertie perpendiculaire a l'axe, au centre de masse

        CMt = obj.cm(t);

        Ir = obj.Nose.Ir + obj.Nose.m*(CMt.^2-CMt*obj.Nose.cm);
        Ir = Ir + sum([obj.Stage.Ir] + [obj.Stage.m].*...
                (CMt-[obj.Stage.z]-[obj.Stage.cm]).^2);
        Ir = Ir + obj.Tail.Ir + obj.Tail.m*(...
            (CMt-obj.Tail.z)^2-(CMt-obj.Tail.z)*obj.Tail.cm);

        % calcule du moment d'inertie pour les ailerons
        phi = 2*pi/obj.Fins.N; % angle entre les ailerons
        for i = 1:obj.Fins.N
            phi_i = phi*(i-1); % angle de rotation du sys de coordonnees
            Ir_fin = cos(phi_i)^2*obj.Fins.Itheta+sin(phi_i)^2*obj.Fins.Ir;
            Ir = Ir + Ir_fin + obj.Fins.m*(...
            (CMt-obj.Fins.z)^2-(CMt-obj.Fins.z)*obj.Fins.cmz);
        end

        Ir = Ir + sum([obj.Cylinder.Ir] + [obj.Cylinder.m].*(...
            (CMt-[obj.Cylinder.z]).^2-(CMt-[obj.Cylinder.z]).*...
            [obj.Cylinder.cm]));

        Ir = Ir + sum([obj.Coupler.Ir] + [obj.Coupler.m].*(...
            (CMt-[obj.Coupler.z]).^2-(CMt-[obj.Coupler.z]).*...
            [obj.Coupler.cm]));

        Ir = Ir + sum([obj.Parachute.m].*(CMt-[obj.Parachute.z]).^2);

        Ir = Ir + sum([obj.Point.m].*(CMt-[obj.Point.z]).^2);

        Motor_m = cellfun(@(c) c(t), {obj.Motor.m});
        Motor_Ir = cellfun(@(c) c(t), {obj.Motor.Ir});
        Motor_cm = cellfun(@(c) c(t), {obj.Motor.cm});

```

```
            Ir = Ir + sum(Motor_Ir + Motor_m.*...
                (CMt-[obj.Motor.z]-Motor_cm).^2);
        end

        function [CNa_tot, zCP] = aeroCoeff(obj, alpha, M, theta)
            % calcule des proprietes aerodynamiques de la fusee
            % INPUT:
            %   - alpha : angle d'attaque (rad)
            %   - M     : nombre de Mach
            %   - theta : angle de rotation des ailerons

            CNa_Nose = obj.Nose.CNa(alpha);
            zCP_Nose = obj.Nose.zCP;
            CNa_NoseBL = obj.Nose.CNaBL(alpha, obj.k);

            CNa_Tail = obj.Tail.CNa(alpha);
            zCP_Tail = obj.Tail.zCP+obj.Tail.z;
            CNa_TailBL = obj.Tail.CNaBL(alpha, obj.k);

            CNa_StageBL = cellfun(@(c) c(alpha, obj.k), {obj.Stage.CNaBL});
            zCP_StageBL = [obj.Stage.zCPBL] + [obj.Stage.z];

            CNa_Fins = obj.Fins.CNa(M, theta);
            zCP_Fins = obj.Fins.zCP+obj.Fins.z;

            CNa_tot =   (CNa_Nose + CNa_NoseBL + CNa_Tail + CNa_TailBL + ...
                         sum(CNa_StageBL) + CNa_Fins);

            zCP =   (zCP_Nose*(CNa_Nose+CNa_NoseBL)+zCP_Tail*(CNa_Tail+CNa_TailBL)
+...
                    sum(zCP_StageBL.*CNa_StageBL)+zCP_Fins*CNa_Fins)/CNa_tot;
        end

        function check(obj)
            % Controle de la definition de la fusee

            % check : toutes les parties sont definies ?

            if (isempty(obj.Nose))
                error('Must define a nose cone.');
            elseif (isempty(obj.Stage))
                warning('No stage defined.');
            elseif (isempty(obj.Tail))
                warning('No tail defined.');
            elseif (isempty(obj.Fins))
                warning('No fins defined.');
            elseif (isempty(obj.Motor))
                error('Must define a motor.');
            elseif (isempty(obj.Cylinder))
                warning('No Cylinder defined.');
            elseif (isempty(obj.Coupler))
                warning('No Coupler defined.');
            elseif (isempty(obj.Parachute))
                warning('No parachute defined.');
            elseif (isempty(obj.Point))
                warning('No point mass defined.');
```

```
            end

            % check : diametre de reference
            if isnan(obj.d) || obj.d <=0
                error(['Reference diameter d is not defined yet or has a',...
                        'value of 0. Define a nose cone with non-zero base',...
                        'diameter.']);
            end
        end

        function printSpecs(obj)
            % affiche les specs de la fusee

            display('********************');
            display('Rocket Specifications');
            display('********************');

            display(['* L     : ' num2str(obj.getLength()) ' [m]']);

            % reference diameter
            D = obj.Nose.D;
            display(['* Dref : ' num2str(D) ' [m]']);

            % liftoff weight
            m0 = obj.m(0);
            display(['* m0   : ' num2str(m0) ' [kg]']);

            % propellant weights
            display('* propellant masses :');
            for motor = obj.Motor
                display(['** ' motor.id ' : ' num2str(motor.mp) ' [kg]']);
            end
            display('* total propellant mass');
            display(['** mp   : ' num2str(sum([obj.Motor.mp])) ' [kg]']);

            % static margin
            display('* stabililty margin');
            [~, zcp] = obj.aeroCoeff(0, 0, 0);
            display(['* ' num2str((zcp-obj.cm(0))/obj.d) ' calibers'])
        end

        function m = getAll_m(obj, t)
            % retourne toutes les masses dans un ordre arbitraire, mais le
            % meme que getAll_z et getAll_l
            Motor_m = cellfun(@(c) c(t), {obj.Motor.m});
            m = [obj.Nose.m, [obj.Stage.m], obj.Tail.m, obj.Fins.m,...
                Motor_m, [obj.Cylinder.m], [obj.Coupler.m], [obj.Parachute.m], [obj.Point.m]];
        end

        function z = getAll_z(obj)
            % retourne toutes les positions dans un ordre arbitraire, mais le
            % meme que getAll_m et getAll_l
            z = [0, [obj.Stage.z], obj.Tail.z, obj.Fins.z,...
                [obj.Motor.z], [obj.Cylinder.z], [obj.Coupler.z], [obj.Parachute.z], [obj.
    Point.z]];
        end
```

```
        function l = getAll_l(obj)
            % retourne toutes les positions dans un ordre arbitraire, mais le
            % meme que getAll_m et getAll_z
            l = [obj.Nose.L, [obj.Stage.L], obj.Tail.L, obj.Fins.Cr,...
                [obj.Motor.L], [obj.Cylinder.L], [obj.Coupler.L], [obj.Parachute.L], [obj.Point.L]];
        end

        function L = getLength(obj)
            % retourne la longueure de la fusee
            L = max(obj.getAll_z()+obj.getAll_l());
        end

    end
end

function CNa = cna_fins(N, rt, S, d, Cr, Ct, xt, M, theta)
    % cna_fins
    % calculate NACD of N (3 or 4) trapezoidal fins.
    % INPUTS:
    %   - N     : number of fins
    %   - rt    : radius of body at fin attachement
    %   - S     : span length of fin
    %   - d     : reference diameter (diameter at base of cone)
    %   - Cr    : fin root chord
    %   - Ct    : fin tip chord
    %   - xt    : longitudinal distance between leading edge of
    %             root chord and leading edge of tip chord. [m]
    %   - M     : Mach number
    %   - theta : angle of attack of fin along r axis.
    % RETURN:
    %   - cna   : NACD [1/rad]

    % check d
    if isnan(d) || d <=0
        error(['Reference diameter d is not defined yet or has a',...
        'value of 0. Define a nose cone with non-zero base diameter.']);
    end

    % check for M<1
    if (M>1)
        warning('Mach number must be < 1. Code cannot guarantee validity for supersonic speeds.');
    end

    % check for 1<=N<=8
    if (N<1 || N>8)
        error('The number of fins must be between 1 and 8.');
    end
   Aref      = pi*d^2/4;
   beta      = sqrt(1-M^2);
   gammac    = atan((xt+Ct/2-Cr/2)/S); % midchord angle
   Afin      = (Cr+Ct)/2*S; % fin Area
   CNa1      = 2*pi*S^2/Aref/(1+sqrt(1+(beta*S^2/(Afin*cos(gammac)))^2)); % CNa for one fin
```

```
    CNa1     = (1+rt/(rt+S))*CNa1; %corrected CNa for fin-body interference

    % Multiple fins
    if (N == 1)
        CNa = CNa1*sin(theta)^2;
    elseif (N == 2)
        CNa = CNa1*(sin(theta)^2+sin(theta+pi)^2);
    elseif (N == 3)
        CNa = CNa1*1.5*(1-0.15*cos(3*theta/2)^2);
    elseif (N == 4)
        CNa = CNa1*2*(1-0.06*sin(2*theta)^2);
    elseif (N == 5)
        CNa = 2.37*CNa1;
    elseif (N == 6)
        CNa = 2.74*CNa1;
    elseif (N == 7)
        CNa = 2.99*CNa1;
    elseif (N == 8)
        CNa = 3.24*CNa1;
    end
end
```

B.3 Simulate.m

```
    function Results = Simulate(R, S)
%SIMULATE effectue la simulation de la fusee
%   INPUTS :
%       - R     : objet 'Rocket'
%       - S     : objet 'Simulation'
%   OUTPUTS :
%       - Results: structure de resultats contenant les instants de temps
%       discrets des int?grateurs (tsim), les valeurs des variables d'etat
%       aux instants de temps discrets (Xsim), l'angle d'attaque (alpha),
%       la marge de stabilite en calibres (calibre), la force de
%       cisaillement (T) et le moment de flexion (M).

    %%%%
```

```
    T = zeros(length(S.tquer), S.nquer);
    M = zeros(length(S.tquer), S.nquer);
    xquer = linspace(0, R.getLength(), S.nquer);

    %%%%%%%%%%%%%%%%%%%%%%%%
    % Find start time
    %%%%%%%%%%%%%%%%%%%%%%%%
    % when motor thrust compensates gravity
    FT_i = min(find(R.Motor.ThrustCurve(:,2)>cos(S.phi0)*R.m(0)*g));
    t_start = R.Motor.ThrustCurve(FT_i,1);


    %%%%%%%%%%%%%%%%%%%%%%%%
    % Integration options
    %%%%%%%%%%%%%%%%%%%%%%%%

    % integrator options launch
    options_launch = odeset('Events',@events_launch,'OutputFcn',@odeplot, 'OutputFcn', @output,...
                        'Refine', 1);
    tspan_launch = [t_start, 10];

    % integrator options flight
    options_flight = odeset('Events', @events_flight, 'OutputFcn', @output,...
                        'Refine', 1);

    %%%%%%%%%%%%%%%%%%%%%%%%
    % Integration
    %%%%%%%%%%%%%%%%%%%%%%%%

    display('Started Sim...')

    % launch
    x0_launch = [0, 0, 0, 0, S.phi0, 0];
    [tsim_launch, Xsim_launch] = ode45(@(t, x) stateEquation(t, x, R, S, 0), tspan_launch, x0_launch, options_launch);
    display('Cleared launch pad:');
    display(['t = ' num2str(tsim_launch(end)) ' sec']);
    display(['Z = ' num2str(Xsim_launch(end, 2)) ' m']);
    display(['V = ' num2str(Xsim_launch(end, 4)) ' m/s']);

    % propulsed flight
    tspan_flight_withFt = [tsim_launch(end), R.Motor.ThrustCurve(end,1)];
    [tsim_flight_withFt, Xsim_flight_withFt] = ode45(@(t, x) stateEquation(t, x, R, S, 1), tspan_flight_withFt, Xsim_launch(end, :), options_flight);
    display('Motor burnout:');
    display(['t = ' num2str(tsim_flight_withFt(end)) ' sec']);
    display(['Z = ' num2str(Xsim_flight_withFt(end, 2)) ' m']);
    display(['V = ' num2str(Xsim_flight_withFt(end, 4)) ' m/s']);

    % coasting
    tspan_flight_withOutFt = [R.Motor.ThrustCurve(end,1), S.tfin];
    [tsim_flight_withOutFt, Xsim_flight_withOutFt] = ode45(@(t, x) stateEquation(t, x, R, S, 2), tspan_flight_withOutFt, Xsim_flight_withFt(end, :), options_flight);
    display('Apogee:');
    display(['t = ' num2str(tsim_flight_withOutFt(end)) ' sec']);
```

```
    display([ 'Z = ' num2str(Xsim_flight_withOutFt(end, 2)) ' m']);
    display([ 'V = ' num2str(Xsim_flight_withOutFt(end, 4)) ' m/s']);


    %%%%%%%%%%%%%%
    % Output
    %%%%%%%%%%%%%%

    tsim = [tsim_launch; tsim_flight_withFt; tsim_flight_withOutFt];
    Xsim = [Xsim_launch; Xsim_flight_withFt; Xsim_flight_withOutFt];

    Results.tsim = tsim;
    Results.Xsim = Xsim;
    Results.alpha = alpha;
    Results.calibre = calibre;
    Results.flexion.T = T;
    Results.flexion.M = M;
    Results.xquer = xquer;
    Results.accel.Vx_dot = Vx_dot_out;
    Results.accel.Vz_dot = Vz_dot_out;

    %%%%%%%%%%%%%%%%%%
    % State Equation
    %%%%%%%%%%%%%%%%%%

    function deriv = stateEquation(t, x, R, S, phase)
    % deriv = stateEquation(t, x, R, S, phase) Equation d'etat pour
    % l'integration


        % definition des variables a integrer
        X = x(1);        % position horizontale dans le repere terrestre
        Z = x(2);        % position verticale dans le rep?re terrestre
        Vx = x(3);
        Vz = x(4);
        phi = x(5);      % angle de rotation par rapport a la verticale
        phi_dot = x(6); % derive de l'angle de rotation

        % definition des constantes

        % matrice de rotation: fusee -> terrestre
        Q = [cos(phi), sin(phi); -sin(phi), cos(phi)];

        % derivation discrete
        % etape de temps de derivation
        dt = 0.1;

        % environnement
        % etat de l'air
        %  - a     : vitesse du son
        %  - rho   : densit?
        [~, a, ~, rho] = stdAtmos(Z);
        % vent relatif (incident)
        Vi = [S.v_vent - Vx; -Vz];

        % Nombre de Mach
```

```
        Mach = sqrt(Vx^2+Vz^2)/a;

        % Proprietes geometriques
        % diametre de reference
        d = R.d;
        % Aire de reference
        Aref = pi*d^2/4;
        L = R.Tail.z+R.Tail.L;

        % Proprietes de masse
        bt = max([R.Motor.bt]);
        m = R.m(t);
        if (t==0)
            m_dot = (R.m(dt)-R.m(0))/dt;
        elseif (t<bt)
            m_dot = (R.m(t+dt)-R.m(t-dt))/(2*dt);
        else
            m_dot = 0;
        end
        Ir = R.Ir(t);
        if (t==0)
            Ir_dot = (R.Ir(dt)-R.Ir(0))/dt;
        elseif (t<bt)
            Ir_dot = (R.Ir(t+dt)-R.Ir(t-dt))/(2*dt);
        else
            Ir_dot = 0;
        end
        CM = R.cm(t);

        % Proprietes aerodynamiques
        % angle de rotation
        theta = 0;
        % angle d'attaque
        a = [sin(phi); cos(phi)];
        if (norm(Vi) == 0)
            alpha_tmp = 0;
        else
            cr = cross([a;0],[-Vi;0]);
            alpha_tmp = atan2(cr(3),dot(a,-Vi));
        end
        % coefficient de force normale et position du centre de pouss?e
        [CNa, CP] = R.aeroCoeff(alpha_tmp, Mach, theta);
        % force normale
        N = 0.5*rho*norm(Vi)^2*Aref*CNa*alpha_tmp;
        % force de trainee
        CD = R.CD;
        D = 0.5*rho*norm(Vi)^2*Aref*CD;

        % Proprietes du moteur
        if (t<R.Motor.ThrustCurve(1,1))
            Ft = 0;
        elseif (t<R.Motor.ThrustCurve(end,1));
            Ft = interp1(R.Motor.ThrustCurve(:,1)', R.Motor.ThrustCurve(:,2)', t);
        else
            Ft = 0;
        end
```

```
        epsilon = 0;

        %Calcul du calibre
        calibre_temp = (CP-CM)/d;

        % Forces dans le repere (n, a)
        % Poussee
        Force_T = [sin(epsilon); cos(epsilon)]*Ft;
        Force_N = [N;0];
        Force_D = [sin(alpha_tmp); -cos(alpha_tmp)]*D;
        Force_G = [sin(phi); -cos(phi)]*m*g;

        % Equation d'etat

        % launch
        if phase == 0
            % Force du rail sur la fusee
            Force_R = [sin(phi); 0]*m*g;

            X_dot   = Vx;
            Z_dot   = Vz;
            V_dot   = (Q*(Force_T+Force_N+Force_D+Force_G-Force_R) - m_dot*[Vx; Vz])/m;
            Vx_dot  = V_dot(1);
            Vz_dot  = V_dot(2);
            phi_dot = 0;
            phi_ddot= 0;

        % propulsed flight
        elseif phase == 1

            X_dot   = Vx;
            Z_dot   = Vz;
            V_dot   = (Q*(Force_T+Force_N+Force_D+Force_G) - m_dot*[Vx; Vz])/m;
            Vx_dot  = V_dot(1);
            Vz_dot  = V_dot(2);
            phi_dot = phi_dot;
            phi_ddot= -((N+D*sin(alpha_tmp))*(CP-CM)+sin(epsilon)*Ft*(L-CM)+Ir_dot*phi_dot)/Ir;


        % coast
        elseif phase == 2

            X_dot   = Vx;
            Z_dot   = Vz;
            V_dot   = (Q*(Force_N+Force_D+Force_G) - m_dot*[Vx; Vz])/m;
            Vx_dot  = V_dot(1);
            Vz_dot  = V_dot(2);
            phi_dot = phi_dot;
            phi_ddot= -((N+D*sin(alpha_tmp))*(CP-CM)+sin(epsilon)*Ft*(L-CM)+Ir_dot*phi_dot)/Ir;


        end

```

```
        deriv = [X_dot Z_dot Vx_dot Vz_dot phi_dot phi_ddot]';
    end



    %%%%%%%%%%%%%%%%%%
    % Status function
    %%%%%%%%%%%%%%%%%%

    function status = output(t,x,flag)
    % status = output(t, x, flag) function is called each time integrator
    % does a valid step.

        % as long as status = 0, ode45 will keep on runing... can be used
        % to create a stop condition.
        status = 0;

        % at initialization , and after each succesful step
        if isempty(flag) || strcmp(flag, 'init')

            % calculate angle of attack
            alpha(end+1) = alpha_tmp;
            calibre(end+1) = calibre_temp;
            Vx_dot_out = [Vx_dot_out, Vx_dot];
            Vz_dot_out = [Vz_dot_out, Vz_dot];

            % calculate flexion
            if(tquer_i<=length(S.tquer) & t>=S.tquer(tquer_i))
                Q = [cos(x(5)), sin(x(5)); -sin(x(5)), cos(x(5))];
                a = Q*[Vx_dot; Vz_dot+9.8];
                m = R.getAll_m(t(end));
                z = R.getAll_z();
                l = R.getAll_l();
                [T(tquer_i,:), M(tquer_i,:)] = Flexion(m', l', z', xquer, a(1), xquer(end));

                % increment query time index
                tquer_i = tquer_i + 1;
            end

        end

    end

    %%%%%%%%%%%%%%%%%%
    % Event functions
    %%%%%%%%%%%%%%%%%%

    function [value,isterminal,direction] = events_launch(tsim, Xsim)
        %fonction qui gere l'evenement d'arrivee en fin de rampe

        verif_x = Xsim(1)/sin(S.phi0) - S.L_ramp;
        verif_z = Xsim(2)/cos(S.phi0) - S.L_ramp;
        value = [verif_x, verif_z, 0, 0, 0, 0]; % On verifie pour x et z
        isterminal = [1, 1, 0, 0, 0, 0]; % stop the integration
        direction = [0, 0, 0, 0, 0, 0]; % negative direction

```

```
    end

    function [value, isterminal, direction] = events_flight(tsim, Xsim)
        % fonction qui detecte l'apogee

        Vz = Xsim(4);
        value = [0, 0, 0, Vz, 0, 0];
        isterminal = [0, 0, 0, 1, 0, 0];
        direction = [0, 0, 0, -1, 0, 0];

    end

end
```

B.4 DrawGraph.m

```

function DrawGraph(R, S, results)
% DrawGraph
%   Affiche la serie de graphe selon l'envie de l' utilisateur . Le choix est
%   dans l'INPUT S.G(i)
%   Notice : Color star in red is the end of the thrust
%            Color star in blue is the maximum height
%   INPUTS :
%       - R     : objet 'Rocket'
%       - S     : objet 'Simulation'
%       - results : structure de resultats de la simulation. Retourne par
%                   la fonction simulate().

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Definition des variables
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%
    tsim = results.tsim;
    Xsim = results.Xsim;
    alpha = results.alpha;
    calibre = results.calibre;
    M = results.flexion.M;


    %%%%%%%%%%%%%%%%%%%%%%
    % Points importants
    %%%%%%%%%%%%%%%%%%%%%%

    % fin de combustion
    i_burnout = min(find(results.tsim>R.Motor.bt));

    % apogee, ou conclusion de la simulation
    i_finSim = max(find(results.Xsim(:,2)));


    %%%%%%%%%%%%%%%%%%%%%%
    % Plots
    %%%%%%%%%%%%%%%%%%%%%%

    % Proprietes de masse et de poussee
```

```
    if S.getGraphValue('mass') == true
        t = linspace(0, max([R.Motor.bt])+15, 100);
        for it = 1:length(t)
            mass(it) = R.m(t(it));
            cm(it) = R.cm(t(it));
            Iz(it) = R.Iz(t(it));
            Ir(it) = R.Ir(t(it));
        end
        figure
        subplot(5,1,1)
        plot(t, mass);
        title('Mass properties');
        ylabel('m [kg]');
        subplot(5,1,2)
        plot(t, cm);
        ylabel('CM [m]');
        subplot(5,1,3)
        plot(t, Iz);
        ylabel('Iz [kg*m^2]');
        subplot(5,1,4)
        plot(t, Ir);
        ylabel('Ir [kg*m^2]');
        subplot(5, 1, 5)
        plot(R.Motor.ThrustCurve(1:end-1,1), R.Motor.ThrustCurve(1:end-1,2));
        ylabel('Ft [N]');
        xlabel('t [s]');
    end

    % Proprietes aerodynamiques
    if S.getGraphValue('aero') == true
        alphaAvantSimu = linspace(-30, 30, 100)*pi/180;
        figure
        for alpha_i = 1:length(alphaAvantSimu)
            [CNa(alpha_i), zCP(alpha_i)] = R.aeroCoeff(alphaAvantSimu(alpha_i), 0, 0);
        end
        subplot(2,1,1)
        plot(alphaAvantSimu*180/pi, CNa.*alphaAvantSimu);
        title('Aerodynamic properties at M = 0 & \theta = 0.')
        ylabel('CN');
        grid on;
        subplot(2,1,2)
        plot(alphaAvantSimu*180/pi, zCP);
        ylabel('zCP [m]');
        grid on;
        xlabel('alpha [^\circ]');
    end

    % Dessin de la fusee
    if S.getGraphValue('rocket') == true
        drawRocket(R);
    end

    % Altitude vs. temps
    if S.getGraphValue('altitude') == true
        figure;hold on;
        title('Altitude')
```

```
        plot(tsim(i_burnout,1), Xsim(i_burnout,2), '*r');
        plot(tsim(i_finSim,1), Xsim(i_finSim,2), '*b');
        plot(tsim, Xsim(:,2));
        xlabel('t [s]');
        ylabel('z [m]');
    end

    % Trajectoire
    if S.getGraphValue('trajectory') == true
        figure; hold on;
        title ('Rocket Trajectory')
        plot(Xsim(:,1), Xsim(:,2));
        plot(Xsim(i_burnout,1), Xsim(i_burnout,2), '*r');
        plot(Xsim(i_finSim,1), Xsim(i_finSim,2), '*b');
        daspect([1 1 1]);
        quiver(Xsim(1:100:end,1), Xsim(1:100:end,2), 0.001*sin(Xsim(1:100:end, 5)), 0.001*cos(
    Xsim(1:100:end, 5)));
        xlabel('horizontal distance from launchpad');
        ylabel(' vertical altitude');
    end

    % Angle par rapport ? la verticale
    if S.getGraphValue('verticalAngle') == true
        figure; hold on;
        title ('Rocket angle');
        plot(tsim(i_burnout,1), Xsim(i_burnout,5), '*r');
        plot(tsim(i_finSim,1), Xsim(i_finSim,5), '*b');
        plot(tsim, Xsim(:,5));
        xlabel('t [s]');
        ylabel('\phi [rad]');
    end

    % Angle d'attaque
    if S.getGraphValue('attackAngle') == true
        figure; hold on;
        title ('Angle of attack');
        plot(tsim(i_burnout,1), alpha(i_burnout), '*r');
        plot(tsim(i_finSim,1), alpha(i_finSim), '*b');
        plot(tsim, alpha);
        xlabel('t [s]');
        ylabel('\alpha [rad]');
    end

    % Marge de stabilite
    if S.getGraphValue('stability') == true
        figure; hold on;
        title (' Stabiltie statique lors du vol');
        plot(tsim(i_burnout,1), calibre(i_burnout), '*r');
        plot(tsim(i_finSim,1), calibre(i_finSim), '*b');
        plot(tsim, calibre);
        xlabel('t [s]');
        ylabel('Calibre [-]');
    end

    % Moment de flexion
    if S.getGraphValue('flexion') == true
```

```
        figure; hold on;
        title ('Flexion load');
        plot(results.xquer, M);
        legend(['t = ' num2str(S.tquer(1))], ['t = ' num2str(S.tquer(2))], ['t = ' num2str(S.tquer(3))], ['t = ' num2str(S.tquer(4))], ['t = ' num2str(S.tquer(5))])
        xlabel('x [m]');
        ylabel('M(x) [Nm]');
    end

end
```

Annexe C

Code source Teensy

```

/* Rocket flight measurement for Bachelor Project
 * Educational purpose
 * EPFL, Lausanne, Switzerland
 * Based on Adafruit Libraries & Exemples
 * Code written by Emilien Mingard, Mechanical Engineering Student
 */


/* General Libraries */
#include <Wire.h> // I2C communication
#include <SD.h> // SD card data saving
#include <SPI.h> // SPI communication, not needed

/* Sensors Libraries */
#include <Adafruit_Sensor.h>    // Global function made by Adafruit for all their sensors
#include <Adafruit_BMP085_U.h>  // Barometric and Temperature sensor
#include <Adafruit_LSM303_U.h>  // Accelerometer & Magnetometer sensors
#include <Adafruit_L3GD20_U.h>  // Gyro sensor

/* Global definition */
#define Nb_Param 12             // Number of measured parameters
#define Buffer_Length 330       // Ensure a block size less than 16KB
#define LEDPIN 13               //Led is on Teensy 3.5's pin number 13
#define LEDOn digitalWrite(LEDPIN,HIGH) // Create easy acces to LED
#define LEDOff digitalWrite(LEDPIN,LOW) // Create easy acces to LED
#define Bmp_Measure 30          // Indiquate number of measure without pressure
#define BUTTONPIN 2             // Choose the button to be on pin #2

/* Calibration */
#define Corr_Magn_x -4.38 // Correction Compute by MATLAB programm
#define Corr_Magn_y 11.95
#define Corr_Magn_z -0.75
#define Gain_accel_x 11.6347
#define Gain_accel_y 12.0021
#define Gain_accel_z 12.4891

/* Assign unique ID to sensors*/
Adafruit_BMP085_Unified bmp = Adafruit_BMP085_Unified(10085);          // Barometric
    and Temperature sensors
```

```
Adafruit_LSM303_Accel_Unified accel = Adafruit_LSM303_Accel_Unified(54321); // Accelerometer sensor
Adafruit_LSM303_Mag_Unified mag = Adafruit_LSM303_Mag_Unified(12345);       // Magnetometer sensor
Adafruit_L3GD20_Unified gyro = Adafruit_L3GD20_Unified(20);                // Gyro sensor

int main(){
  // Setup
  Serial.begin(9600); // Start Serial for debug

  /* Initialize the Led(s) and button(s) */
  pinMode(LEDPIN, OUTPUT); // Set the Led as an output
  pinMode(BUTTONPIN,INPUT); // Set the button as input
  LEDOn;delay(1000);LEDOff;delay(150);LEDOn;delay(150);LEDOff; // Led sequence for end of reboot

  /* SD card setup */
  File myFile;                                    // Current file to report
  const int chipSelect = BUILTIN_SDCARD;          // The board is the teensy 3.5
  Serial.print(" Initializing SD card...");
  if (!SD.begin(chipSelect)) {                    // Check if SD card
    Serial.println("SD: initialization failed");  // Report if an error occurs with SD card
    }
  Serial.println("SD: initialization done.");     // Confirm SD card

  /* Files names */
  if (SD.exists("VOL1")) {                        // Check if file
    SD.remove("VOL1");                            // Remove and create new to replace old one
    Serial.println("Test has been removed.");     // Security check if needed
    }

  /* New .txt file we use to write data */
  myFile = SD.open("VOL1", FILE_WRITE);           // Open or create .txt file
  if (myFile) {
    Serial.println("File: opening done");         // Check if file opened
    myFile.close();
    } else {
    Serial.println("File: opening failed");       // Report an error to open file
    }

  /* Initialize the sensors */
  bmp.begin();                    // Start barometer
  mag.enableAutoRange(true);      // Autorange
  mag.begin();                    // Start magnetometer
  accel.begin();                  // Start accelerometer
  gyro.enableAutoRange(true);     // Start gyroscope
  gyro.begin();                   // Autorange

  /* Initialize the state machine */
  enum{Measure,Write,Wait};       // States
  int state = Measure;            // Initial state
  bool wait = 0;                  // Condition for waiting state
  int count = 0;                  // Counter for the barometric sensor, use only few times

  /* Initialize data buffer */
  float data[Buffer_Length*Nb_Param];  // Initialize buffer to right length
```

```

/* Wait to Start Measurements */
int value = 0;                    // Loop to compare old and new value for the pin
int old_value = 0;
bool go = 1;
while(go){
  value = digitalRead(BUTTONPIN); // Start to measure when button is release
  if(value && !old_value){
    go = 0;
    }
  old_value = value;
  }
int T = millis();       // Save offset in time

// Loop
while(1){
  switch(state){
    case Measure:
    for(int i=0;i<Buffer_Length;i++){
      data[i*Nb_Param+0] = millis()-T;    // Use time offset
      sensors_event_t event;              // Get a new event at each iteration, event gives new value from I2C protocol

      /* Magnetic field measurement */
      mag.getEvent(&event);               // Request new values
      data[i*Nb_Param+1] = event.magnetic.x+Corr_Magn_x;
      data[i*Nb_Param+2] = event.magnetic.y+Corr_Magn_y;
      data[i*Nb_Param+3] = event.magnetic.z+Corr_Magn_z;

      /* Acceleration field measurement */
      accel.getEvent(&event);             // Request new values
      data[i*Nb_Param+4] = event.acceleration.x*Gain_accel_x;
      data[i*Nb_Param+5] = event.acceleration.y*Gain_accel_y;
      data[i*Nb_Param+6] = event.acceleration.z*Gain_accel_z;

      /* Gyroscope measurement */
      gyro.getEvent(&event);              // Request new values
      data[i*Nb_Param+7] = event.gyro.x;
      data[i*Nb_Param+8] = event.gyro.y;
      data[i*Nb_Param+9] = event.gyro.z;

      /* Pressure, Altitude & Temperature measurement */
      if(count%(Bmp_Measure-1) == 0){
        LEDOn;                            // Led is use to visualize when pressure is measured
        bmp.getEvent(&event);             // Request new values
        float temperature;                // Need to create float for temperature measurement
        bmp.getTemperature(&temperature);     // Get temperature
        data[i*Nb_Param+11] = temperature;     // Temperature has to be placed before any event.pressure
        data[i*Nb_Param+10] = event.pressure;  // Get the pressure
        }
      else{
        LEDOff;
        data[i*Nb_Param+10] = data[(i-1)*Nb_Param+10];  // Recopy old value when not
```

```
measured
        data[i*Nb_Param+11] = data[(i-1)*Nb_Param+11];  // Recopy old value when not measured
        }
      count++;                                  // Increment counter
      value = digitalRead(BUTTONPIN);           // Read button and save value
      if(value && !old_value){                  // Check if we want to wait
        wait = 1;
        }
      old_value = value;
      }
    if(wait){
      state = Wait;                             // If button was pressed
      }
    else{
      state = Write;                            // Still flying
    break;

    case Write:
    LEDOff;
    myFile = SD.open("VOL1",FILE_WRITE);                    // Open the file we use to store datas
    myFile.write( (const byte *) &data[0],4*Buffer_Length*Nb_Param);  // Write binary values
    myFile.close();                                         // Close file to saves datas
    state = Measure;
    count = 0;                                              // Reset barometric counter
    break;

    case Wait:
    LEDOn;          // Led code for waiting state
    break;
    }
  }
}
```

Annexe D

Script Python pour Abaqus

```
# -*- coding: mbcs -*-
from part import *
from material import *
from section import *
from assembly import *
from step import *
from interaction import *
from load import *
from mesh import *
from optimization import *
from job import *
from sketch import *
from visualization import *
from connectorBehavior import *
from Param import * # Importe les parametre depuis le fichier cree par MATLAB

# Définiton du Step
mdb.models['Model-1'].StaticStep(name='Efforts', previous='Initial')

# Génération du Tube
mdb.models['Model-1'].ConstrainedSketch(name='__profile__', sheetSize=2*Ltube)
mdb.models['Model-1'].sketches['__profile__'].ConstructionLine(point1=(0.0,
    -Ltube), point2=(0.0, Ltube))
mdb.models['Model-1'].sketches['__profile__'].FixedConstraint(entity=
    mdb.models['Model-1'].sketches['__profile__'].geometry[2])
mdb.models['Model-1'].sketches['__profile__'].Spot(point=(0.0, 0.0))
mdb.models['Model-1'].sketches['__profile__'].CoincidentConstraint(
    addUndoState=False, entity1=
    mdb.models['Model-1'].sketches['__profile__'].vertices[0], entity2=
    mdb.models['Model-1'].sketches['__profile__'].geometry[2])
mdb.models['Model-1'].sketches['__profile__'].FixedConstraint(entity=
    mdb.models['Model-1'].sketches['__profile__'].vertices[0])
mdb.models['Model-1'].sketches['__profile__'].Line(point1=(300.0, -50.0),
    point2=(300.0, -475.0))
mdb.models['Model-1'].sketches['__profile__'].VerticalConstraint(addUndoState=
    False, entity=mdb.models['Model-1'].sketches['__profile__'].geometry[3])
mdb.models['Model-1'].sketches['__profile__'].VerticalDimension(textPoint=(
    -158.773803710938, -33.088134765625), value=0.0, vertex1=
    mdb.models['Model-1'].sketches['__profile__'].vertices[0], vertex2=
    mdb.models['Model-1'].sketches['__profile__'].vertices[1])
```

```
mdb.models['Model-1'].sketches['__profile__'].ObliqueDimension(textPoint=(
    456.475219726563, -156.617553710938), value=Ltube, vertex1=
    mdb.models['Model-1'].sketches['__profile__'].vertices[1], vertex2=
    mdb.models['Model-1'].sketches['__profile__'].vertices[2])
mdb.models['Model-1'].sketches['__profile__'].DistanceDimension(entity1=
    mdb.models['Model-1'].sketches['__profile__'].geometry[3], entity2=
    mdb.models['Model-1'].sketches['__profile__'].geometry[2], textPoint=(
    74.9766845703125, 295.588256835938), value=Dtube)
mdb.models['Model-1'].Part(dimensionality=THREE_D, name='Tube', type=
    DEFORMABLE_BODY)
mdb.models['Model-1'].parts['Tube'].BaseShellRevolve(angle=360.0,
    flipRevolveDirection=OFF, sketch=
    mdb.models['Model-1'].sketches['__profile__'])
del mdb.models['Model-1'].sketches['__profile__']

# Materiaux, CRFP
mdb.models['Model-1'].Material(name='CFRP')
mdb.models['Model-1'].materials['CFRP'].Elastic(table=((132400.0, 8160.0,
    0.321, 4040.0, 4040.0, 3537.0), ), type=LAMINA)
mdb.models['Model-1'].materials['CFRP'].Density(table=((1.439e-06, ), ))
mdb.models['Model-1'].CompositeShellSection(idealization=NO_IDEALIZATION,
    integrationRule=GAUSS, layup=(SectionLayer(thickness=plythickness,
    orientAngle=-45.0, numIntPts=3, material='CFRP'), SectionLayer(
    thickness=plythickness, numIntPts=3, material='CFRP'), SectionLayer(thickness=
    plythickness,
    orientAngle=45.0, numIntPts=3, material='CFRP')), name='CFRP',
    poissonDefinition=DEFAULT, preIntegrate=OFF, symmetric=False, temperature=
    GRADIENT, thicknessModulus=None, thicknessType=UNIFORM, useDensity=OFF)
mdb.models['Model-1'].parts['Tube'].Set(faces=
    mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
    ), name='Set-1')
mdb.models['Model-1'].parts['Tube'].SectionAssignment(offset=0.0, offsetField=
    '', offsetType=BOTTOM_SURFACE, region=
    mdb.models['Model-1'].parts['Tube'].sets['Set-1'], sectionName='CFRP',
    thicknessAssignment=FROM_SECTION)
mdb.models['Model-1'].parts['Tube'].MaterialOrientation(additionalRotationType=
    ROTATION_NONE, axis=AXIS_1, fieldName='', localCsys=None, orientationType=
    GLOBAL, region=Region(
    faces=mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(mask=(
    '[#1 ]', ), )))

# Field output
mdb.models['Model-1'].FieldOutputRequest(createStepName='Efforts', name=
    'Ply_Stress', sectionPoints=(1, 2, 3, 4, 5, 6, 7, 8, 9), variables=('S',
    'MISES', 'MISESMAX', 'U', 'RF'))

# Agencement de la pièces dans Assembly et BC d'encastrement sur la thrust plate
mdb.models['Model-1'].rootAssembly.DatumCsysByDefault(CARTESIAN)
mdb.models['Model-1'].rootAssembly.Instance(dependent=ON, name='Tube-1', part=
    mdb.models['Model-1'].parts['Tube'])
mdb.models['Model-1'].rootAssembly.Set(edges=
    mdb.models['Model-1'].rootAssembly.instances['Tube-1'].edges.getSequenceFromMask(
    ('[#4 ]', ), ), name='Set-1')
mdb.models['Model-1'].DisplacementBC(amplitude=UNSET, createStepName='Efforts',
    distributionType=UNIFORM, fieldName='', fixed=OFF, localCsys=None, name=
    'Thrust Plate', region=mdb.models['Model-1'].rootAssembly.sets['Set-1'],
```

```
    u1=0.0, u2=0.0, u3=0.0, ur1=UNSET, ur2=UNSET, ur3=UNSET)

# Effort Réservoire
mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posuptank,
    principalPlane=XZPLANE)
mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posdowntank,
    principalPlane=XZPLANE)
mdb.models['Model-1'].rootAssembly.regenerate()
mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
    mdb.models['Model-1'].parts['Tube'].datums[4], faces=
    mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
    ))
mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
    mdb.models['Model-1'].parts['Tube'].datums[5], faces=
    mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
    ))
mdb.models['Model-1'].rootAssembly.Surface(name='Surf-1', side1Faces=
    mdb.models['Model-1'].rootAssembly.instances['Tube-1'].faces.getSequenceFromMask(
    ('[#2 ]', ), ))
mdb.models['Model-1'].SurfaceTraction(createStepName='Efforts',
    directionVector=((0.0, 1.0, 0.0), (0.0, 0.0, 0.0)), distributionType=
    UNIFORM, field='', localCsys=None, magnitude=load[0], name='Tank', region=
    mdb.models['Model-1'].rootAssembly.surfaces['Surf-1'], traction=GENERAL)

# Effort Modules, defini que si la masse est positive
if load[1] > 0.0:
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posup[0],
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posup[0]+ebague,
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posdown[0]-
    ebague,
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posdown[0],
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[8], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#4 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[9], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[10], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[11], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].rootAssembly.regenerate()
    mdb.models['Model-1'].rootAssembly.Surface(name='Surf-2', side1Faces=
        mdb.models['Model-1'].rootAssembly.instances['Tube-1'].faces.getSequenceFromMask(
        ('[#a ]', ), ))
    mdb.models['Model-1'].SurfaceTraction(createStepName='Efforts',
```

```
        directionVector=((0.0, 1.0, 0.0), (0.0, 0.0, 0.0)), distributionType=
        UNIFORM, field='', localCsys=None, magnitude=load[1], name='Module 1', region=
        mdb.models['Model-1'].rootAssembly.surfaces['Surf-2'], traction=GENERAL)

if load[2] > 0.0:
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posup[1],
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posup[1]+ebague,
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posdown[1]-
    ebague,
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posdown[1],
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[16], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[17], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[18], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[19], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].rootAssembly.regenerate()
    mdb.models['Model-1'].rootAssembly.Surface(name='Surf-3', side1Faces=
        mdb.models['Model-1'].rootAssembly.instances['Tube-1'].faces.getSequenceFromMask(
        ('[#a ]', ), ))
    mdb.models['Model-1'].SurfaceTraction(createStepName='Efforts',
        directionVector=((0.0, 1.0, 0.0), (0.0, 0.0, 0.0)), distributionType=
        UNIFORM, field='', localCsys=None, magnitude=load[2], name='Module 2', region=
        mdb.models['Model-1'].rootAssembly.surfaces['Surf-3'], traction=GENERAL)

if load[3] > 0.0:
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posup[2],
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posup[2]+ebague,
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posdown[2]-
    ebague,
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posdown[2],
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[24], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[25], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
```

```
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[26], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[27], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].rootAssembly.regenerate()
    mdb.models['Model-1'].rootAssembly.Surface(name='Surf-4', side1Faces=
        mdb.models['Model-1'].rootAssembly.instances['Tube-1'].faces.getSequenceFromMask(
        ('[#a ]', ), ))
    mdb.models['Model-1'].SurfaceTraction(createStepName='Efforts',
        directionVector=((0.0, 1.0, 0.0), (0.0, 0.0, 0.0)), distributionType=
        UNIFORM, field='', localCsys=None, magnitude=load[3], name='Module 3', region=
        mdb.models['Model-1'].rootAssembly.surfaces['Surf-4'], traction=GENERAL)

if load[4] > 0.0:
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posup[3],
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posup[3]+ebague,
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posdown[3]-
    ebague,
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posdown[3],
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[32], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[33], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[34], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[35], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].rootAssembly.regenerate()
    mdb.models['Model-1'].rootAssembly.Surface(name='Surf-5', side1Faces=
        mdb.models['Model-1'].rootAssembly.instances['Tube-1'].faces.getSequenceFromMask(
        ('[#a ]', ), ))
    mdb.models['Model-1'].SurfaceTraction(createStepName='Efforts',
        directionVector=((0.0, 1.0, 0.0), (0.0, 0.0, 0.0)), distributionType=
        UNIFORM, field='', localCsys=None, magnitude=load[4], name='Module 4', region=
        mdb.models['Model-1'].rootAssembly.surfaces['Surf-5'], traction=GENERAL)

if load[5] > 0.0:
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posup[4],
        principalPlane=XZPLANE)
```

```
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posup[4]+ebague,
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posdown[4]-
    ebague,
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posdown[4],
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[40], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[41], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[42], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[43], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].rootAssembly.regenerate()
    mdb.models['Model-1'].rootAssembly.Surface(name='Surf-6', side1Faces=
        mdb.models['Model-1'].rootAssembly.instances['Tube-1'].faces.getSequenceFromMask(
        ('[#a ]', ), ))
    mdb.models['Model-1'].SurfaceTraction(createStepName='Efforts',
        directionVector=((0.0, 1.0, 0.0), (0.0, 0.0, 0.0)), distributionType=
        UNIFORM, field='', localCsys=None, magnitude=load[5], name='Module 5', region=
        mdb.models['Model-1'].rootAssembly.surfaces['Surf-6'], traction=GENERAL)

if load[6] > 0.0:
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posup[5],
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posup[5]+ebague,
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posdown[5]-
    ebague,
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].DatumPlaneByPrincipalPlane(offset=posdown[5],
        principalPlane=XZPLANE)
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[48], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[49], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[50], faces=
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].parts['Tube'].PartitionFaceByDatumPlane(datumPlane=
        mdb.models['Model-1'].parts['Tube'].datums[51], faces=
```

```
        mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask(('[#1 ]', ),
        ))
    mdb.models['Model-1'].rootAssembly.regenerate()
    mdb.models['Model-1'].rootAssembly.Surface(name='Surf-7', side1Faces=
        mdb.models['Model-1'].rootAssembly.instances['Tube-1'].faces.getSequenceFromMask(
        ('[#a ]', ), ))
    mdb.models['Model-1'].SurfaceTraction(createStepName='Efforts',
        directionVector=((0.0, 1.0, 0.0), (0.0, 0.0, 0.0)), distributionType=
        UNIFORM, field='', localCsys=None, magnitude=load[6], name='Module 6', region=
        mdb.models['Model-1'].rootAssembly.surfaces['Surf-7'], traction=GENERAL)

# Meshing
mdb.models['Model-1'].parts['Tube'].setMeshControls(elemShape=QUAD, regions=
    mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask((
    '[#7 fffff ]', ), ), technique=SWEEP)
mdb.models['Model-1'].parts['Tube'].setElementType(elemTypes=(ElemType(
    elemCode=S8R, elemLibrary=STANDARD), ElemType(elemCode=STRI65,
    elemLibrary=STANDARD)), regions=(
    mdb.models['Model-1'].parts['Tube'].faces.getSequenceFromMask((
    '[#7 fffff ]', ), ), ))
mdb.models['Model-1'].parts['Tube'].seedPart(deviationFactor=0.1,
    minSizeFactor=0.1, size=h)
mdb.models['Model-1'].parts['Tube'].generateMesh()

# Début de l'étude
mdb.Job(atTime=None, contactPrint=OFF, description='', echoPrint=OFF,
     explicitPrecision =SINGLE, getMemoryFromAnalysis=True, historyPrint=OFF,
    memory=90, memoryUnits=PERCENTAGE, model='Model-1', modelPrint=OFF,
    multiprocessingMode=DEFAULT, name='Job-1', nodalOutputPrecision=SINGLE,
    numCpus=2, numDomains=6, numGPUs=0, queue=None, resultsFormat=ODB, scratch=
    '', type=ANALYSIS, userSubroutine='', waitHours=0, waitMinutes=0)
mdb.jobs['Job-1'].submit(consistencyChecking=OFF)
```

Annexe E

Divers acquis

Various things learned on the way and to be considered in rocket design…

Bachelor project "Rocket Science"
Summer semester 2017
EPFL
Brunner Eric
Burkhard Jonathan
Eckstein Maxime
Mingard Emilien
Rey Julien

Assembly organisation

Checklists: good idea to have checklists for everything! Workshop assembly, transportation preparation, site assembly, pre-flight tests, launch pad preparation, etc…

Tasks order: good idea to have rocket assembly procedure with clear task order (i.e. gant chart) so everybody knows what to do and when. Nothing more unpleasant to have to disassemble parts and modules again and again because something else needed to go inside first…

Roles & Resp: tasks should have a responsible (event. deputy as well).

Toolbox: on top of sourcing parts for the rockets, all required tools need to be sourced as well, good idea to have a checklist with minimum toolbox content for site assembly (could be shipped overseas or bought locally: warning: metric vs imperial system)

Assembly

Loctite: secure all screws with Loctite (all assembly screws that are not supposed to be disassembled)

Electrical connections: use only secured electrical connectors and NO screw terminal connections strips, they can get loose.

Li-Poly batteries: do not use (may burn on hard landings)

Nose cone: no carbon fiber material for nose cone if antenna inside (faraday cage!) better in glass fiber or PE

Electronics: foam or silent bloc mounts for all electronic cards/equipment

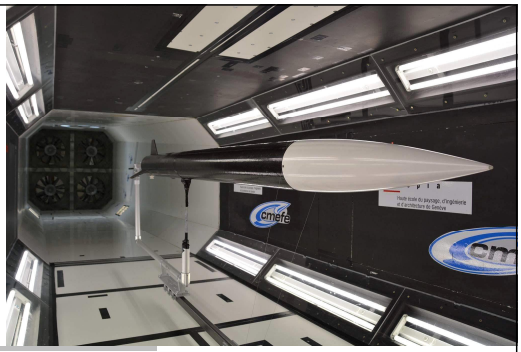
Wind tunnel

Preparation: wind tunnel testing is expensive, usually very little time is available for testing, therefore team needs to be very well prepared.

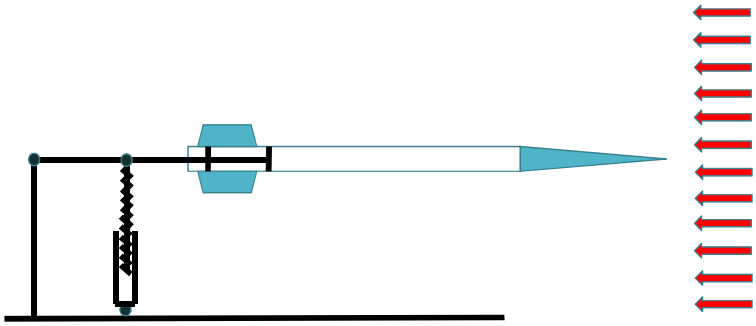
Protocol: what we want to test needs to be defined clearly so we don't get lost or miss some data afterward (i.e. missing drag measure or the setup without the rocket, etc…)

Setup: all material to support the rocket in the wind tunnel need to be fabricated as well: need to be adjustable (test the lift for diff. angles) and need to be very rigid to avoid bad measurements. Setup also need to be out of the flow as much as possible i.e. rocket is maintain into horizontal position from the back (by the motor tube or thrust plate).

Wind tunnel



Hepia Genève



Documents and presentation

Naming: be careful and make sure everybody always knows which rocket we are talking about (prototypes, wind tunnel model, competition rocket, backup rocket, etc…) in every discussions, presentations, reports, analysis, modelisations, simulations, etc…

Documentation habits: in large projects, it is very useful if everybody is familiar and makes good use of numbering, revision index, versioning, etc… Documentation guideline would be excellent to have before starting the project.

Helpful mistakes: documentation of what works and what not for next generation is very important to avoid same mistakes again. Even more important since team members are changing over the years.

Small margin items : ensure to have available support documentation/calculations for all small margin items: Everything that is close to rules level (without much security coefficient) needs strong proof it will work (else no one will let you fly).

Documents and presentation

Competition documents: Don’t neglect the documentation for the competition since it’s worth 50% of the judges grade.

Site conditions

Warning for electronic equipment: very high temperature in black bodies at sun (i.e. black rocket sitting half an hour on launch pad) some electronic components may not like it.

Warning for GPS units: after crossing the Atlantic: seems the would need to be powered for a while (1h) before they can be used accurately.

Warning for motor (any type): high environmental temperatures or internal temperatures might facilitate combustion. Therefore the resultant thrust profile might vary up to 20% from the expected one.

Structure

Glue at high temperatures: Glue might release if heated to high temperatures. This is particularly critical if the fins are glued to the motor mount.

CFRP parts with RUAG:

1. Present your needs, requirements, layup plan and plan of construction.
2. Assembly on site shouldn't take more than a day.
3. 6 weeks of Pre-Notice is recommended.

System

Mass Margin: The system must have a mass margin (stability wise), such that trim weight can be added to ensure stability and even achieve a closer apogee to the target.

Simulation

Hand calculations: Always provide proof that a simulation gives reasonable results by presenting “back of the envelop” type justifications that at least provide an idea for the expected order of magnitude. For example, go through the barrowman equations by hand at least once if using OpenRocket.

Critical points: Stability margin, angle of attack, velocity when clearing the launch rail, apogee, time to apogee, drift, …

Electronics

Microcontroller connection: The UART connection to the microcontroller must be readily accessible from some outer surface of the launch vehicle to be able to debug or flash the controller with last minute updates.

Soldering: Soldering should be analyzed with a microscope to be 100% sure that it will hold.

Literature

Title	Description	link
Handbook of model rocketry, Harry G. Stine and Bill Stine	Handbook written by the founder of the NAR and model rocketry as we know it today. Good starting point if you have no idea what rocketry is.	-
The theoretical prediction of the center of pressure, James. S. and Judith A. Barrowman	Original paper behind center of pressure formulas.	-
Calculating the center of pressure of a rocket, James Barrowman, Centuri, technical report 33	Comprehensive explanation on how to calculate the CP position and why it's so important.	https://drive.google.com/drive/u/1/folders/0B0AvWensgD8aS2p6VWRiZE1oUDg
OpenRocket	Free rocket flight simulation program. View technical documentation for in depth presentation of used models.	http://openrocket.sourceforge.net/

Literature

Title	Description	link
Atmospheric and space flight dynamics (Matlab and Simulink), Ashish Tewari	Detailed explanations of models to simulate flight dynamics of trans-atmospheric and space vehicles.	-
PDAS (Public domain aeronautical software)	Public domain code to predict flight vehicle properties (includes DATCOM, but not missile DATCOM)	http://www.pdas.com/datcom.html
1976, US Standard atmosphere	Original 1976, NASA, standard atmosphere document.	https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19770009539.pdf
Rocket Propulsion Elements, George Paul Sutton	All you need to know about the different types of rocket propulsion, burn models, chemistry, etc...	-

Annexe F

Collaboration multi-projets

Collaboration with Avionics Team

Summary of cross project activities : Air-brakes test

Bachelor project “Rocket Science”

Summer semester 2017

EPFL

Brunner Eric

Burkhard Jonathan

Eckstein Maxime

Mingard Emilien

Rey Julien

+

Bachelor project “Rocketry Avionics”

Summer semester 2017

EPFL

David Jule

Dussaux Pierre

Jamar Lucas

Jerinic Milos

Context

In 2017 two bachelor projects are directly related to the build, launch and recovery of a rocket for IREC competition (summer 2018)

Teams:

- 5 members of EPFL Rocket Team / Structure (J. Burkhard, E. Mingard, E. Brunner, M. Eckstein, J. Rey) are all involved in the same bachelor project with professor J. Cugnoni.
- 4 members of EPFL Rocket Team / Avionics (L. Jamar, P. Dussaux, J. David, M. Jerinic) are all involved in a control bachelor project with professor P. Muhlhaupt.

Interaction:

- Time has been spent between the members of the 2 teams. It could seems out of scope but it is both very important and very enriching for everyone involved. This documents intends to collect traces of common discussions and collaborative work.

Activities

Experience and knowledge sharing about our on board data recording module (hard and software)

Structure coaching, cross team brainstorming, design reviews

Aero and stability knowledge sharing

Understanding of minimum simulation requirements on both side

Definition of data exchange between teams

On going developments

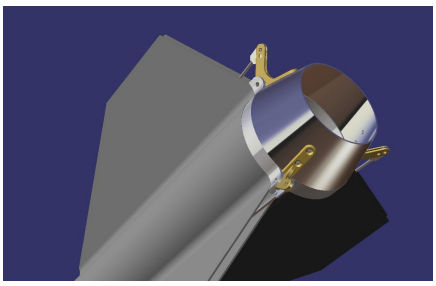
Rocket braking system IREC (option 1)

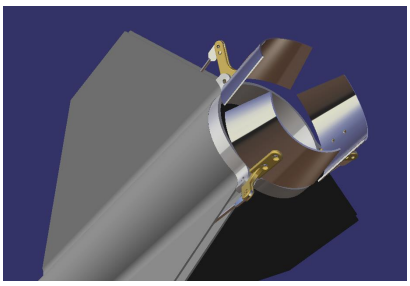
Boat tail integrated wings

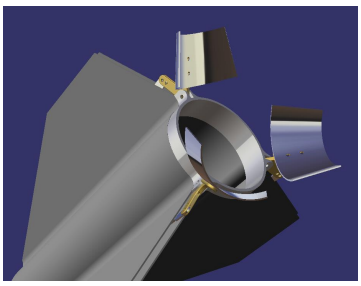
+ no aero or stability impact when closed

+ no main tube structure integrity issue

- complicated actuation along hot motor or throughout main fins







On going developments

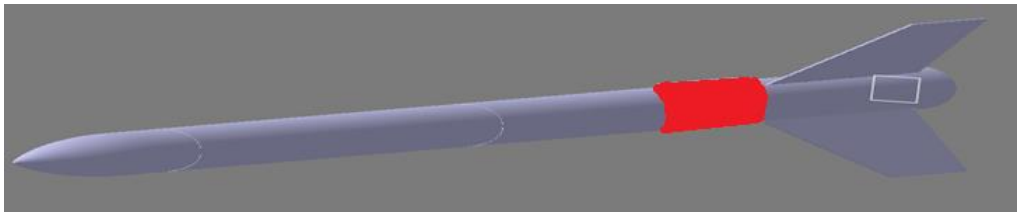
Rocket braking system IREC (option 2)

Specially built body section between N2O tank and motor

+ no aero impact

? Stability effect to be determined (air flow on fins when brakes are open)

+ easier direct actuation



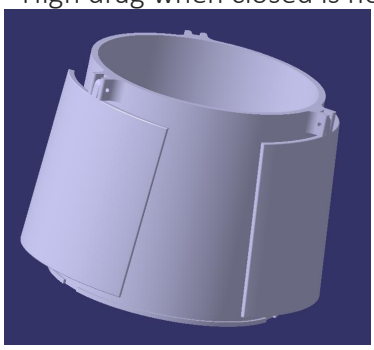
Prototype: proof of concept

Rocket Braking system test module for level 1 rockets

Optional module to be attached arround existing 4 inches rockets.

Braking wings to be aligned between the main fins

High drag when closed is not an issue as it is only a demonstrator



Prototype: build

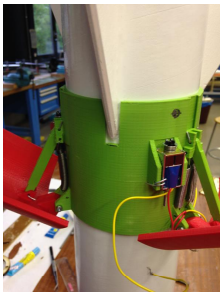
The rocket Braking system 1st prototype is a one-shot opening mechanism.

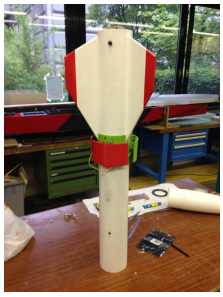
The control team 3D-printed the entire spring loaded mechanism and added solenoids to trigger the opening of the 3 wings.

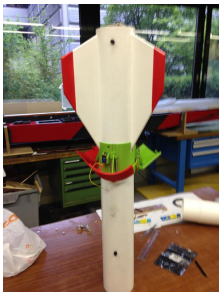
The rocket used for the test has been built and operated by the Rocket Science team during the real flight test.

Test fitting the mechanism of the rocket main tube.

Pictures of closed system and deployed system



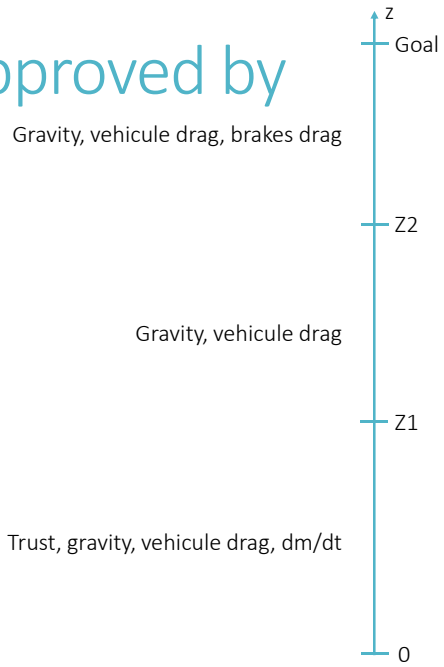




Strategy: discussed and approved by both teams

Braking strategy and simulation requirements:

- 3 stage flight:
  - *Boosted part*
  - *Free flight*
  - *Free flight with brake wings out*
- Pre-flight simulation:
  - *Estimate range for Z1, V1, Z2 to reach given goal*
  - *Create 2 entry table (Z1,V1) with Z2 values*
- On flight:
  - *Altimeter and accelerometer give Z1 and V1 (motor stop)*
  - *2 dimentions interpolation to fix Z2 (brake start point)*
  - *Altimeter compares Z2 (interpolated from table) wih live value and trigger brake opening*



Pre-flight simulation implementation

Phase 1: Integrate state model of powered flight with drag during the motor's burn time.

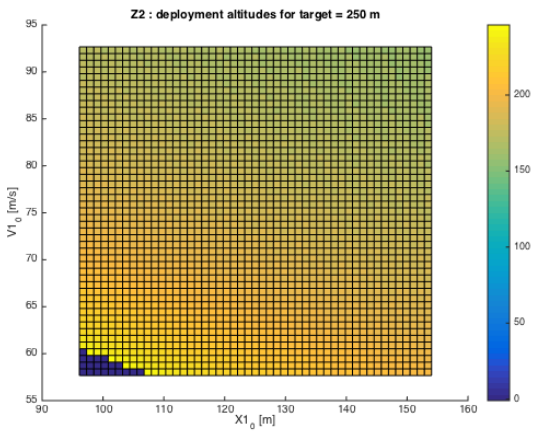
Phase 2: Analytically determine a set of apogees for a span of opening altitudes and burn out conditions that will contain the optimal brake deployment altitude.

Phase 3: Determine the optimum braking altitude for every plausible burn out conditions.

Pre-flight simulation : results

Study case:
- Julien's DX3 Madcow rocket
- Motor: Aerotech H123W
- Brake size: 50x200mm
- Number of brakes: 4
- Brake aerodynamic drag coeff: $C_D = 1$
- Target altitude: 250
- Motor uncertainty factor: 20%

Results:
- Extreme burn out states:
  - 96.13158 m, 57.6645 m/s
  - 154.0175 m, 92.6693 m/s
- Deployment Matrix quality (relative number of out of range values): 0.8%

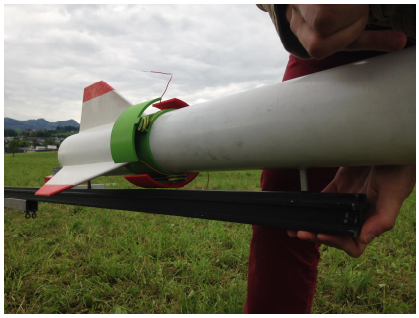


Dark Blue = out of range value (too low, too slow or too high, too fast).

Test day

Both teams actively participated on 20. May launch day in Zurich. The day ended with one successful launch of the rocket fitted with the braking system prototype.

Flight Card and motor spec.

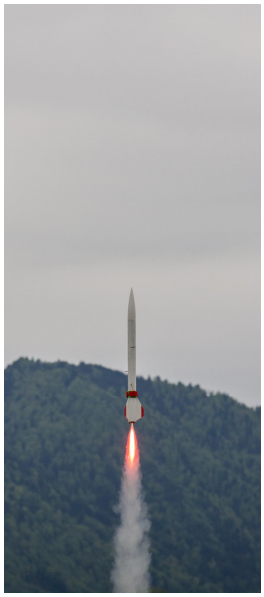


View of the extended rail buttons to clear the launch rail

Rocket on the launch rail with air-brake system armed

Test day

Jule, Julien, Lucas and Pierre prior flight



Lift-off

Happy recovery of the rocket after brake test

Annexe G

Certification niveau 1

Journée certification niveau 1 (+ mesures expérimentales)

01.04.2017

Bachelor project “Rocket Science”
Summer semester 2017
EPFL
Brunner Eric
Burkhard Jonathan
Eckstein Maxime
Mingard Emilien
Rey Julien

Organisation

Responsable Tripoli Suisse: Herr Thüring (+41794023916)

Tripoli registration: à faire par chacun online(70$ pour membership et assurance)

Transport: voiture privée jusqu’à Cernier (NE) départ EPFL ou à convenir

Prendre du cash pour taxes de lancement (25.- par x)

Pic-nic

Horaire

1er avril 11-17h (confirmation météo 24 mars matin / info sur slack)

2 avril 11-17h

Attention: rendez-vous le matin en avance! (rdv à fixer)

Certification pense-bête

Moteur 38mm imposés (casing récupérables, H ou I à définir selon fusée, Thüring prend un peu de tout)

CP (calcul et dessiné sur la fusée)

CG (sans moteur dessiné sur fusée + sur place avec moteur)

Mission (estimation apogée cible, modèle open-rocket prêt)

Vitesse max selon poids et parachute

Observation sur site

Launch pad (design, mise en place, leveling, etc…)

Ignition box and cables (what they use, etc…)

Procédure de lancement

Résumé déroulement de la journée

Mesures / capteurs

Accélération 3 axes

Magnéto 3 axes

Gyro 3 axes

Baromètre

Température (corr. baromètre)

Balance (masse fusée)

Validation simulation mission (altitude, vitesse, acc.), données fiables pour recovery?, apogée, trajectoire 3D, vitesse max (M<1), angle max, etc…

Outils, matériel

Balance de cuisine
Scotch
Zipties
Epoxy
Sangle parachute
Clés pour fixations moteur (fourche + inbus)
Tente?
PC avec Teensy et open-rocket
Chiffons
Graisse
WD40

Montage électronique

Schéma

Code Arduino

Initialisation/calibration (Procédure)

Procédure d'enclenchement/manipulation (start-stop acquisition)

Récupération données

Post-analysis

Download SD card to computer

Using matlab to analyse recorded results

Budget

Fusées: 4 kits level 1 et 2 à 150.-

Petit matériel, colle époxy, environ 50.- par fusée

Electronique embarqué: 1 module pour tous, 115.-

Moteur: 4 casing à 100.- + 4 grain load à 50.-

Inscriptions tripoli 4x 70.-

Taxes 4x 25.- (peut-être offert par Herr Thüring)

Essence: 2 voitures 60.-

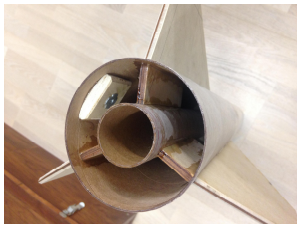
Budget journée: 2000.-

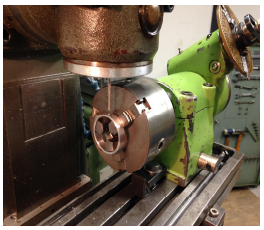
Détail Budget Elec.

ID	NAME	RETAILER	POWER	COST
IMU-2	Adafruit 9-DOF IMU Breakout - L3GD20 + LSM303	Play-Zone		24.9
BMP-1	Adafruit BMP180 Sensor (Pressure/Temperature/Altitude)	Play-Zone		13.9
MU-1	Teensy USB Board 3.5	Play-Zone		39.9
PS-3	LM2596 DC-DC Power Converter	Play-Zone	Input 5-36V -	12.9
Batterie	1600mAh	Hobbyking		15
SHIPPING				9
TOTAL				115.6

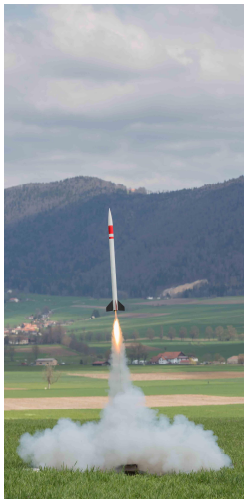
Construction

Différents kits de fusée en train d'être assemblés par les membres de l'équipe au LMAF (EPFL). Découpe, colle époxy et ponçage…





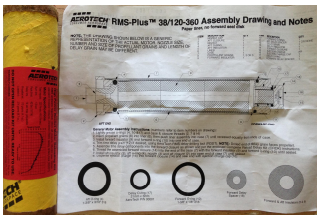
Launch day



Lift-off

Descent

Rocket Science Team with EPFL Rocket Team



Learning to put
together rocket motors

Annexe H

Certification niveau 2

Tripoli Level 2 Certification

-Checklist

-Ideas and technical solutions

Bachelor project "Rocket Science"
Summer semester 2017
EPFL
Brunner Eric
Burkhard Jonathan
Eckstein Maxime
Mingard Emilien
Rey Julien

Rules and Requirements for level 2 certification: ADMIN

Level 1 certification obtained

Valid Tripoli Rocketry Association membership

Ready to go throught the comprehensive written examination (prior to fly – see advance certif. Study guide for contents)

Motor ordered via local PREFECT or TAP (Mr. Turing in CH)

Rules and Requirements for level 2 certification: TECH

Have your own rocket built and ready for motor type J,K or L

Simulation done and motor choice confirmed

Static stablility calculation done with new motor weight and new avionics

Mandatory electronic chute ejection system for recovery implemented (depending on site regulations it might be required to have double deployment and/or redondant system)

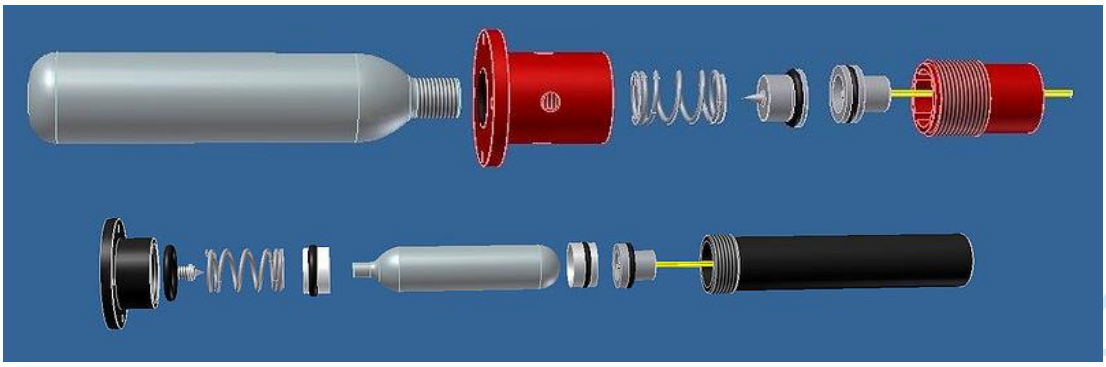
Ejection system on groud test done and ready for pre-flight demo

Ejection mechanisms ideas / solutions

CO2 cartridge system (explosive charge operated)

Arduino type micro-controlled (with timer or/and altimeter)

Market price ~ 150 CHF

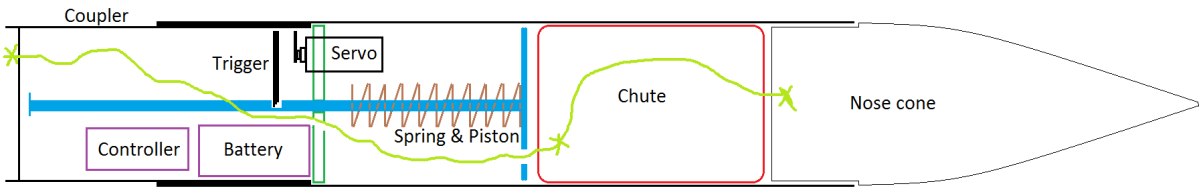


Ejection mechanisms ideas / solutions

Spring loaded (Servo operated trigger)

Arduino type micro-controlled (with timer or/and altimeter)

Cheaper and no explosive content



Ejection mechanisms ideas / solutions

Double mix deployments (drogue with motor charge and Servo operated trigger for main chute)

Arduino type micro-controlled Servo(with timer or/and altimeter)

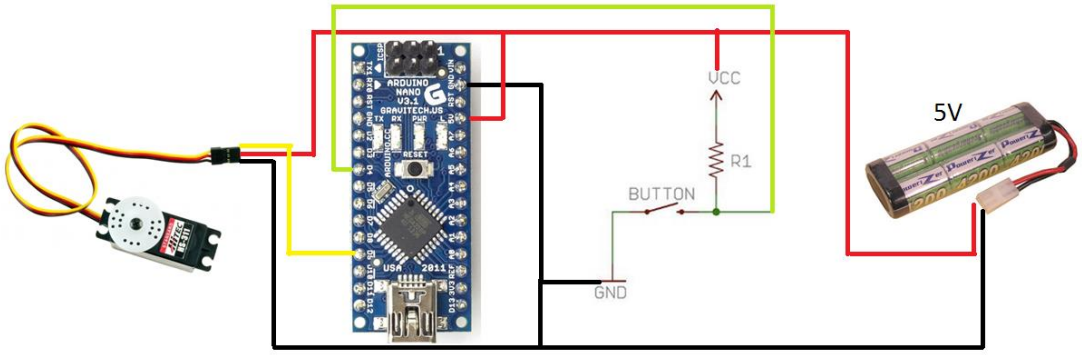
Drogue chute takes out the main chute (initially retained by trigger hook)

On board electronics: simple example

Arduino uno or nano or similar controller

DC supply (Li-ion battery with tension limiter)

Wiring diagram (with NO switch, can also be made for NC)



Timer trigger

Possible solutions

Wired to motor ignition:

-Not very precise, 12 or 24V <> controller IO capabilities

-Need a way to disconnect from launch pad

+Much more fancy

NC pin switch attached to launch pad:

+Real lift of time

+Powered by same 5V onboard power

+Pin slides off rocket and remain attached to launch pad

Accelerometer trigger:

+No need for outside rocket body connection

-Need to wire and code for accelerometer

-Risk to trigger it manipulating the rocket to launch pad (event, remove before flight switch)

Altimeter trigger

Altimeter trigger:

+No need for outside rocket body connection

+Help to solve the problem coming from bad drag coefficients used in simulation for timer estimation

-Need to wire and code for altimeter

(make sure there is a procedure to initialize the sensor, else the ground will not be your 0)

Note:

It is a very good idea to setup a system with dual triggering both on the sensor side and mechanical side.

i.e. altimeter trigger + timer as backup, or motor ejection + timer as backup, or any combinations that make sense and decrease the risk of malfunctioning and crash.

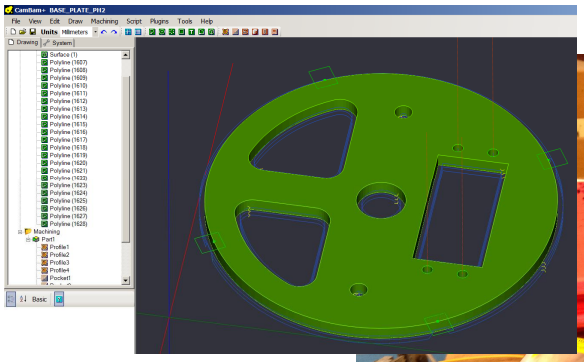
Code Lines

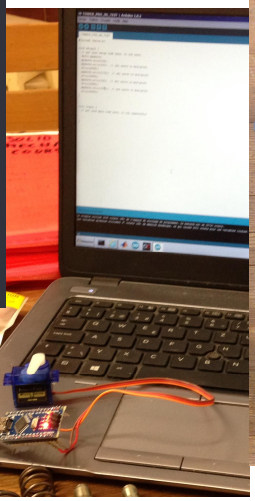
Arduino nano code simple example using the servo library

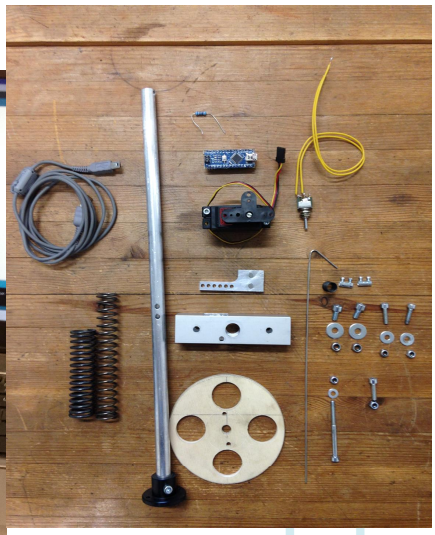
SERVO_CHUTE_TIMER

```
#include <Servo.h>

int chute_delay = 5;                                  //delay in sec

const int switchPin = 4;                               //Switch Connected to PIN 4
int switchState = 0;                                   // Variable for reading Switch status
Servo myservo;                                         //Declaration ServoType variable

int servoCLOSEangle = 10;                              // from 0 to 180
int servoOPENangle = 170;                              // from 0 to 180

void setup()
  {
  pinMode(LED_BUILTIN, OUTPUT);                         // initialize digital pin LED_BUILTIN as an output
  pinMode(switchPin, INPUT);                            //Switch PIN is input with PULLUP
  myservo.attach(9);                                    //Set pin for servo input
  myservo.write(servoCLOSEangle);                       //Initialize Servo to close posish
  }


void loop()
  {
  switchState = digitalRead(switchPin);                 //Reads the status of the switch.
  if (switchState == LOW)                               //If the switch is pressed
      {
      delay(chute_delay*1000);
      myservo.write(servoOPENangle);
      delay(2000);
      myservo.write(servoCLOSEangle);
      delay(1000);
      }
```

Building mechanism





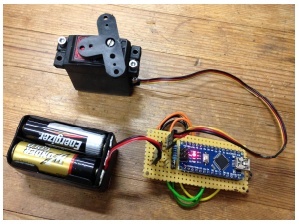


-some CAM for CNC machining

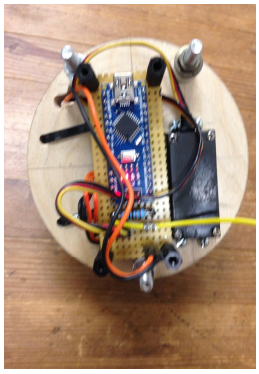
-Arduino testing

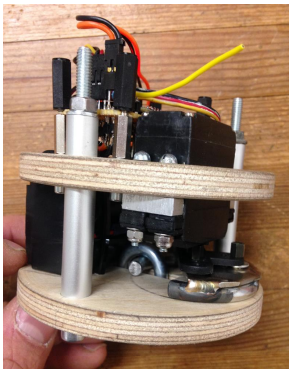
-material preparation

Building mechanism and chute



Test setup





Module fitting inside the rocket body

Making parachute…

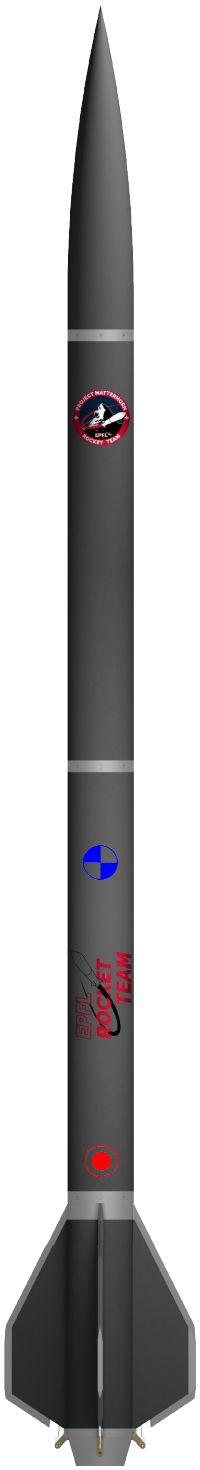
Launch day

Annexe I

Posters

ROCKET SCIENCE - AEROSTRUCTURE

Jonathan Burkhard, Eric Brunner, Maxime Eckstein, Emilien Mingard & Julien Rey
Dr. Joël Cugnoni



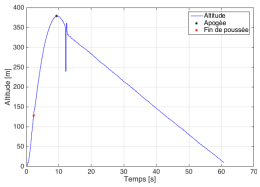
INTRODUCTION

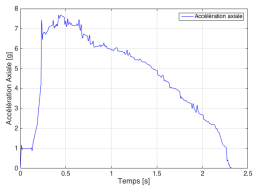
Dans le cadre d'activités liées aux fusées amateures qui ont récemment pris naissance au sein de l'EPFL, le projet Rocket Science a pour but de comprendre les enjeux principaux dans le dimensionnnement de la structure d'une fusée et de développer les outils nécessaires à effectivement prévoir la réussite d'un vol avec un tel véhicule.

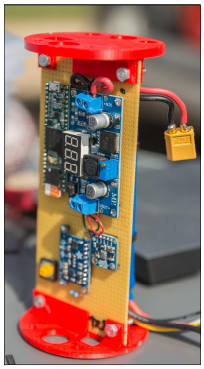
CAHIER DES CHARGES

SPEC	VALUE	FLEX	EMIT
Stabilité statique	1-2 calibres	0	IREC
Masse	9 [kg]	-5 to +1	ERT
Diametre	170 [mm]	0	ERT
Longueur	3300 [mm]	-700 to +200	ERT
Apogée	3048 [m]	0 to +300	PROJECT

SYSTÈME EMBARQUÉ



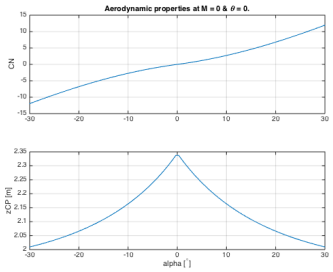




Pour valider les codes de simulations, plusieurs mesures ont été effectuées lors de vols d'essai. On souhaite comprendre les séquences composant un vol pour déterminer les efforts critiques agissant sur une fusée. Deux graphes illustrent les mesures d'altitude et d'accélération en vol. On peut reconnaître une courbe typique de poussée.

MARGE DE STABILITÉ

Afin d'assurer un vol stable, il faut vérifier que les forces aérodynamiques agissants sur la fusée contribuent à la redresser dans la direction de sa trajectoire plutôt que de la lancer dans une vrille. Pour ce faire, il faut que le centre de pression (point d'application des forces aérodynamiques normales résultantes) soit 1 à 2 calibres derrière le centre de masse à tous moments du vol.



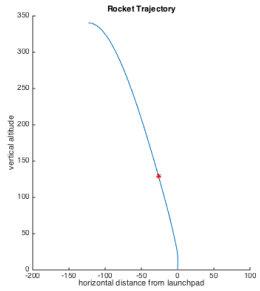
SIMULATION

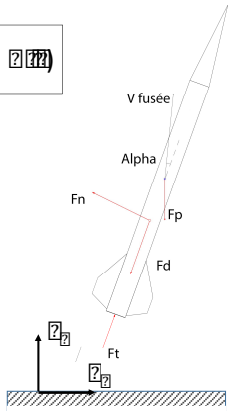
Pour prédire les performances de notre design, nous avons développé un code de simulation.

Objectifs:

- Å Créer un outil de dimensionnement
- Å Etudier la stabilité de la fusée
- Å Estimer la trajectoire
- Å Prévoir la sensibilité à des perturbations (vent)

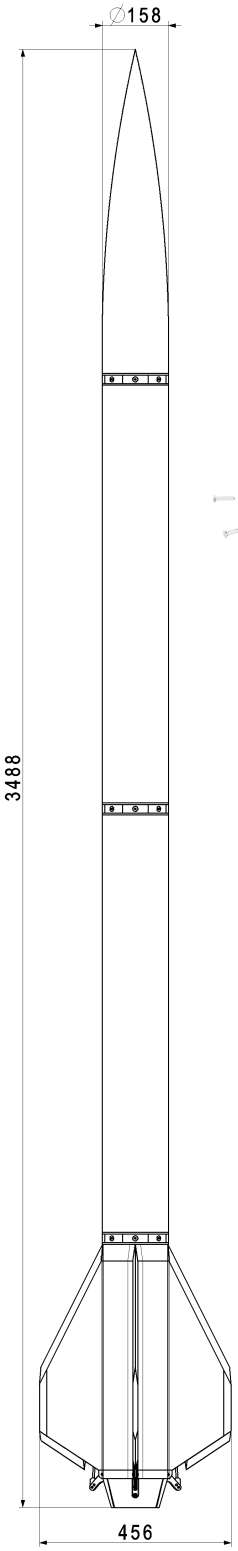
$$\frac{d\vec{V}}{dt} = \frac{1}{m}\left(\vec{F}_g + \vec{F}_p + \vec{F}_n + \vec{F}_d\right) - \vec{F}_t$$





ROCKET SCIENCE - AEROSTRUCTURE

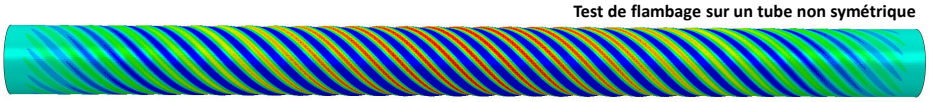
Jonathan Burkhard, Eric Brunner, Maxime Eckstein, Emilien Mingard & Julien Rey
Dr. Joël Cugnoni



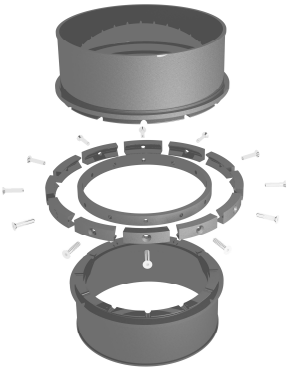
TUBE & NOSE CONE

Le tube et le nez sont le squelette d'une fusée, ils assurent l'intégrité de la structure. Lors de l'accélération, ils soutiennent l'entier des efforts produits par les modules, le moteur et le payload ainsi que les ailerons et le parachute. Les contraintes sont généralement élevées et une étude mécanique en flambage latéral et axial est nécessaire.

Test de flambage sur un tube non symétrique



COUPLEURS



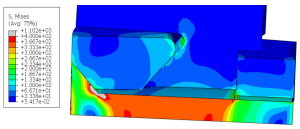
Réalisation d'un coupleur

Le cas de charge maximal sur le coupleur correspond à l'ouverture du parachute principal et représente une force de traction de 20kN.

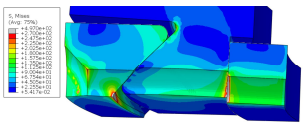
On assure l'intégrité de la fusée selon les critères suivants :

- Déplacement radial inférieur à 3% Résultat : 50µm
- Pas de plastification globale des éléments.
  - Vis M3x20-8.8
  - Pièce aluminium 6061-T6

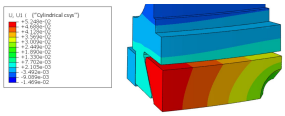
Contrainte dans la vis en acier



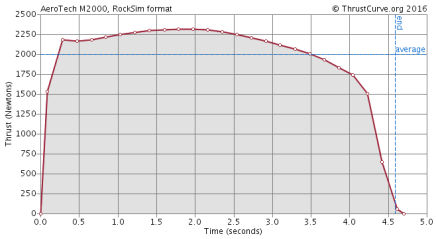
Contrainte dans l'assemblage alu



Déplacement radial sous traction

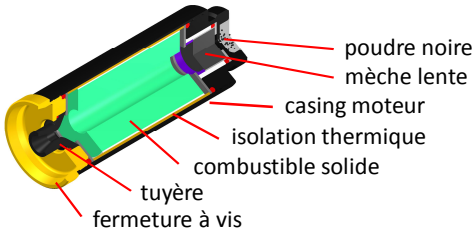


MOTEUR



Courbe de poussée typique

Moteur à combustible solide partiellement réutilisable



Les moteurs utilisés dans nos fusée de test étaient tous de type combustible solide. Le perchlorate d'ammonium et la poudre d'aluminium sont les deux principaux composants des recharges disponibles sur le marché. Le corps du moteur ainsi que les fermetures sont réutilisables alors que la tuyère et les joints sont à usage unique. Les courbes de poussée sont mises à disposition par les fabricants, ce qui est pratique pour les simulations.

AILERONS & AEROFREINS

Les ailerons baissent le centre de pression ce qui augmente la stabilité. A l'aide d'un contrôle actif, les aérofreins permettent de ralentir le bolide afin d'atteindre l'apogée souhaitée de façon plus précise.

Séquence d'ouverture des aérofreins

Sources:
1. Moteur : http://www.aerotech-rocketry.com

Bibliographie

[1] Jean-Philippe ANSERMET. *Traité de Physique, Mécanique*. Presses polytheciques romandes, 2013. ISBN : 978-2-88915-024-3.

[2] James. S. BARROWMAN et Judith A. BARROWMAN. *Theoretical prediction of the center of pressure*. R & D Project.

[3] Eric BRUNNER. *Rapport de projet Éléments finis : coupleur de fusée. Dimensionnement de la bague externe*.

[4] Maxime ECKSTEIN. *Rapport de projet Éléments finis : coupleur de fusée. Analyse de l'assemblage*.

[5] Jonathan Burkhard ERIC BRUNNER. *Code de simulation matlab*.

[6] Robert GALEJS. "Wind instability, What Barrowman left out". In : (2017). URL : `http://argoshpr.ch/joomla1/articles/pdf/sentinel39-galejs.pdf`.

[7] *List of moments of inertia*. Wikipedia. URL : `https://en.wikipedia.org/wiki/List_of_moments_of_inertia`.

[8] Sampo NISKANEN. "Development of an Open Source model rocket simulation software". Master Thesis. 2009.

[9] Sampo NISKANEN. *OpenRocket technical documentation*.

[10] *Parallel axis theorem*. Wikipedia. URL : `https://en.wikipedia.org/wiki/Parallel_axis_theorem`.

[11] "US Standard atmosphere, 1976". In : (1976).