Simulateur du vol d'une fusée en 3D en Python

Computer-Human Interaction in Learning and Instruction (CHILI)
EPFL

Projet de bachelor

Michaël TASEV

Superviseur·e·s :
Prof. Pierre Dillenbourg & Mme Hala Khodr

Janvier 2021

Table des matières

1 Introduction **1**

2 Structure du code **2**

2.1 `Simulator3D.py` 2

2.2 `Rocket` 3

2.3 `Functions` 3

2.4 `Motors et Parameters` 5

3 Comparaisons **6**

4 Difficultés rencontrées et améliorations **8**

5 Remerciements **9**

1 Introduction

L'EPFL Rocket Team (ERT) est une association d'étudiant·e·s ayant pour but la construction de fusées et la participation à des compétitions de lancement de ces dernières dans le cadre d'un projet interdisciplinaire de l'EPFL.

Lors de ces compétitions, différents objectifs peuvent être visés, notamment le lancement à 10'000 ou 30'000 pieds (respectivement 3048m et 9144m). Le lancement d'une fusée à ces hauteurs nécessite de savoir à quoi va ressembler le vol. C'est pourquoi l'ERT a développé un simulateur sur Matlab permettant de simuler le vol d'une fusée ainsi que récolter des données sur celui-ci.

Parmi les données intéressantes à récolter, on peut noter la position, la vitesse, le moment d'intertie, l'angle d'attaque de la fusée ou encore le coefficient de trainée (drag coefficient). Toutes ces données peuvent être exprimées en fonction du temps, mais aussi en fonction de l'altitude ou du coefficient de trainée (il sera par exemple plus utile d'avoir l'altitude en fonction du coefficient de trainée ou de la vitesse par exemple).

Afin de rendre l'utilisation du simulateur plus intuitive pour tout utilisateur, une interface graphique utilisateur (GUI) semblait indispensable. Matlab n'est pas spécialement conçu pour faire des interfaces graphiques, et le logiciel n'est pas gratuit. Le choix d'utilisation de Python semble naturel, de par sa facilité d'apprentissage et l'accessibilité des différents modules adaptés à nos besoins. Certains projets de bachelor se sont déjà occupés de traduire le simulateur 1D en python, ainsi que le début de la GUI [2] [1].

Le but de ce projet de semestre (ainsi que celui de Sayid Derder) est de passer le simulateur 3D en Python, et d'assurer une utilisation efficace de la GUI, c'est-à-dire de l'adapter au simulateur 3D et de lui ajouter des fonctionnalités, afin que le simulateur Python soit opérationnel et aussi précis que sur Matlab.

2 Structure du code

Le projet a été de créer l'architecture du simulateur 3D, créer les différents dossiers et classes, les traduire de Matlab, et s'assurer qu'elles fonctionnent effectivement.

Le simulateur 3D est composé d'une classe principale, `Simulator3D.py`, d'un dossier `Rocket` contenant des classes définissant la fusée, d'un dossier `Functions` contenant des fonctions d'assistance (helper functions, rangées en 2 dossiers principaux, `Math` et `Models`), deux dossiers `Motors` et `Parameters` et d'une classe de test `Test.py`. La classe `GUI ROCKET.py` s'occupe de l'intégration du simulateur 3D à l'interface graphique, améliorée par Sayid Derder dans le cadre de ce même projet de semestre.

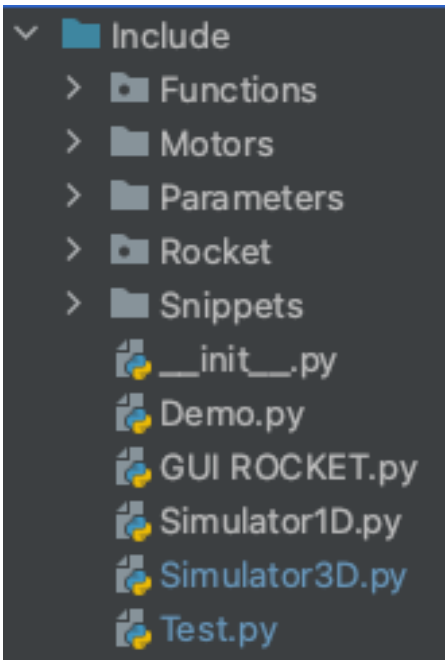


FIGURE 1 – Arborescence globale

2.1 `Simulator3D.py`

Cette classe comporte tout d'abord les différentes fonctions d'état de la fusée :

1. `Dynamics_Rail_1DOF`, décrit la fusée sur le rail.
2. `Dynamics_6DOF`, décrit la fusée en train de monter jusqu'à l'apogée.
3. `Dynamics_Parachute_3DOF`, décrit la fusée en train de descendre avec un parachute.

4. `Dynamics_3DOF`, décrit la chute de la fusée sans parachute.
5. `Nose_Dynamics_3DOF`, décrit la chute du nez de la fusée.
6. `Nose_Dynamics_6DOF`, décrit la chute du nez de la fusée, mais avec 6 degrés de liberté au lieu de 3.
7. `Payload_Dynamics_3DOF`, décrit la chute du paquet de la fusée.

Puis plusieurs fonctions appellent un intégrateur en donnant les fonctions d'état décrites ci-dessus comme paramètre, et un événement déclencheur qui marque la fin de l'intégration, `event` :

1. `RailSim` appelle `Dynamics_Rail_1DOF` avec `event` = `off_rail`, lorsque la fusée n'est plus sur le rail.
2. `FlightSim` appelle `Dynamics_6DOF` avec `event`=`apogee`, quand la fusée atteint son apogée.
3. `DrogueParaSim` appelle `Dynamics_Parachute_3DOF` avec `event` = `MainEvent`, qui est le moment où la fusée atteint 300m d'altitude et qu'elle déploie son parachute principal.
4. `MainParaSim` appelle aussi `Dynamics_Parachute_3DOF`, mais avec un `event` différent : `CrashEvent`, le moment où la fusée touche le sol.
5. `CrashSim`, `Nose_CrashSim_3DOF`, `Nose_CrashSim_6DOF`, et `PayloadCrashSim` appellent chacune leur fonction d'état correspondante avec toujours le même `event` : `CrashEvent`

2.2 `Rocket`

Le dossier `Rocket` comporte les classes `Body.py`, `Cone.py`, `Fins.py`, `Motor.py`, `Parachute.py`, `Rocket.py`, `Stage.py`. Toutes ces classes définissent les différents éléments de la fusée. Une `Rocket` est composée d'une liste de `Stage`, chaque `Stage` étant composé de listes de `Fins`, de `Motor`, et de `Parachute`.

Une dernière classe, `RocketFiled.py`, a été pensée pour prendre comme entrée un fichier texte contenant toutes les informations sur la fusée (comme sur Matlab) ; finalement ces fonctions ont été intégrées dans la classe `GUI ROCKET.py`

2.3 `Functions`

Le dossier `Functions` contient trois sous-dossiers, `Math`, `Models`, et `Utilities`. Dans `Math`, ce sont des classes qui contiennent des "simples" fonctions mathématiques, comme

le passage d'une matrice de rotation à son quaternion correspondant.

Dans `Models`, on trouve des fonctions qui traitent des comportements aerodynamiques de la fusée, comme `wind_model` qui simule le vent à un temps donné, ou `normal_lift` qui calcule l'intensité de la force normale s'appliquant au centre de pression de la fusée.

Le dernier dossier `Utilities` n'est pas utilisé car il comporte les fonctions utiles pour le fichier `RocketFiled.py`, que nous n'utilisons finalement pas. Il a été décidé de les laisser pour qu'un·e autre utilisateur·trice puisse en faire usage si besoin.

L'arborescence est visible sur la figure 2

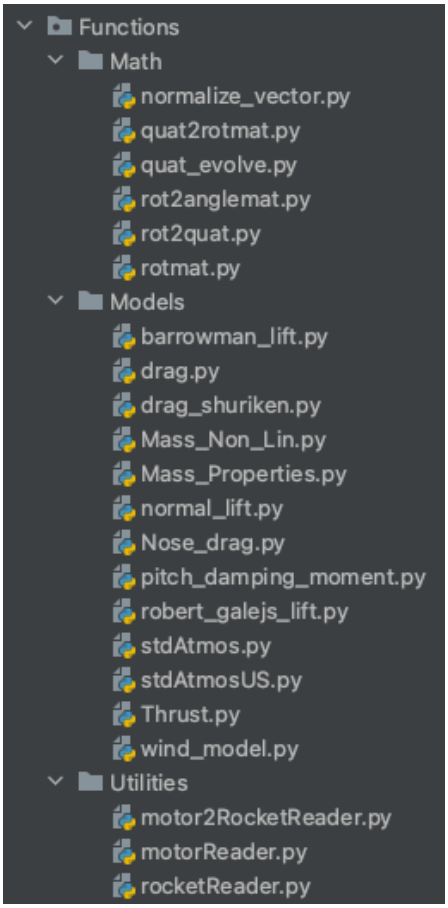


FIGURE 2 – Arborescence des fonctions utilitaires

Chacune de ces fonctions est documentée à l'aide d'une description de ce qu'elle fait, des paramètres d'entrée et de retour, de telle sorte qu'un·e nouvel·le utilisateur·trice

puisse comprendre le code, comme sur la figure 3 :

```
def normal_lift(rocket: Rocket, alpha, k, m, theta, Galejs):
    """
    Computes the normal force intensity applied to the center of pressure according to Barrowman's theory and
    corrections for extreme aspect ratio bodies proposed by Robert Galejs.

    Parameters
    ----------
    rocket          : Rocket object
    alpha           : angle of attack [rad]
    k               : Robert Galejs' correction factor
    m               : Mach number
    theta           : Roll angle [rad]
    Galejs          : Flag indicating use of Galejs' correction or not [1 or 0]

    Returns
    -------
    CNa             : Normal lift derivative versus delta coefficient derivative [1/rad]
    Xp              : Center of pressure
    CNa_barrowman   : Normal lift coefficient derivatives of rocket components according to Barrowman theory [1/rad]
    Xp_barrowman    : Center of pressure of rocket components according to Barrowman theory [1/rad]

    """
```

FIGURE 3 – La fonction `normal_lift` documentée

2.4 `Motors et Parameters`

Ces deux dossiers contiennent des fichiers textes auxquels `GUI ROCKET.py` accède pour lire les données et, en se basant sur celles-ci, créer une fusée. `GUI ROCKET.py` a aussi accès à ces fichiers en écriture lorsqu'un·e utilisateur·rice entre manuellement les données depuis l'interface graphique.

3 Comparaisons

Une fois le simulateur traduit, il faut s'assurer qu'il soit le plus adapté à nos besoins. Afin de le tester, nous avons simulé le vol avec la fusée ayant décollé en juillet 2020 à Wasserfallen, et avec l'environnement de Wasserfallen. Nous avons comparé les résultats du simulateur Python avec celui de Matlab, et avec différentes méthodes d'intégrations (en Pyhton). Pour choisir la méthode optimale, l'objectif est de trouver un équilibre entre la précision, le temps d'exécution, et la fluidité des courbes. Voici les données analysées :

FIGURE 4 – Données sur les différents intégrateurs

	Intégrateur	Apogée [m]	Écart relatif	Temps d'éxecution [s]	Nombre de pas d'intégration
Matlab	ode45	1635.9789	-	2.0884	45
Python	RK45	1631.8045	0.255 %	12.22	8
	RK23	1632.0798	0.238 %	12.86	9
	DOP853	1633.1227	0.175 %	16.98	8
	Radau	1632.0594	0.240 %	26.05	9
	BDF	1632.2800	0.226 %	16.65	19
	LSODA	1631.9795	0.244 %	15.38	17

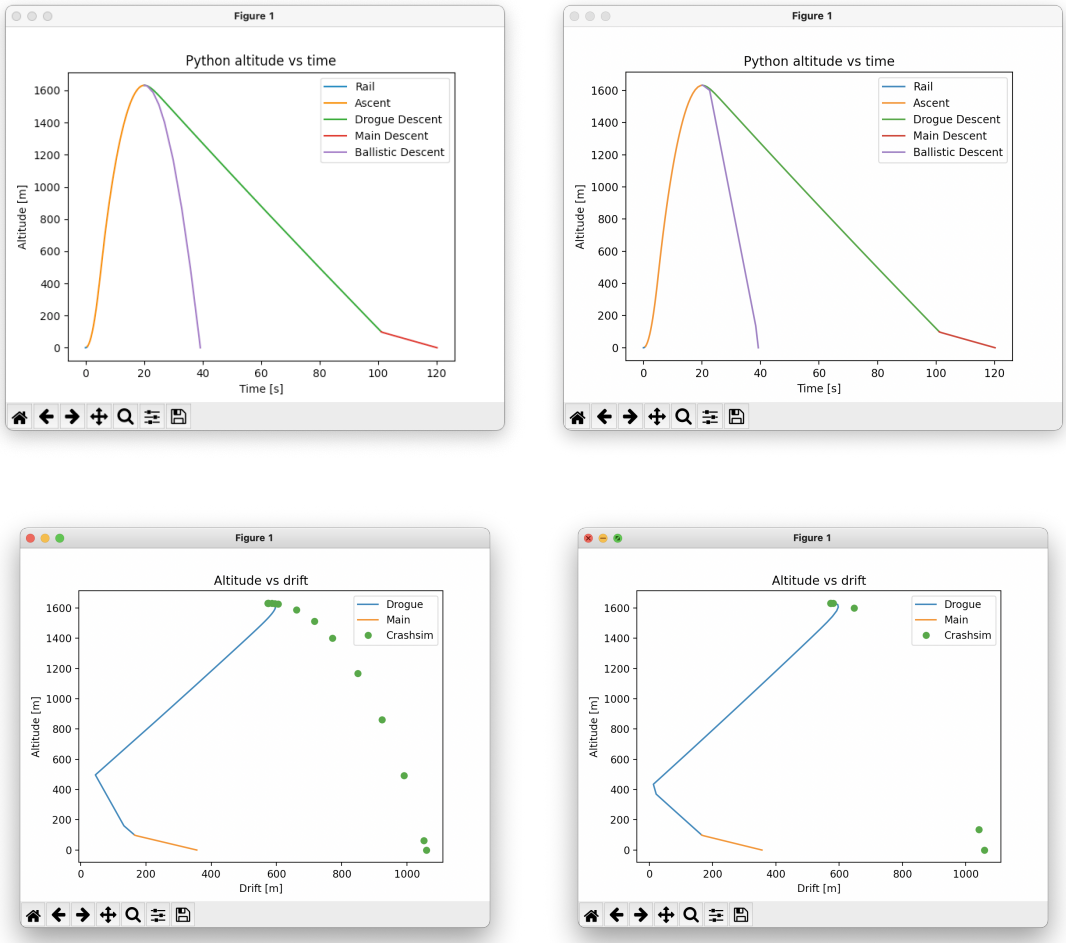
L'intégrateur correspond à la méthode d'intégration utilisée, l'apogée à la hauteur maximale atteinte par la fusée dans le simulateur.

L'écart relatif avec le simulateur de Matlab est faible, il n'est en absolu que de quelques mètres, et ce quel que soit l'intégrateur. C'est un résultat encourageant, cela signifie que l'on peut se baser sur d'autres critères pour choisir l'intégrateur, comme le temps d'exécution ou la qualité des graphiques.

Le temps d'exécution est un paramètre qui pose problème sur chacun des intégrateurs Pyhton testés, l'intégrateur le plus rapide étant 6x plus lent que sur Matlab. On peut cependant déjà observer que `Radau` est particulièrement lent, il est donc peu probable que nous l'utilisions.

Enfin, le nombre de pas d'intégration correspond au nombre de pas que l'intégrateur effectue lors de l'intégration de `Dynamics_3DOF`, soit la simulation du crash de la fusée. Dans les images ci dessous, on peut observer que 8 pas ne sont pas suffisants car cela ne fournit pas des données visuellement esthétiques, comme le montrent les images de

la figure 5. Ce paramètre permet d'écarter les intégrateurs ayant un nombre de pas insatisfaisant et réduit le choix à deux méthodes, BDF ou LSODA.

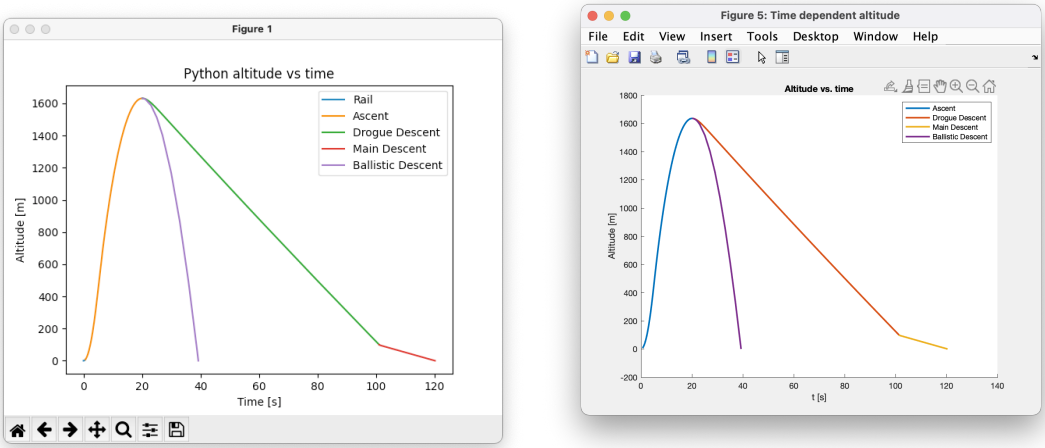


(a) LSODA, 17 pas (b) RK45, 8 pas

FIGURE 5 – Deux intégrateurs avec des nombres de pas différents.

Enfin, pour choisir entre BDF et LSODA, le choix doit être fait entre gagner respectivement 0.018% de précision (approximativement 30cm sur 1631m), ou 1 seconde de temps d'exécution. L'utilisation de LSODA a été choisie afin de gagner 1 seconde, mais il est très aisé de changer de méthode d'intégration si la précision devient importante.

Une fois l'intégrateur choisi, nous voyons dans la figure 6 le graphique obtenu avec LSODA à côté de celui produit par Matlab.



(a) LSODA, 17 pas (b) Matlab's ode45, 45 pas

FIGURE 6 – Intégrateur LSODA (Python) à gauche, Matlab à droite.

4 Difficultés rencontrées et améliorations

Le passage d'un code en Matlab en Python a amené son lot de difficultés [4] [3]. Bien qu'étant similaires, les langages utilisés requièrent des manières de penser qui leur sont propres. Notamment en ce qui concerne l'intégrateur de Matlab qui permet de joindre une fonction `OutPutFnc` qui est appelée à chaque itération, permettant de récolter des données tout au long du vol et tracer des graphiques.

Cette fonctionnalité n'étant pas disponible dans la fonction `solve_ivp` en Python, la récolte des données n'a malheureusement pas pu se faire de cette façon. Une solution pourrait être d'appeler l'intégrateur plusieurs fois avec les différentes données souhaitées en paramètre, mais cela ralentirait considérablement l'exécution.

Certains modèles sont améliorables dans le simulateur Matlab : `wind_model` peut être complété, plusieurs phénomènes aérodynamiques peuvent être pris en compte. Le simulateur Python peut intégrer ces améliorations directement.

5 Remerciements

J'aimerais remercier chaleureusement Mme Hala Khodr, pour son suivi, ses idées d'améliorations et sa présence tout au long du projet, ainsi que le Professeur Pierre Dillenbourg et son laboratoire CHILI de donner l'opportunité à des étudiant·e·s de faire partie de projets innovants et intéressants.

Je tiens aussi à remercier Antoine Scardigli, Team Leader de la team simulation de l'EPFL Rocket Team, pour son engagement au sein de l'association et du projet, sa disponibilité et pour son aide durant tout le projet.

Je remercie Sayid Derder pour son travail indispensable et de haute qualité, permettant à ce projet d'aboutir.

Enfin, je tiens à remercier tous les bénévoles de l'EPFL Rocket Team pour leur travail et la bonne ambiance qui règne dans l'association, ceci malgré les circonstances qui rendent l'échange compliqué.

Références

[1] E. Brunner and E. Mingard. Simulation avancéee de la trajectoire d'une fuséee et application à du contrôle actif. Available at `https://drive.google.com/file/d/1NmPEQHXj_WdAWp5o0HSiVY8Qrk85zD1M/view` or in the repository as 'Rapport Eric Simulation.pdf'.

[2] H. Faure, P. Germann, and J. Triomphe. Projet d'ingénieurie simultanée. Available at `https://drive.google.com/file/d/1VyvWYEkqEmsPeKv7qHpeXtRQc6_njB3q/view` or in the repository as 'ERT Report.pdf'.

[3] Matlab documentation. Available at `https://www.mathworks.com/help/matlab/`, Accédé 15.01.2021.

[4] NumPy documentation. Available at `https://numpy.org/doc/stable/reference/`, Accédé 15.01.2021.