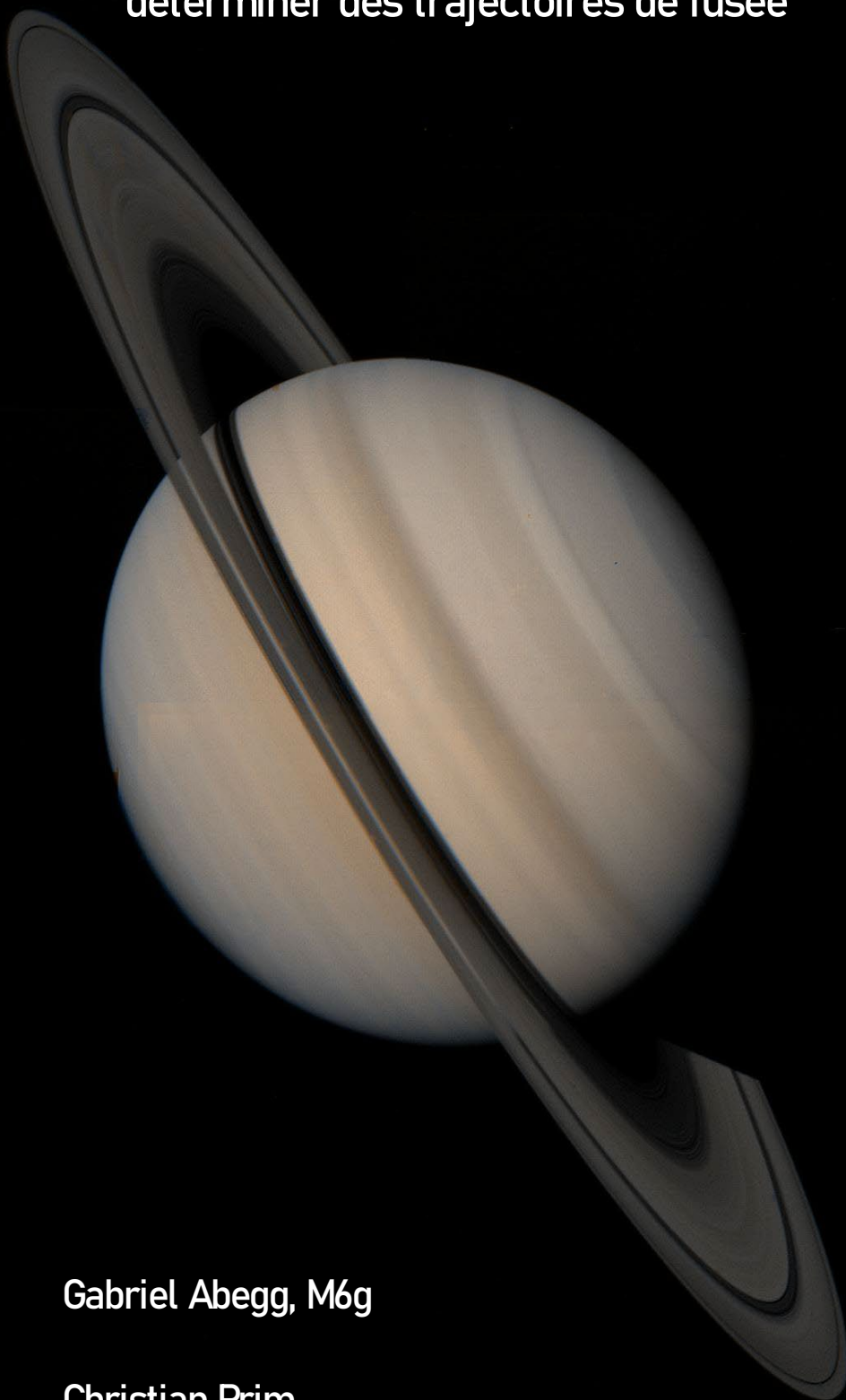


En route vers les planètes

La programmation d'un outil pour
déterminer des trajectoires de fusée



Auteur Gabriel Abegg, M6g

Mentor Christian Prim
Experte Annette Prieur

Zurich, novembre 2024

Table des matières

Preface	1
1 Introduction	1
1.1 But	1
1.2 Outils et Méthodes	1
1.3 Questions directrices	2
2 Théorie et réalisation	2
2.1 Planifier la trajectoire	2
2.1.1 Approximations	2
2.1.2 Développement du problème à deux corps	3
2.1.3 Intégration de l'équation de motion [12]	4
2.1.4 Introduction d'autres éléments importants	5
2.1.5 Le système de coordonnées perifocal [15]	6
2.2 Dédution du rayon de la sphère d'influence [18]	7
2.3 Intégration numérique et simulations	9
2.3.1 Méthode d'Euler	9
2.3.2 Méthode Runge-Kutta du quatrième ordre	9
2.3.3 Méthode de Gauss-Jackson	10
2.3.4 Summed Adams	10
2.3.5 Comparaison des méthodes d'intégration	10
2.3.6 Commentaire sur l'implémentation	11
2.4 Problème de Kepler [27]	12
2.5 Problème de Gauss [28]	13
2.6 Sélection de l'orbite de transfert	13
2.6.1 Déterminer la direction du mouvement	14
2.6.2 La notion de "Porkchop plot"	15
2.6.3 Limites physiques dans le choix d'une orbite de transfert héliocentrique[35]	15
2.7 Idées pour une approximation par tronçons de coniques complète	16
2.7.1 Les orbites dans les sphères d'influences	17
3 Résultats	20
3.1 Simulation d'orbites planétaires	20
3.2 Simulation de la fusée	20
3.3 Trajectoire simple	22
3.4 Assistance Gravitationnelle	23

4	Interprétation	24
4.1	Simulation d'orbites planétaires	24
4.2	Simulation de la fusée	24
4.3	Trajectoire simple	24
4.4	Assistance Gravitationnelle	25
5	Résumé	25
	Annexe	34
A	Le programme	34
A.1	kepler.py	34
A.2	compare.py	36
A.3	helpers1.py	41
A.4	general_definitions.py	42
A.5	horizons.py	46
A.6	planetary_movement.py	48
A.7	planetary_movement2.py	54
A.8	test2.py	59
A.9	vis.py	62
A.10	pmvisualization.py	63
A.11	simpletraj.py	65
A.12	swingby.py	75
A.13	rocket.py	91
B	Documents supplémentaires	103
B.1	Tableau des écarts	103
B.2	Choix du périapse minimal	105
B.3	Considérations pour les masses barycentriques	105

Préface

J'avais toujours une fascination indescriptible pour tout ce qui est hors de mes connaissances. Le fait de pouvoir m'immerger dans un sujet de mon choix était donc une occasion me promettant de pouvoir, non pas entièrement, mais mieux comprendre les lois de l'univers. L'astrodynamique m'a particulièrement beaucoup intéressé : c'est ce domaine scientifique qui est la base de l'exploration de l'espace.

Lors de mon travail, j'étais surpris par l'accessibilité des sujets abordés. Souvent, il y avait plusieurs livres contenant des réponses à mes questions. J'exprime ma gratitude envers la NASA qui met librement à disposition d'énormes quantités de données sur de nombreux corps célestes de notre système solaire [1]. Sans leur générosité de partager des données et connaissances avec le public, ce projet n'aurait jamais été possible. Le livre «Fundamentals of Astrodynamics» par Roger R. Bate, Donald D. Mueller, Jerry E. White et William W. Saylor m'a particulièrement bien aidé.

Une grande partie de la recherche et du programme a été réalisée à Carouge, Genève, lors de mon séjour linguistique d'un semestre. La finalisation et l'optimisation de mon programme ainsi que la rédaction ont largement eu lieu à Zurich. Quand j'ai commencé ce travail, je n'aurais jamais pu imaginer le progrès dans la compréhension de la mécanique newtonienne, ni que je travaillerais plus de 400 heures dessus. Je suis infiniment reconnaissante à mes parents de m'avoir soutenue émotionnellement lorsque le programme ne fonctionnait pas encore.

1 Introduction

1.1 But

Je souhaite créer un logiciel qui détermine une trajectoire de fusée à partir de la Terre jusqu'à une planète cible, en utilisant une approximation par tronçons de coniques. Il nous fournira une date de départ et d'arrivée, ainsi qu'une vitesse et une position permettant d'initier la trajectoire ciblée. Le $\Delta\vec{v}$ nécessaire pour établir une orbite stable autour de la planète sera ensuite calculé. Pour atteindre les planètes Saturne, Uranus et Neptune, une assistance gravitationnelle est utilisée au niveau de Jupiter.

1.2 Outils et Méthodes

Le logiciel sera entièrement programmé dans la version 3.10.12 de Python, un langage à la fois flexible et compréhensible. Toutefois, Python peut être très lent par rapport à d'autres langages de programmation. Pour contrebalancer cela, j'ai essayé d'utiliser les tableaux à n dimensions du module NumPy version 1.26.4 (ndarrays), permettant de rendre le processus plus efficace [2].

La visualisation des données a été réalisée à l'aide de Matplotlib [3], [4]. Pour pouvoir estimer le temps d'exécution d'un programme, le module tqdm [5] a été utilisé. Les tests ont été réalisés sur le « Windows Subsystem for Linux » version 2 qui utilise Ubuntu version 22.04.

1.3 Questions directrices

1. Lesquelles des méthodes qui sont utilisées dans la communauté aérospatiale pour planifier une trajectoire interplanétaire, pourrais-je utiliser ? Quelles modifications doivent être apportées pour rester dans le cadre d'un travail de maturité ?
2. Quel est l'écart de l'approximation par tronçons de coniques par rapport à la simulation à N corps ?
3. Comment réaliser une telle simulation ?
4. Comment et avec quelle précision peut-on obtenir les données nécessaires pour la simulation ? Est-ce qu'elles sont précises ?
5. Est-ce qu'il est possible d'obtenir accès à une simulation d'une agence spatiale ?
6. Quel est l'écart de ma simulation ?

2 Théorie et réalisation

2.1 Planifier la trajectoire

2.1.1 Approximations

La planification d'une trajectoire de fusée réaliste est une tâche très complexe. Même en négligeant des effets perturbateurs tels que la pression de radiation solaire, le fait que les planètes ne sont pas des sphères parfaites et des effets relativistes, nous devons toujours trouver une trajectoire de vol dans un système à N corps [6]. Une approximation simple et probablement l'une des seules réalisables dans le cadre d'un travail de maturité est l'approximation par tronçons de coniques. Cette approche consiste à définir des régions sphériques dans lesquelles uniquement l'influence d'une seule planète est prise en compte, ce qui nous permet d'utiliser la solution analytique du problème à deux corps [7]. Le rayon de cette «sphère d'influence» est dérivé dans la section 2.2. Dans les simulations permettant de calculer la trajectoire des planètes et de la fusée, toute masse de planète avec ses lunes est considérée comme concentrée sur leur barycentre commun. Seules les accélérations dues à la gravitation newtonienne sont prises en compte.

2.1.2 Développement du problème à deux corps

Équation du mouvement relatif [8]

Nous allons considérer que nous disposons de la position de deux objets dans un système de coordonnées inertiel. Les positions $\vec{r}_M$ et $\vec{r}_m$ des objets M et m sont données. $\vec{r}$ désigne un vecteur de M à m . Selon la loi universelle de la gravitation de Newton, les accélérations $\ddot{r}_m$ agissant sur m et $\ddot{r}_M$ agissant sur M sont :

$$\begin{aligned}\ddot{r}_m &= -\frac{G \cdot M}{r^3} \cdot \vec{r} \\ \ddot{r}_M &= \frac{G \cdot m}{r^3} \cdot \vec{r}\end{aligned}$$

Vu que $\ddot{r}_m - \ddot{r}_M$ est généralement équivalent à $\frac{d^2}{dt^2}(\vec{r}_m - \vec{r}_M)$, nous pouvons simplement soustraire les deux équations pour obtenir l'accélération de m relative à M :

$$\ddot{\vec{r}} = -\frac{G(M + m)}{r^3} \cdot \vec{r}$$

Pour rendre cette équation plus jolie, la constante μ nommée paramètre gravitationnel est introduite. Dans tous les cas où $M \gg m$ nous pouvons utiliser la valeur $\mu \equiv GM$. Dans des cas où la masse m n'est pas négligeable, $\mu \equiv G(M + m)$ doit être utilisé. On obtient :

$$\ddot{\vec{r}} = -\frac{\mu}{r^3} \cdot \vec{r} \quad (1)$$

Note sur les vecteurs

Des élèves qui ne sont pas encore familiarisés avec les vecteurs, comme c'était mon cas au début du travail, peuvent penser que $\dot{r} = v$ ce qui peut créer de la confusion. r désigne la magnitude du vecteur de position $\vec{r}$ tandis que v est la magnitude de la vitesse. $\dot{r}$ est donc le changement dans la magnitude de $\vec{r}$ et est égale à $\dot{r} = v \cos \gamma$ avec $\gamma = \angle(\vec{r}, \vec{v})$.

L'énergie mécanique spécifique [9]

Pour obtenir une équation scalaire, nous pouvons multiplier l'équation 1 par le vecteur $\dot{\vec{r}}$. Nous pouvons donc écrire :

$$\begin{aligned}\dot{\vec{r}} \cdot \ddot{\vec{r}} + \dot{\vec{r}} \cdot \vec{r} \frac{\mu}{r^3} &= 0 \\ v\dot{v} + \frac{\mu}{r^3} r\dot{r} &= 0\end{aligned}$$

Il est évident que $\frac{d}{dt}(\frac{v^2}{2}) = v\dot{v}$ et $\frac{d}{dt}(-\frac{\mu}{r}) = \frac{\mu}{r^2}\dot{r}$. On en arrive donc à la formule :

$$\frac{d}{dt}\left(\frac{v^2}{2} + c - \frac{\mu}{r}\right) = 0$$

Grâce à cette transformation, nous obtenons une expression qui ne varie pas avec le temps, soit une constante. Cette constante est l'énergie mécanique spécifique E . La constante d'intégration

peut être interprétée comme la référence nulle pour l'énergie potentielle. Comme il s'agit d'un choix arbitraire, nous pouvons le mettre à zéro, ce qui place notre référence nulle à l'infini. Cela équivaut à :

$$E = \frac{v^2}{2} - \frac{\mu}{r} \quad (2)$$

Conservation du moment cinétique Le moment cinétique $\vec{h} = \vec{r} \times \vec{v}$ est une grandeur physique qui va nous aider à décrire les orbites à deux corps. Le lecteur intéressé trouvera la preuve de sa conservation dans [10]. Elle décrit le sens de la rotation et le plan sur lequel l'orbite se trouve. Remarquons qu'aux apsides, l'angle entre $\vec{r}$ et $\vec{v}$ est de 90° . Par conséquent, la magnitude de $\vec{h}$ s'écrit $h = r_p v_p = r_a v_a$, où r_p et v_p représentent la distance et la vitesse au périapse, et r_a et v_a la distance et la vitesse à l'apoapse [11].

2.1.3 Intégration de l'équation de motion [12]

Nous ne pouvons pas encore intégrer directement l'équation 1. Pour faire cela, nous devons d'abord effectuer le produit vectoriel avec le moment cinétique. On trouve alors :

$$\ddot{\vec{r}} \times \vec{h} = \frac{\mu}{r^3} (\vec{h} \times \vec{r})$$

Notez que le changement de la position entre $\vec{h}$ et $\vec{r}$ entraîne la disparition du signe négatif. Puisque $\vec{h}$ est une constante, le côté gauche est équivalent à $\frac{d}{dt}(\dot{\vec{r}} \times \vec{h})$. En substituant la définition de $\vec{h}$ et en appliquant la propriété du produit triple vectoriel, on obtient :

$$\frac{\mu}{r^3} ((\vec{r} \times \vec{v}) \times \vec{r}) = \frac{\mu}{r^3} \left[\underbrace{\vec{v}(\vec{r} \cdot \vec{r})}_{r^2} - \underbrace{\vec{r} \times (\vec{r} \cdot \vec{v})}_{r \cdot \dot{r}} \right] = \mu \left(\frac{\vec{v}}{r} - \frac{\dot{r}}{r^2} \vec{r} \right)$$

Comme on peut le constater maintenant, cela est équivalent à $\frac{d}{dt} \left(\frac{\vec{r}}{r} \right)$. Cette expression peut facilement être intégrée, ce qui permet d'obtenir :

$$\dot{\vec{r}} \times \vec{h} = \mu \left(\frac{\vec{r}}{r} \right) + \vec{B}$$

$\vec{B}$ est la constante vectorielle d'intégration. Sa signification deviendra plus claire par la suite. Nous calculons le produit scalaire avec $\vec{r}$, ce qui donne :

$$\begin{aligned} \underbrace{\vec{r} \cdot \vec{v} \times \vec{h}}_{=\vec{r} \times \vec{v} \cdot \vec{h}} &= \mu \left(\frac{\vec{r} \cdot \vec{r}}{r} \right) + \vec{r} \cdot \vec{B} \\ \Rightarrow h^2 &= \mu r + r B \cos \nu \end{aligned}$$

Finalement, cela devient :

$$r = \frac{h^2/\mu}{1 + (B/\mu) \cos \nu} \quad (3)$$

En comparant ce résultat à l'équation générale d'une section conique, on remarque que la solution du problème à deux corps ne permet d'obtenir que des sections de conique. L'angle ν ci-dessous est l'angle entre le vecteur de position le plus proche du point focal, nommé périapside, et la position $\vec{r}$. Puisque l'angle entre la constante d'intégration $\vec{B}$ et $\vec{r}$ correspond à l'angle ν ci-dessous, le vecteur $\vec{B}$ doit pointer dans la direction du périapside.

$$r = \frac{p}{1 + e \cos \nu} \quad (4)$$

De cette relation résulte notamment que :

$$p = \frac{h^2}{\mu} \quad (5)$$

2.1.4 Introduction d'autres éléments importants

Grâce à des considérations géométriques, on peut montrer que pour chaque section conique sauf une parabole [13] :

$$p = a(1 - e^2) \quad (6)$$

où a est le demi-grand axe. Grâce à cette information et en raison du fait que l'anomalie vraie est nulle au périapside, l'équation 4 devient [14] :

$$r_p = \frac{p}{1 + e} = a(1 - e) \quad (7)$$

En utilisant l'équation 2 qui décrit l'énergie dans l'orbite et en utilisant le concept du moment cinétique introduit dans la section 2.1.2, nous pouvons écrire l'équation d'énergie spécifique de la manière suivante [11] :

$$E = \frac{h^2}{2r_p^2} - \frac{\mu}{r_p}$$

En prenant en considération que

$$p = \frac{h^2}{\mu} = a(1 - e^2) \Rightarrow h^2 = \mu a(1 - e^2)$$

et en utilisant l'équation 7, l'énergie spécifique mécanique devient [11] :

$$\begin{aligned} E &= \frac{\mu a(1 - e^2)}{2a^2(1 - e)^2} - \frac{\mu}{a(1 - e)} \\ &= \frac{\mu(1 - e)(1 + e)}{2a(1 - e)^2} - \frac{2\mu}{2a(1 - e)} \\ &= \frac{\mu(-1 + e)}{2a(1 - e)} \\ &= -\frac{\mu}{2a} \end{aligned} \quad (8)$$

La résolution de l'équation 6 en e conduit à [11] :

$$e = \sqrt{1 - \frac{p}{a}}$$

Grâce aux équations 5 et 8, nous pouvons remplacer a par $-\mu/2E$ et p par h^2/μ , ce qui conduit à l'équation suivante [11] :

$$e = \sqrt{1 + \frac{2Eh^2}{\mu^2}} \quad (9)$$

Cette formule sera utile dans la section 2.6.3.

2.1.5 Le système de coordonnées périfocal [15]

L'introduction des deux vecteurs d'unité $\hat{P}$ et $\hat{Q}$ permet de décrire facilement la position du satellite sous forme vectorielle. Cela devient évident en observant la figure 1. En appliquant les notions de trigonométrie, on obtient :

$$\vec{r} = r \cos \nu \hat{P} + r \sin \nu \hat{Q} \quad (10)$$

$$\text{avec } r = \frac{p}{1 + e \cos \nu} \quad (11)$$

En différenciant cette équation par rapport au temps, sachant que les deux vecteurs $\hat{P}$ et $\hat{Q}$ sont des constantes, nous obtenons :

$$\vec{v} = (\dot{r} \cos \nu - r \dot{\nu} \sin \nu) \hat{P} + (\dot{r} \sin \nu + r \dot{\nu} \cos \nu) \hat{Q}$$

Selon le livre «Fundamentals of Astrodynamics», on a $v \cos \phi = r \dot{\nu}$ où ϕ est l'angle de trajectoire de vol. Cela devient évident en observant l'image 2 : seule la composante de la vitesse perpendiculaire à la position contribue à une variation de l'anomalie vraie. De plus, il faut considérer le rapport entre la vitesse et la distance plutôt que la vitesse elle-même ; plus la distance est grande, plus la variation de l'anomalie vraie est faible. Étant donné que $v \cos \phi = \frac{h}{r}$, on peut reformuler : $h = r^2 \dot{\nu}$. De plus, nous avons vu que $p = h^2/\mu$. En différenciant l'équation 11, nous obtenons :

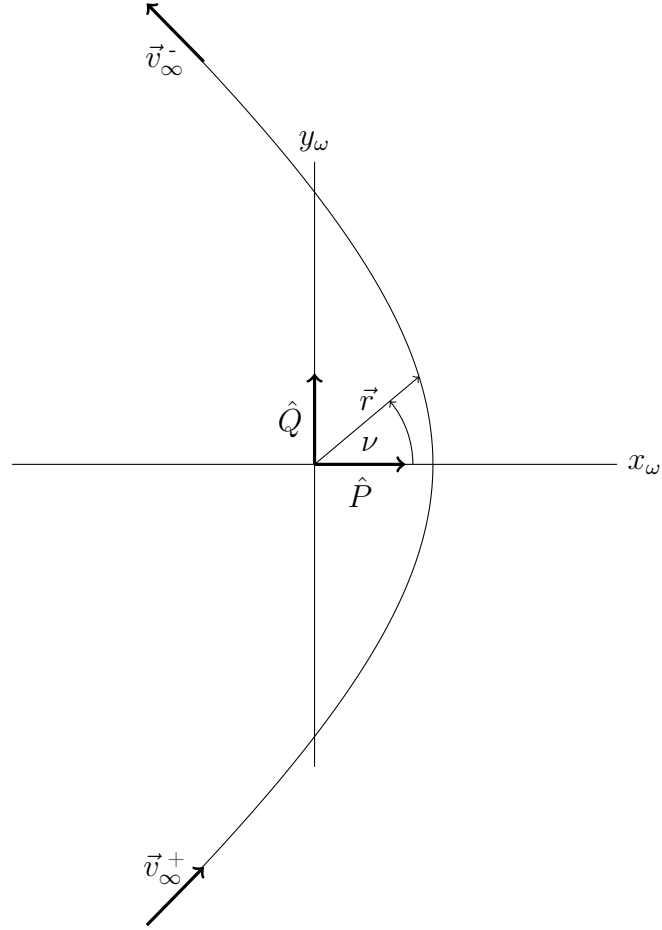


FIGURE 1 – Le système périfocal (Adapté de [16])

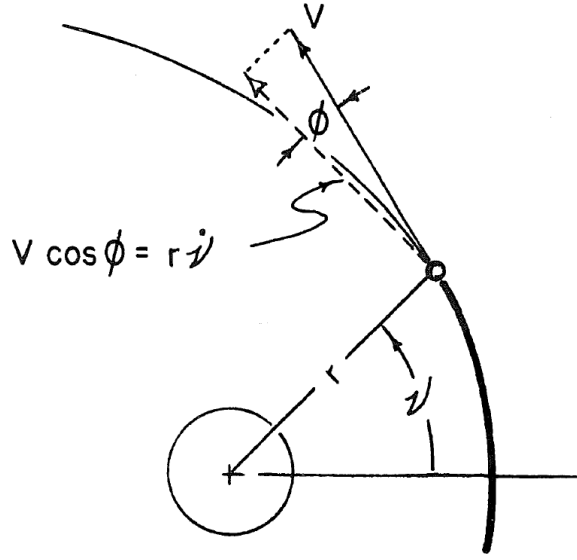


FIGURE 2 – Composante horizontale de v . Tiré de [17]

$$\begin{aligned}
 \frac{1}{r}p &= 1 + e \cos \nu \xrightarrow{\frac{d}{dt}} \\
 -\frac{1}{r^2}\dot{r}p &= -e\dot{\nu} \sin \nu \\
 \frac{1}{h}\dot{r}p &= e \sin \nu \\
 \dot{r} &= \sqrt{\frac{\mu}{p}}e \sin \nu
 \end{aligned} \tag{12}$$

et

$$\begin{aligned}
 \dot{r} &= -p(1 + e \cos \nu)^{-2}e\dot{\nu}(-\sin \nu) \\
 \dot{r} &= r(1 + e \cos \nu)^{-1}e\dot{\nu} \sin \nu \\
 \xRightarrow{\text{équation 12}} \sqrt{\frac{\mu}{p}}e \sin \nu &= r(1 + e \cos \nu)^{-1}e\dot{\nu} \sin \nu \\
 r\dot{\nu} &= \sqrt{\frac{\mu}{p}}(1 + e \cos \nu)
 \end{aligned}$$

On peut mettre cela dans l'équation précédente, ce qui équivaut à :

$$\vec{v} = \sqrt{\frac{\mu}{p}} \left[-\sin \nu \vec{P} + (e + \cos \nu) \vec{Q} \right] \tag{13}$$

2.2 Dédution du rayon de la sphère d'influence [18]

L'idée fondamentale de la sphère d'influence (SOI ou «Sphere of Influence» en anglais) a déjà été développée dans la section 2.1.1. Pour l'utiliser, nous devons connaître le rayon de cette sphère imaginaire. L'approche proposée par Laplace est la suivante. La dérivation suivante est

tirée de Vallado [18]. Nous avons deux corps célestes : A (p. ex. le Soleil) et B (p. ex. la Terre) et nous souhaitons déterminer la sphère d'influence de B . Nous commençons par décrire l'accélération du satellite relative à B . Elle vaut :

$$\ddot{\vec{r}}_{B, \text{ sat}} = \underbrace{-\frac{G(m_B + m_{\text{sat}})}{r_{B, \text{ sat}}^3} \vec{r}_{B, \text{ sat}}}_{\ddot{\vec{r}} \text{ principale}} + \underbrace{Gm_A \left(\frac{\vec{r}_{\text{sat}, A}}{r_{\text{sat}, A}^3} - \frac{\vec{r}_{B, A}}{r_{B, A}^3} \right)}_{\ddot{\vec{r}} \text{ perturbatrice}}$$

Note : Nous avons déjà vu l'accélération principale $\ddot{\vec{r}}_{\text{principale}}$ dans la section 2.1.2. Il est clair que dans le référentiel de B l'accélération perturbatrice $\ddot{\vec{r}}_{\text{perturbatrice}}$ ne peut pas seulement être celle exercée sur le satellite par A , car le référentiel B est lui-même accéléré.

De la même manière, dans le référentiel de A :

$$\ddot{\vec{r}}_{A, \text{ sat}} = \underbrace{-\frac{G(m_A + m_{\text{sat}})}{r_{A, \text{ sat}}^3} \vec{r}_{A, \text{ sat}}}_{\ddot{\vec{r}} \text{ principale}} + \underbrace{Gm_B \left(\frac{\vec{r}_{\text{sat}, B}}{r_{\text{sat}, B}^3} - \frac{\vec{r}_{A, B}}{r_{A, B}^3} \right)}_{\ddot{\vec{r}} \text{ perturbatrice}}$$

Laplace a défini la sphère d'influence comme l'endroit où le rapport entre l'accélération perturbatrice et l'accélération principale est le même dans les deux référentiels. En négligeant la masse m_{sat} et en remplaçant $\vec{r}_{B, A}$ par le vecteur $\vec{D}$, on obtient :

$$\frac{m_A \left| \frac{\vec{r}_{A, \text{ sat}}}{r_{A, \text{ sat}}^3} - \frac{\vec{D}}{D^3} \right|}{m_B \left| \frac{\vec{r}_{B, \text{ sat}}}{r_{B, \text{ sat}}^3} \right|} = \frac{m_B \left| \frac{\vec{r}_{\text{sat}, B}}{r_{\text{sat}, B}^3} - \frac{\vec{r}_{A, B}}{r_{A, B}^3} \right|}{m_A \left| \frac{\vec{r}_{\text{sat}, A}}{r_{\text{sat}, A}^3} \right|}$$

Les deux termes sont développés ci-dessous en utilisant la relation $\vec{r}_{\text{SOI}} = \vec{r}_{B, \text{ sat}} = \vec{D} - \vec{r}_{\text{sat}, B}$ et les hypothèses simplificatrices $r_{\text{SOI}} \ll D$ et $r_{\text{SOI}} \ll |\vec{r}_{A, \text{ sat}}|$ ainsi que $\vec{r}_{\text{sat}, A} \cong \vec{D}$.

$$\begin{aligned} \frac{m_A \left| \frac{\vec{r}_{A, \text{ sat}}}{r_{A, \text{ sat}}^3} - \frac{\vec{D}}{D^3} \right|}{m_B \left| \frac{\vec{r}_{B, \text{ sat}}}{r_{B, \text{ sat}}^3} \right|} &\cong \frac{m_A \left| \frac{-\vec{r}_{\text{SOI}}}{D^3} \right|}{\frac{m_B}{r_{\text{SOI}}^2}} = \frac{m_A r_{\text{SOI}}^3}{m_B D^3} \\ \frac{m_B \left| \frac{\vec{r}_{\text{sat}, B}}{r_{\text{sat}, B}^3} - \frac{\vec{r}_{A, B}}{r_{A, B}^3} \right|}{m_A \left| \frac{\vec{r}_{\text{sat}, A}}{r_{\text{sat}, A}^3} \right|} &= \frac{m_B \left| -\frac{\vec{r}_{\text{SOI}}}{r_{\text{SOI}}^3} + \frac{\vec{D}}{D^3} \right|}{m_A \left| \frac{\vec{r}_{\text{sat}, A}}{r_{\text{sat}, A}^3} \right|} = \frac{m_B \left| \frac{\vec{D} \cdot r_{\text{SOI}}^3 - \vec{r}_{\text{SOI}} \cdot D^3}{r_{\text{SOI}}^3 D^3} \right|}{m_A \left| \frac{\vec{r}_{\text{sat}, A}}{r_{\text{sat}, A}^3} \right|} = \frac{m_B \left| \frac{\vec{D} r_{\text{SOI}} (r_{\text{SOI}}^2 - D^2)}{r_{\text{SOI}}^3 D^3} \right|}{m_A \left| \frac{\vec{r}_{\text{sat}, A}}{r_{\text{sat}, A}^3} \right|} \\ &\cong \frac{m_B \frac{1}{r_{\text{SOI}}^2}}{\frac{m_A}{D^2}} = \frac{m_B D^2}{m_A r_{\text{SOI}}^2} \end{aligned}$$

Nous pouvons maintenant mettre les fractions en équation ...

$$\frac{m_B D^2}{m_A r_{\text{SOI}}^2} = \frac{m_A r_{\text{SOI}}^3}{m_B D^3}$$

... et résoudre. Le résultat est le suivant :

$$r_{\text{SOI}} = D \left(\frac{m_B}{m_A} \right)^{\frac{2}{5}} \quad (14)$$

2.3 Intégration numérique et simulations

Pour évaluer les orbites des planètes et la trajectoire de la fusée, il faut résoudre des problèmes à N corps, un problème si difficile, que la seule solution analytique trouvée est sous forme de série de Taylor, qui converge trop lentement pour être utilisée en pratique [19].

L'intégration numérique est donc la seule manière de solution convenant à ce problème. Cependant, elle n'est pas parfaite. Plus précisément, deux types de déviations d'une solution exacte existent :

La première, appelée « erreur d'arrondi » (en anglais : « roundoff error »), est due au fait que l'ordinateur ne peut effectuer des calculs qu'avec une précision finie et commet donc de petites erreurs à chaque étape de calcul. Pour contrebalancer cet effet, mon programme utilise des `np.longdouble`, ce qui constitue des valeurs à virgule flottante avec 128 bits.

La seconde, appelée « erreur de troncature », est due à l'ordre de la méthode et dépend du pas d'intégration (en anglais : step-size) [20].

2.3.1 Méthode d'Euler

Dû à sa simplicité et à sa rapidité, la méthode d'Euler est probablement la première méthode d'intégration numérique que l'on apprend. Pourtant, tout avantage de cette méthode est surcompensé par son imprécision, ce qui rend impossible l'obtention de résultats exacts avec elle. C'est cette insuffisance qui a poussé les mathématiciens à chercher des méthodes plus précises. À partir des conditions initiales, la pente est calculée. Pour de petits pas d'intégration, celle-ci est approximativement constante, ce qui permet de calculer un nouveau point de la manière suivante [21] :

$$y_{n+1} = y_n + \frac{dy}{dt} \cdot \Delta t$$

2.3.2 Méthode Runge-Kutta du quatrième ordre

La méthode de Runge-Kutta du quatrième ordre est l'une des méthodes les plus populaires pour résoudre des problèmes à valeurs initiales. L'idée derrière cette méthode est la suivante : on évalue la dérivée à plusieurs endroits dans l'intervalle à extrapoler, puis on en prend une moyenne pondérée. Pour identifier les facteurs déterminants le lieu et la façon de pondérer cette moyenne, on les déclare de manière qu'ils approximent une série de Taylor de l'ordre donné. Vous trouverez la dérivation de la méthode de Runge-Kutta du deuxième ordre dans la source [22]. Sans déduire moi-même celle du quatrième ordre, qui devrait s'obtenir avec une démarche similaire, j'utiliserai la formule présentée dans « Fundamentals of Astrodynamics ». L'idée de diviser l'équation de mouvement, qui est une équation différentielle ordinaire du second ordre, en deux équations du premier ordre pour qu'elle soit soluble, est tirée de [23].

2.3.3 Méthode de Gauss-Jackson

La méthode d'intégration Gauss-Jackson est rapide et précise. Le fait qu'il s'agisse d'une méthode de double intégration la rend idéale pour le calcul des trajectoires des satellites ou d'autres objets, dans le cadre desquels il faut toujours déduire la position à partir de l'accélération. Il s'agit d'un intégrateur en plusieurs étapes (multi-step method), ce qui signifie qu'il prend en compte plusieurs points précédents et les extrapole. L'avantage de cette caractéristique est une précision supérieure, car le changement d'accélération dans l'intervalle à intégrer est mieux représenté. Cependant, cela nécessite l'utilisation de points précédents, appelés « backpoints » ou « points de référence ». Il est donc indispensable de suivre une procédure de démarrage pour générer les premiers points. Pour cela, j'ai utilisé la méthode de Runge-Kutta présentée précédemment. Gauss-Jackson est une méthode prédictor-correcteur : d'abord un nouveau point est calculé, puis corrigé en prenant en compte la nouvelle accélération. Un point peut être corrigé plusieurs fois ou pas. Un soi-disant « mid-corrector » est utilisé dans la procédure de démarrage pour corriger les points calculés à l'aide de Runge-Kutta. Pour que l'extrapolation des accélérations fonctionne, la fonction doit être continue et régulière sur l'ensemble des points de référence. Si les accélérations agissantes changeaient spontanément, ce qui ne serait pas le cas dans notre approche simplifiée, la procédure doit être redémarrée. La dérivation présentée dans [24] utilise l'opérateur de différence arrière ∇ , mais des dérivations utilisant les opérateurs de différence avant Δ ou de différence centrée δ sont algébriquement équivalentes [24].

2.3.4 Summed Adams

La version sommée du prédictor d'Adams-Bashforth et du correcteur d'Adams-Moulton. Obtenue comme étape intermédiaire dans la dérivation de l'intégrateur Gauss-Jackson, cet intégrateur est une méthode d'intégration simple ("single-integration methods") pour résoudre des équations différentielles du premier ordre. Elle est utilisée en combinaison avec ce dernier pour obtenir la vitesse. [24]

2.3.5 Comparaison des méthodes d'intégration

Pour tester ces méthodes moi-même, j'ai créé le programme `compare.py`, qui se trouve dans l'annexe A.2. Après exactement mille orbites, le programme compare les positions $\vec{r}_1$ obtenues avec la position initiale $\vec{r}_p$. Pour mieux voir la différence entre les méthodes d'intégration, la norme du vecteur $\vec{r}_1 - \vec{r}_p$ est prise et divisée par $|\vec{r}_p|$. Vous trouverez le résultat dans le tableau 1.

Face à ce tableau, nous pouvons immédiatement exclure l'utilisation de la méthode d'intégration d'Euler, en raison de son immense imprécision. Il est plus difficile de prendre une telle décision pour les méthodes de Gauss-Jackson et Runge-Kutta : même si la méthode de Gauss-Jackson est environ vingt fois plus précise, elle est plus de quatre fois plus lente. L'analyse du

Méthode (pas d'intégration)	temps d'exécution	$ \Delta\vec{r} / \vec{r}_p $	$ \Delta\vec{v} / \vec{v}_p $
Méthode d'Euler (30 s)	10 s	3351	0.9873
Runge-Kutta 4 (30 s)	49 s	0.3601	0.2208
Gauss-Jackson/Summed Adams (30 s)	3 min 22 s	0.01519	0.01937

TABLE 1 – Comparaison des méthodes d'intégration numérique proposées

programme révèle rapidement que cela est dû à de nombreuses boucles « for » qui pourraient être remplacées par des opérations NumPy, rendant ainsi le code beaucoup plus efficace. Cela est dû au fait que la partie du programme responsable de l'intégration Gauss-Jackson a été programmée au tout début du projet, alors que je ne connaissais pas encore les pouvoirs de NumPy. Une autre explication de cette longue durée vient du fait que plusieurs « backpoints » sont utilisés, dont les accélérations doivent être recalculées à chaque itération si l'on ne veut pas les sauvegarder. De plus, le correcteur de Gauss-Jackson et Summed Adams corrige chaque point, ce qui prend également du temps.

Selon [25], il est clairement recommandé d'utiliser la méthode d'intégration numérique Gauss-Jackson pour des problèmes orbitaux, ce qui est la raison principale de l'avoir choisi. Toutefois, comme une réduction du pas d'intégration pour la méthode de Runge-Kutta conduit à un résultat excellent (ce qui n'est pas montré dans le tableau 1), la question se pose de savoir s'il valait la peine d'utiliser la méthode Gauss-Jackson, compte tenu de la complexité supplémentaire de cette méthode.

2.3.6 Commentaire sur l'implémentation

Simulation du mouvement planétaire Les implémentations de l'intégrateur de Gauss-Jackson en combinaison avec celui d'Adams sont de l'ordre 8 avec les coefficients pris de [26]. Les programmes utilisant cette méthode d'intégration sont écrits de manière à permettre un changement d'ordre qui ne consiste qu'à adapter les coefficients, ainsi que l'ordre dans le fichier `general_definitions.py`. Vu que les résultats obtenus avec l'ordre 8 sont excellentes, cela n'était pas nécessaire. Le programme qui simule les trajectoires de planètes, ainsi qu'une discussion complète des données obtenues grâce à cette simulation, se trouve dans l'annexe A.6. Les valeurs initiales de cette simulation sont celles mises à disposition par le système de Horizons de la NASA [1].

Il est intéressant de jeter un œil sur l'utilisation des `numpy.ndarrays` dans l'implémentation du programme `planetary_movement2.py`. Grâce à la fonctionnalité de « broadcasting » des tableaux de NumPy, la durée d'exécution du programme a été réduite de pres d'un facteur 40 comparée à la version non optimisée. Des valeurs plus concrètes sont obtenues en simulant 165 ans avec un pas d'intégration de 1000 s et en estimant le temps d'exécution avec le module `tqdm` [5]. Ce module indique un temps d'exécution de 35 à 39 heures pour le programme qui utilise

surtout des boucles « for », tandis que le programme optimisé finit l'exécution après environ 55 minutes. Il faut faire attention à ne pas trop diminuer le pas d'intégration, car l'optimisation du programme permet désormais de calculer des quantités de données si élevées que l'ordinateur n'a plus assez de mémoire vive pour les stocker, ce qui provoque l'arrêt du processus.

Simulation de la fusée Mes expérimentations ont montré qu'il faut un pas d'intégration d'environ 100 s pour simuler une orbite de satellite stable autour de la Terre, en tenant compte des effets perturbateurs des autres planètes. Comme les positions planétaires sont calculées avec un pas d'intégration de 1000 s, nous devons trouver un moyen efficace d'interpoler ces positions pour pouvoir calculer l'accélération agissante. J'ai d'abord pensé à utiliser la solution au problème de Kepler. Mais cette méthode présente deux défauts : le premier est que les positions des corps célestes sont données par rapport au barycentre et non pas par rapport au Soleil. Au contraire, le problème de Kepler, qui est une solution au problème à deux corps relatif, donne toujours la position par rapport au corps principal. Il est impossible de convertir l'un de ces référentiels à l'autre sans connaître le mouvement du Soleil dans l'intervalle. Même si ce mouvement est probablement négligeable, cette méthode présente un autre inconvénient : le temps d'exécution. L'interpolation est une tâche idéale pour la vectorisation avec NumPy, car les ensembles de points à interpoler sont indépendants les uns des autres. Autrement dit, il n'est pas nécessaire de calculer séparément les points 1, 11, etc., à partir des points 0, 10, etc. Chaque point de la forme $10n + 1$ peut être calculé simultanément, ce qui rend le processus très efficace. À mon avis, créer une version vectorisée de la fonction résolvant le problème de Kepler aurait été très complexe. Comme il est faisable d'écrire une telle fonction pour calculer les accélérations de nombreux points en parallèle, la méthode de Runge-Kutta s'est imposée pour effectuer cette interpolation.

Un pas d'intégration de 100 s en combinaison avec une longue durée d'intégration peut conduire à un problème d'utilisation excessive de la mémoire vive et à l'arrêt du programme.

2.4 Problème de Kepler [27]

La description d'une orbite en fonction de l'anomalie vraie ν est triviale. Même si cela est très pratique, dans la plupart des cas, nous voulons décrire une orbite en fonction du temps au lieu de l'anomalie vraie. Pour y parvenir, on pourrait, pour chaque type de tronçon de conique, établir une équation décrivant la relation entre le temps et l'anomalie excentrique, calculable à partir de l'anomalie vraie. Ensuite, on pourrait utiliser une méthode par essais et erreurs pour estimer l'anomalie vraie, en ajustant progressivement jusqu'à se rapprocher du temps donné. Cependant, selon le livre « Fundamentals of Astrodynamics », cette approche ne garantit ni des résultats précis ni une convergence fiable pour des trajectoires proches de paraboles. C'est pourquoi une autre méthode est développée, dépendant d'une nouvelle variable x , qui se laisse appliquer pour toutes les coniques. Ce changement de variable s'appelle « transformation de

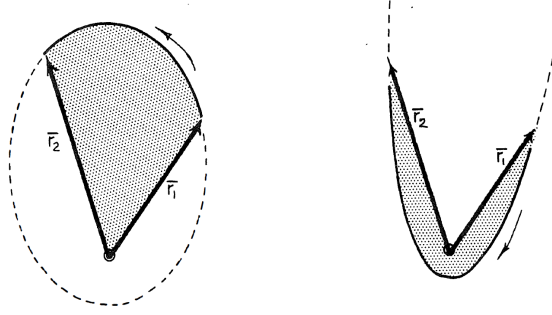


FIGURE 3 – Chemin long et court (Tiré de [29])

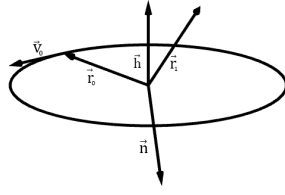
Sundman ». Compte tenu du fait que cette méthode est indépendante du reste de mon travail, j’ai choisi de ne pas développer ce sujet ici. Le lecteur intéressé trouvera la dérivation d’une telle méthode dans le chapitre 4 du livre « Fundamentals of Astrodynamics ». Le résultat est une fonction transcendante et dérivable liant la variable x et le temps, ainsi que la position et la vitesse. Une fois que x est déterminé avec la méthode de Newton-Raphson, la nouvelle position et la nouvelle vitesse peuvent être obtenues. La mise en œuvre se trouve dans l’annexe A.1. La comparaison de cette solution à deux corps avec ma simulation se trouve dans l’annexe B.1

2.5 Problème de Gauss [28]

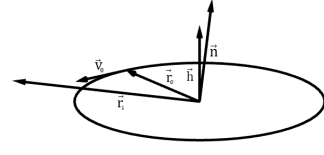
Le problème de Gauss, souvent appelé problème de Lambert, permet de déterminer une orbite lorsque deux positions, le temps de vol entre ces deux positions et la direction du mouvement sont connus. La direction du mouvement est importante, car il existe toujours deux solutions (voir l’image 3) : l’une avec une variation de l’anomalie vraie $\Delta\nu$ inférieure à π et l’autre avec une variation angulaire supérieure à π . Il existe des cas où la méthode présentée dans le livre n’arrive pas à trouver de solution (par exemple, quand les deux vecteurs sont colinéaires et le plan du transfert n’est pas défini). Dans ces cas, np.nan est renvoyé. Ces valeurs sont ignorées par le reste du programme. Il faut noter que la trajectoire courte et la trajectoire longue sont indépendantes l’une de l’autre. Pour les comparer, il faut donc toujours les calculer séparément.

2.6 Sélection de l’orbite de transfert

Au début d’une approximation par tronçons de conique, il faut sélectionner, l’orbite de transfert héliocentrique [30]. Grâce à la simulation des orbites des planètes, nous pouvons exprimer chaque position et vitesse de planète comme une fonction du temps. Si le temps de départ t_0 et le temps d’arrivée t_1 sont donnés, on connaît les valeurs $\vec{r}_0 \cong \vec{r}_{\text{planète } 0}(t_0)$, $\vec{r}_1 \cong \vec{r}_{\text{planète } 1}(t_1)$ et $\Delta t = t_1 - t_0$ aussi. Ce qui nous permet de calculer une première trajectoire seulement sous l’influence du Soleil. Cette première approximation est raisonnable, car la sphère



(a) Cas où $\angle(\vec{h}, \vec{n}) > \frac{\pi}{2}$



(b) Cas où $\angle(\vec{h}, \vec{n}) < \frac{\pi}{2}$

FIGURE 4 – Représentation schématique des deux cas possibles

d'influence est très petite par rapport au reste du transfert [30]. La seule chose qui nous manque pour utiliser la solution au problème de Gauss est donc la direction du mouvement.

2.6.1 Déterminer la direction du mouvement

Normalement, pour déterminer laquelle est énergétiquement meilleure, on devrait évaluer les deux directions de mouvement séparément [31]. Étant donné que notre cible est limitée aux planètes de notre système solaire qui orbitent approximativement toutes dans le même plan et dans le même sens, le moment cinétique $\vec{h}$ de la trajectoire à évaluer doit donc lui ressembler. Cela devient évident lorsque l'on imagine vouloir quitter la Terre : soit on veut accélérer le moins possible en utilisant la vitesse de la planète de manière optimale, soit on veut ralentir le moins possible ; dans ce dernier cas, il reste toujours une partie de la vitesse de la Terre [32].

Pour déterminer s'il faut prendre la trajectoire « courte » avec un changement dans l'anomalie vraie $\Delta\nu < \pi$ ou celle avec $\Delta\nu > \pi$ (voir la figure 3), nous calculons le moment cinétique $\vec{h}$ de la planète de départ $\vec{h}_p = \vec{r}_{\text{planète } 0} \times \vec{v}_{\text{planète } 0}$ et le vecteur $\vec{n} = \vec{r}_{\text{planète } 0} \times \vec{r}_{\text{planète } 1}$. Ce dernier est colinéaire au moment cinétique de la trajectoire « courte » entre $\vec{r}_{\text{planète } 0}$ et $\vec{r}_{\text{planète } 1}$. Par conséquent, si l'angle entre $\vec{h}$ et $\vec{n}$ est plus grand que $\frac{\pi}{2}$, comme le montre le graphique 4a, le changement dans l'anomalie vraie $\Delta\nu$ doit être plus que π ; et si l'inverse est le cas le montre le graphique 4b, le changement dans l'anomalie vraie $\Delta\nu$ doit être moins que π . Cela est implémenté dans le fichier swingby.py :

```

90 n = np.cross(r1v, r2v)
91 h = np.cross(r1v, v1v)
92 dv = unsigned_angle(r1v, r2v)
93 if unsigned_angle(n, h) > np.pi/2:
94     DM = -1
95     dv = 2*np.pi - dv
96 else:
97     DM = 1

```

Où DM signifie "Direction of Motion" ou direction de motion.

2.6.2 La notion de "Porkchop plot"

Une fois que la direction de la motion est déterminée, nous disposons de toutes les informations nécessaires pour établir de grands tableaux permettant de comparer les différentes dates de départ et d'arrivée. La visualisation d'un tel tableau s'appelle « porkchop plot ». Elle est extrêmement précieuse pour le choix des trajectoires, car elle permet de détecter directement les dates d'arrivée et de départ associées à un changement de vitesse optimale [32]. Mais les « porkchop plots » ont encore une autre utilité : en comparant les diagrammes pour deux transferts, il est possible d'identifier des trajectoires favorables pour des manœuvres d'assistance gravitationnelles [33]. Un exemple historique est l'utilisation d'environ 10 000 de ces diagrammes pour la planification des trajectoires de Voyager I et II [34]. Sans visualiser ces données, j'utiliserai ce concept pour automatiser la sélection de l'orbite.

Les deux tableaux calculés dans le programme swingby.py sont appelés pchp1 et pchp2. L'axe zéro du tableau représente les dates de départ et l'axe un des dates d'arrivée. Ils sont générés à l'aide de deux boucles « for » imbriquées, chacune parcourant une plage de dates de départ et d'arrivée. Il est évident qu'il est physiquement impossible de fixer les dates d'arrivée à un niveau inférieur à la date de départ plus un temps de transfert minimal. La limite supérieure de la première boucle peut être définie arbitrairement, tandis que celle de la seconde est donnée par le temps de transfert maximal que nous avons choisi. Cette définition permet d'éviter des tableaux absurdelement larges qui ne sont qu'à moitié remplis, mais rend plus difficile la déduction de la date à partir de l'index.

2.6.3 Limites physiques dans le choix d'une orbite de transfert héliocentrique[35]

Pour qu'un transfert entièrement balistique soit possible, deux conditions doivent être remplies concernant la planète d'assistance gravitationnelle : La première, est que l'énergie de la fusée par rapport à cette planète doit être conservée. Par conséquent, les deux excès de vitesse hyperbolique doivent avoir la même magnitude. Il est important de noter que même si l'énergie par rapport à la planète est invariable, celle par rapport au Soleil change (ce qui est la raison pour laquelle on fait des manœuvres d'assistance gravitationnelles). Elle augmente lorsque la fusée passe derrière la planète et diminue lorsque la fusée passe devant la planète (voir l'image 5). La seconde condition, est que la distance minimale à la planète doit être assez grande pour éviter tout impact. La trajectoire étant hyperbolique, nous pouvons appliquer l'équation d'énergie de la manière suivante :

$$\begin{aligned} E &= \frac{v_\infty^2}{2} = \frac{v_p^2}{2} - \frac{\mu}{r_p} = \frac{h^2}{2 r_p^2} - \frac{\mu}{r_p} \\ \Rightarrow h^2 &= r_p^2 v_\infty^2 + 2\mu r_p \end{aligned} \quad (15)$$

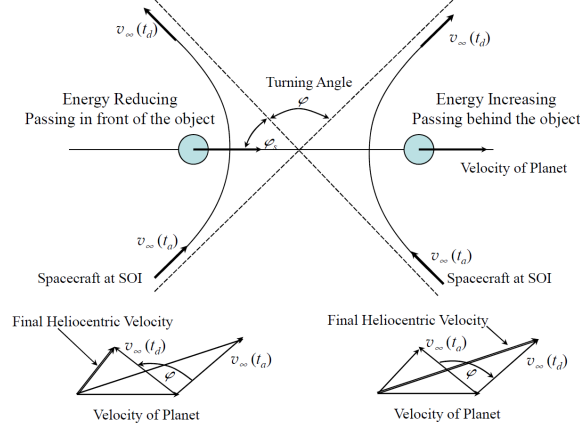


FIGURE 5 – Une visualisation de deux assistances gravitationnelles (Tiré de : [36])

Nous avons vu que l'excentricité d'une orbite à deux corps est donnée par $e = \sqrt{1 + \frac{2Eh^2}{\mu^2}}$. En insérant les expressions ci-dessus dans cette équation, on obtient :

$$e = \sqrt{1 + \frac{v_\infty^2(r_p^2 v_\infty^2 + 2\mu r_p)}{\mu^2}} = \sqrt{\left(\frac{v_\infty^2 r_p}{\mu}\right)^2 + 2\frac{v_\infty^2 r_p}{\mu} + 1} = \frac{v_\infty^2 r_p}{\mu} + 1 \quad (16)$$

Selon [35] l'excentricité est liée à l'angle de rotation supplémentaire de manière $\cos \varphi_s = \frac{1}{e}$. En examinant l'image 5, on constate que celui-ci est lié à l'angle de tournement $\varphi = \pi - 2\varphi_s$. Cela nous permet d'écrire :

$$\begin{aligned} \cos \varphi_s &= \frac{1}{e} = \frac{1}{\frac{v_\infty^2 r_p}{\mu} + 1} \\ \Rightarrow r_p &= \frac{\mu}{|\vec{v}_\infty|^2} \left(\frac{1}{\cos \varphi_s} - 1 \right) = \frac{\mu}{|\vec{v}_\infty|^2} \left(\frac{1}{\cos \frac{\pi - \varphi}{2}} - 1 \right) \end{aligned} \quad (17)$$

Si la distance de périapside de cette trajectoire d'assistance gravitationnelle est inférieure à des distances dangereuses, il ne faut pas la sélectionner. Le tableau indiquant ce choix, se trouve dans l'annexe B.2.

2.7 Idées pour une approximation par tronçons de coniques complète

Lorsque le contour de la trajectoire est déterminé, les hyperboles à l'intérieur des sphères d'influence doivent être calculées. Je n'ai pas trouvé d'informations concrètes à ce sujet dans les livres que j'ai consultés. J'ai donc essayé de développer moi-même une telle méthode à l'aide des outils acquis. L'idée fondamentale de l'algorithme complétant l'approximation par tronçons de coniques est de définir l'intérieur de chaque sphère d'influence de manière que la vitesse $\vec{v}$ pour $r \rightarrow \infty$ soit exactement l'excès de vitesse hyperbolique $\vec{v}_\infty$. Une fois que cette orbite est déterminée, nous pouvons calculer les vecteurs indiquant les points de sortie et d'entrée sur cette sphère d'influence. Grâce à ces nouveaux points de départ et au temps que la fusée passe

sur cette trajectoire à l'intérieur de la SOI, nous pouvons réévaluer la solution au problème de Gauss, comme le montre la suite, en modifiant légèrement l'excès de vitesse hyperbolique requis. Dans une nouvelle itération, cela est pris en considération, enchainant des nouveaux points sur la SOI. Étant donné que ce changement de position est très faible par rapport à la distance de la planète au soleil, les changements effectués deviennent de plus en plus petits et l'algorithme converge. Quand l'ensemble des vecteurs indiquant les positions sur la sphère d'influence bouge en totale moins qu'une valeur c , dans mon programme $c = 0.5$ m, nous pouvons afficher les résultats et terminer le programme.

2.7.1 Les orbites dans les sphères d'influences

Nous allons ensuite élaborer un concept qui nous permettra d'utiliser le système périfocal pour trouver les points de sortie et d'entrée d'une sphère d'influence. Pour cela, nous devons déterminer les constantes suivantes : le paramètre p , l'excentricité e , l'anomalie vraie à la frontière de la sphère d'influence ν_{SOI} et les deux vecteurs unitaires $\vec{P}$ et $\vec{Q}$.

Les orbites hyperboliques Le demi latus rectum p peut être déterminé en se rappelant que $p = h^2/\mu$ et en appliquant l'équation 15. On obtient :

$$p = \frac{r_p^2 v_\infty^2}{\mu} + 2r_p$$

Pour la trajectoire d'arrivée et de départ, on peut librement choisir r_p . Pour la trajectoire de manœuvre d'assistance gravitationnelle, celle-ci est donnée par l'équation 17.

Pour ν_{SOI} , nous pouvons résoudre l'équation 4 selon ν :

$$\nu_{\text{SOI}} = \arccos \frac{\frac{p}{r_{\text{SOI}}} - 1}{e} \quad (18)$$

Il faut faire attention : cette équation ne s'applique qu'aux points sortant de la sphère d'influence. Pour ceux qui entrent, $\nu_{\text{SOI}} = 2\pi - \arccos \frac{\frac{p}{r_{\text{SOI}}} - 1}{e}$.

L'orbite d'arrivée et de départ L'excentricité des orbites peut être déterminée à l'aide de l'équation 16. Mais pour ces orbites, les vecteurs $\hat{P}$ et $\hat{Q}$ ne sont pas clairement définis : il existe en effet un nombre infini de plans qui donnent la même vitesse à l'infini, comme le montre l'image 6. Nous avons donc une certaine liberté pour choisir le vecteur normal $\hat{N}$ au plan de la trajectoire, mais il doit naturellement être perpendiculaire à l'excès de vitesse hyperbolique $\vec{v}_\infty$. (voir la figure 6) Puisque le lancement d'une fusée est le plus efficace à l'équateur [38], nous voulons choisir le vecteur $\hat{N}$ de manière qu'il soit le plus proche que possible du vecteur unitaire vitesse angulaire de la Terre $\vec{\omega}_{\text{Terre}}$. Je l'ai réalisé en projetant $\vec{\omega}_{\text{Terre}}$ sur le plan normal à $\vec{v}_\infty$ et en le faisant un vecteur unitaire en divisant par sa longueur. Les mêmes considérations ont été faites pour les cas où l'on veut descendre à la surface de la planète cible, et en raison

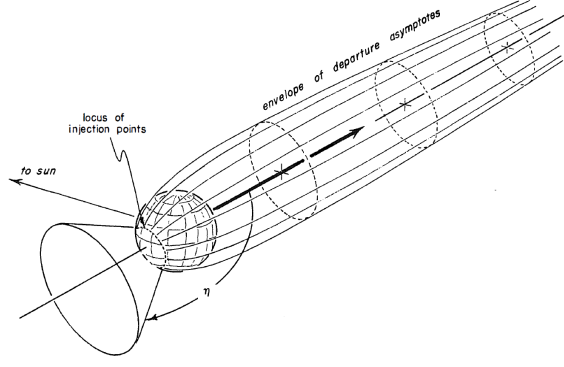


FIGURE 6 – Visualisation de l'ensemble des trajectoires de départ possibles. Tiré de [37]

de la simplicité.

La direction nord d'une planète est définie comme la direction du pôle de rotation « qui se trouve sur le côté nord du plan invariable du système solaire » [39]. La source [40], utilisée pour la Terre et [41], utilisée pour les autres planètes, donnent cette direction dans le référentiel ICRF en utilisant l'ascension droite et la déclinaison. Pour convertir cette formulation en coordonnées cartésiennes, il suffit d'appliquer les considérations trigonométriques trouvées dans [42] pour obtenir :

$$\hat{\omega} = \begin{pmatrix} \cos \delta \cos \alpha \\ \cos \delta \sin \alpha \\ \sin \delta \end{pmatrix} \quad (19)$$

Il faut faire attention : selon la définition de « nord », Vénus et Uranus tournent dans le sens négatif. Pour ces deux planètes, nous devons donc inverser la direction du vecteur obtenu.

Il faut maintenant trouver une expression pour $\hat{P}$ et $\hat{Q}$. Pour cela, je reprends l'équation 13. Comme p , e et μ sont déjà fixés, nous devons maintenant trouver ν_∞ pour pouvoir insérer $\vec{v}_\infty$ dans cette équation. Nous pouvons faire cela en prenant la limite de l'équation 18 de la manière suivante :

$$\nu_\infty = \arccos \left[\lim_{r \rightarrow \infty} \left(\frac{\frac{p}{r} - 1}{e} \right) \right] = \arccos \left(-\frac{1}{e} \right)$$

En effet, comme on peut l'imaginer en regardant l'image 1, on a $\hat{Q} = \hat{N} \times \hat{P}$. La seule variable inconnue dans l'équation décrivant le système de coordonnées perifocal à déterminer est donc $\hat{P}$.

En séparant cette équation vectorielle en un système d'équations :

$$\begin{aligned}\sqrt{\frac{p}{\mu}}v_x &= -\sin \nu P_x + (e + \cos \nu)(N_y P_z - N_z P_y) \\ \sqrt{\frac{p}{\mu}}v_y &= -\sin \nu P_y + (e + \cos \nu)(N_z P_x - N_x P_z) \\ \sqrt{\frac{p}{\mu}}v_z &= -\sin \nu P_z + (e + \cos \nu)(N_x P_y - N_y P_x)\end{aligned}$$

À l'aide de Wolfram Mathematica [43], on peut le résoudre en fonction des composantes de $\hat{P}$, ce qui conduit à :

$$P_x = \sqrt{\frac{p}{\mu}} \frac{(e^2(-v_x) \csc \nu - N_x N_y v_y \csc \nu (e + \cos \nu)^2 - N_x N_z v_z \csc \nu (e + \cos \nu)^2 + N_y^2 v_x \csc \nu (e + \cos \nu)^2 - N_y v_z (e + \cos \nu) + N_z^2 v_x \csc \nu (e + \cos \nu)^2 + N_z v_y (e + \cos \nu) - 2e v_x \cot \nu - v_x \csc \nu)}{e^2 + 2e \cos \nu + 1}$$

$$P_y = -\sqrt{\frac{p}{\mu}} \frac{(2N_x N_y v_x \csc \nu (e + \cos \nu)^2 - 2N_x v_z (e + \cos \nu) + 2N_y^2 v_y \csc \nu (e + \cos \nu)^2 + 2N_y N_z v_z \csc \nu (e + \cos \nu)^2 + 2N_z v_z (e + \cos \nu) + v_y \csc \nu - v_y \cos 2\nu \csc \nu)}{2(e^2 + 2e \cos \nu + 1)}$$

$$P_z = -\sqrt{\frac{p}{\mu}} \frac{\sin \nu (N_x N_z v_x (e \csc \nu + \cot \nu)^2 + N_x v_y \csc \nu (e + \cos \nu) + N_y N_z v_y (e \csc \nu + \cot \nu)^2 - N_y v_x \csc \nu (e + \cos \nu) + N_z^2 v_z (e \csc \nu + \cot \nu)^2 + v_z)}{e^2 + 2e \cos \nu + 1}$$

L'orbite d'assistance gravitationnelle À la différence des orbites précédentes, dans la trajectoire située dans la sphère d'influence de la planète où se déroule l'assistance gravitationnelle, les vecteurs $\hat{P}$, $\hat{Q}$ et $\hat{N}$ sont clairement déterminés par les deux excès de vitesse hyperbolique. $\hat{N}$ est défini par le produit vectoriel des deux excès de vitesse hyperbolique :

$$\vec{N} = \frac{\vec{v}_{\infty}^+ \times \vec{v}_{\infty}^-}{|\vec{v}_{\infty}^+ \times \vec{v}_{\infty}^-|}$$

En examinant l'image 1, nous constatons que les vecteurs unitaires $\vec{P}$ et $\vec{Q}$ peuvent s'écrire :

$$\begin{aligned}\vec{P} &= \frac{\vec{v}_{\infty}^+ - \vec{v}_{\infty}^-}{|\vec{v}_{\infty}^+ - \vec{v}_{\infty}^-|} \\ \vec{Q} &= \frac{\vec{v}_{\infty}^+ + \vec{v}_{\infty}^-}{|\vec{v}_{\infty}^+ + \vec{v}_{\infty}^-|}\end{aligned}$$

Mais cela peut conduire à des erreurs : même si les orbites de transfert héliocentrique ont été choisies de manière à minimiser la différence de taille entre les deux excès de vitesse hyperboliques, une déviation entre les deux valeurs persiste tout de même. Il est donc préférable de diviser les vecteurs $\vec{v}_{\infty}^+$ et $\vec{v}_{\infty}^-$ par leur taille avant d'utiliser l'équation ci-dessus.

L'excentricité e peut directement être déterminée grâce à l'angle de tournement, comme décrit dans l'équation 17.

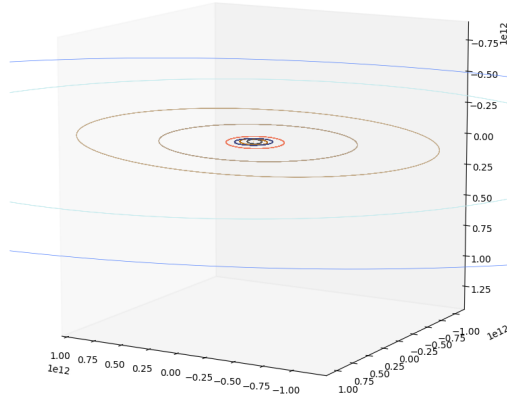


FIGURE 7 – Les orbites des planètes visualisées avec ma simulation

3 Résultats

3.1 Simulation d'orbites planétaires

Étant donné que les valeurs initiales de la simulation sont tirées du système Horizons, il est pertinent de comparer les valeurs obtenues à l'aide de ma simulation avec les valeurs de Horizons. Pour automatiser cette comparaison, le programme `test2.py` a été écrit. Ce programme compare la solution de ma simulation, la solution au problème à deux corps et la valeur de référence de Horizons. Le programme distingue entre la déviance dans la longueur des vecteurs et la distance entre les deux vecteurs de position calculés. Comme ces valeurs ne sont généralement pas intuitives pour nous, la valeur obtenue est normalisée par la distance au soleil et indiquée en pourcentage. Vous trouverez le tableau complet dans l'annexe B.1. En résumé, les écarts de ma simulation par rapport à Horizons se trouvent tous sous 1 % de la distance au soleil. La figure 7 montre une visualisation des orbites calculées.

3.2 Simulation de la fusée

Le test de la simulation avec différents pas d'intégration a montré qu'il est possible d'obtenir une orbite terrestre stable avec un pas d'intégration de 100 s. L'image 8 montre une découpe d'une visualisation d'une telle orbite.

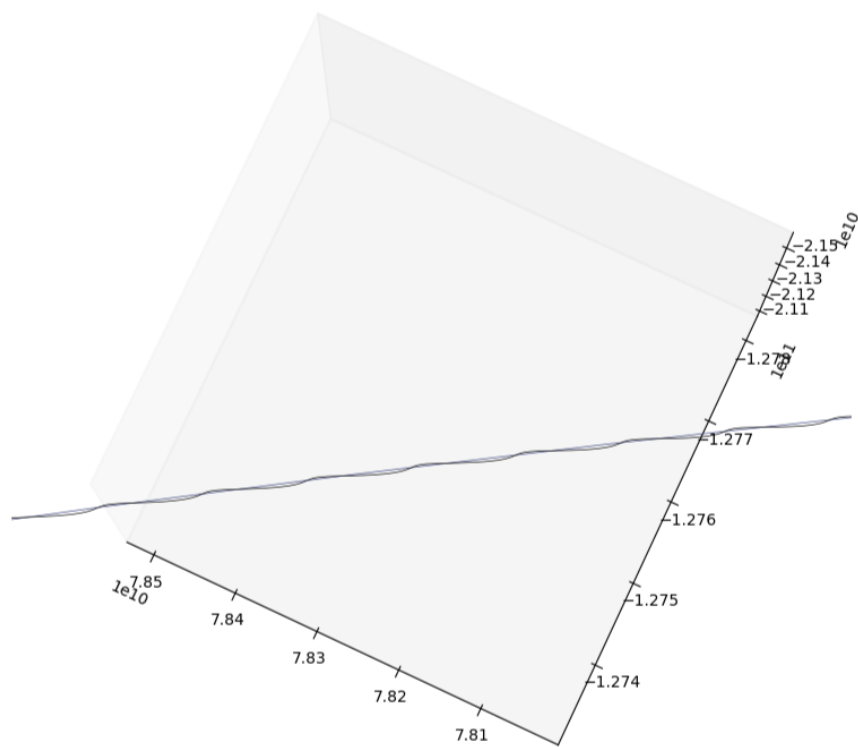


FIGURE 8 – Orbite stable autour de la Terre

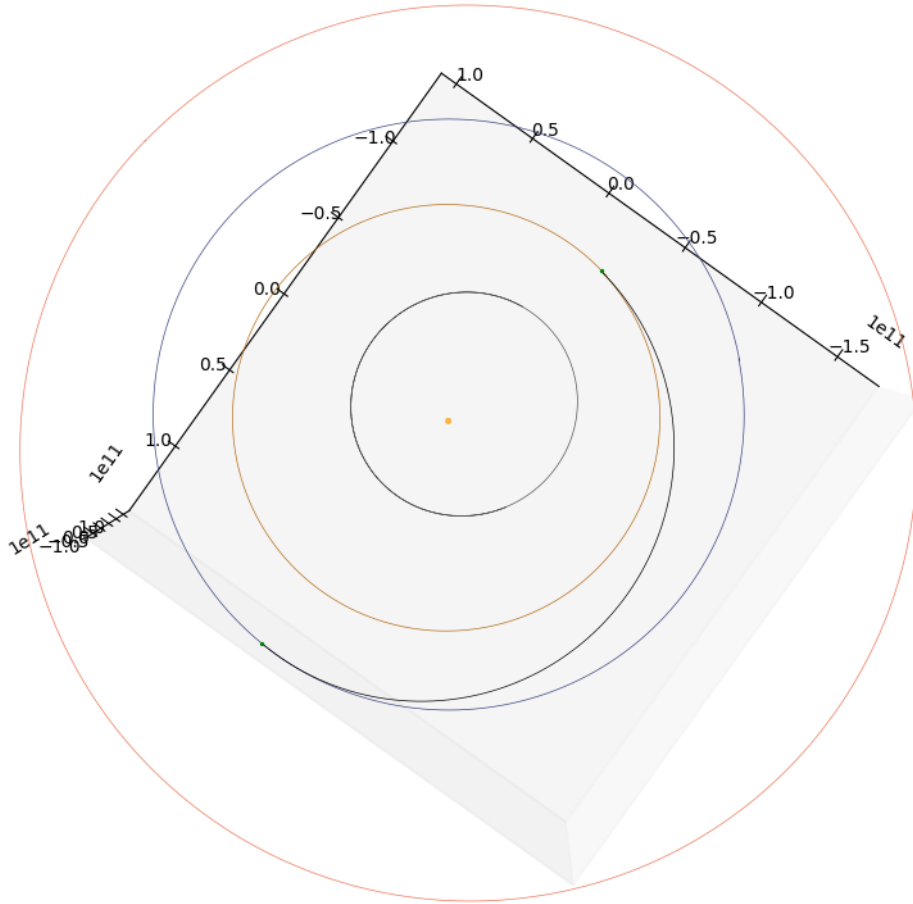


FIGURE 9 – Trajectoire vers Vénus

3.3 Trajectoire simple

Je vais présenter l'exemple d'une trajectoire à Vénus, mais il est possible de choisir n'importe quelle planète jusqu'à Jupiter. L'image 9 représente une simulation utilisant la solution indiquée par le programme `simpletraj.py`. Ce dernier indique également le vecteur d'arrivée dans le référentiel de la planète ciblée. Après la fin de la simulation, ce vecteur d'arrivée, ajouté à la position de la planète, est comparé aux points calculés. La magnitude du point le plus proche du point d'arrivée ciblé est fournie. Pour cette trajectoire, elle est de 171 395 km. Pour une orbite d'attente à une altitude de 185 km le Δv nécessaire pour aller dans l'orbite de transfert héliocentrique est de 3.598 km/s. La fusée quittera la Terre en décembre 2040 de la terre et arrivera à Vénus en mai 2041, après avoir passé 142 jours sur cette trajectoire.

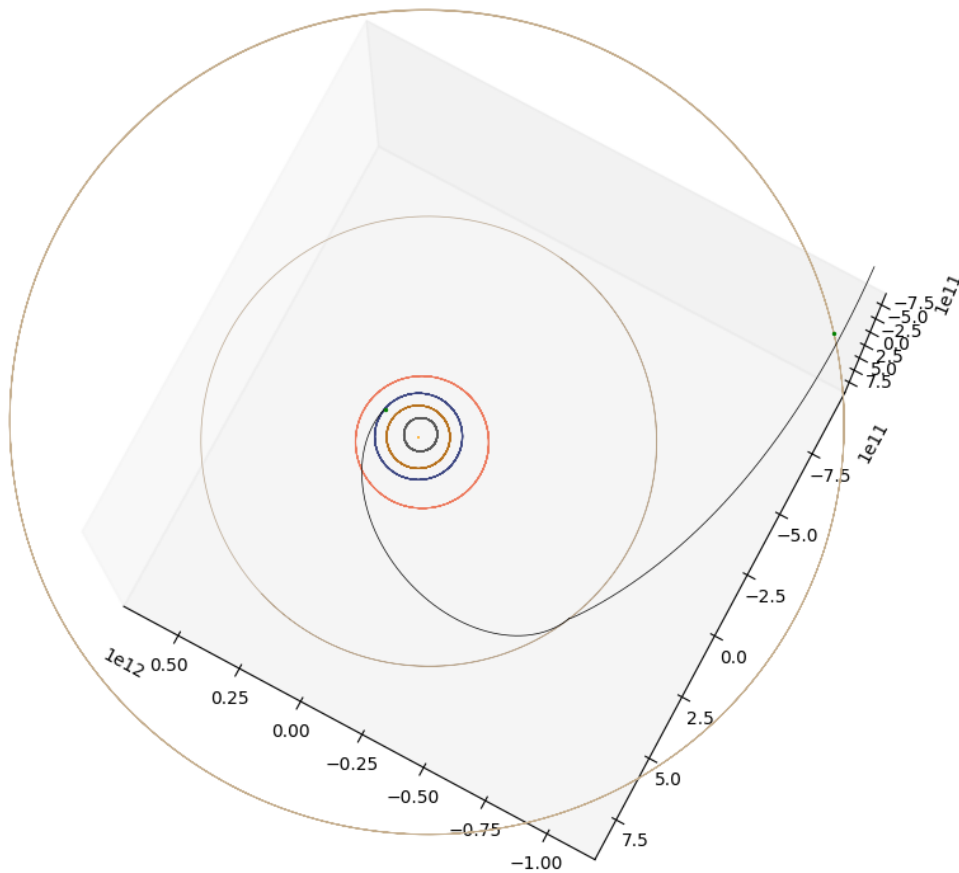


FIGURE 10 – Trajectoire d’assistance gravitationnelle vers Saturne

3.4 Assistance Gravitationnelle

J’ai testé le programme qui détermine des trajectoires d’assistance gravitationnelle, à l’instar de Saturn. Un Δv de 7.434 km/s est nécessaire pour sortir de l’orbite d’attente à une altitude de 185 km. Après la convergence de l’algorithme, la différence dans la magnitude de l’excès de vitesse hyperbolique à la planète d’assistance gravitationnelle est passée de moins de 1 m/s quand le temps d’arrivée et de départ à Jupiter étaient considérés comme identique à 294.5 m/s. La distance la plus proche au périapse calculée est de 194 432 085 km, et la plus proche au centre de la planète est de 194 154 748 km. Le temps passé sur la trajectoire calculée est de 6.5 ans. L’image 10 est une représentation graphique de la simulation de l’orbite décrite par le programme swingby.py.

4 Interprétation

4.1 Simulation d'orbites planétaires

Le tableau se trouvant dans l'annexe B.1 démontre une proximité avec les valeurs de référence. Je peux donc affirmer avec confiance que les valeurs obtenues sont fiables. Les petites déviations qui existent peuvent être expliquées par des erreurs d'arrondi de l'ordinateur ou par le fait que la méthode de Gauss-Jackson est seulement du 8ième ordre, ce qui contribue à l'erreur de troncature [20]. Évidemment, le système de la NASA prend en considération la relativité générale pour ses simulations [44], mon approximation newtonienne pourrait donc être une autre source d'erreur.

4.2 Simulation de la fusée

Selon la littérature consultée [45], une augmentation de l'ordre de 8 à 12 peut améliorer la précision jusqu'à un facteur dix. Un tel changement a été effectué. Selon la même source, une diminution du pas d'intégration à 25 s ou 40 s (il faut obligatoirement choisir un diviseur de 1000 s pour que l'interpolation fonctionne) est encore plus importante pour la précision qu'un changement d'ordre. Cela conduit toutefois non seulement à des temps d'exécution beaucoup plus longs, mais aussi à une utilisation importante de la mémoire vive limitée, ce qui est la raison pour laquelle j'ai renoncé à cette solution. Sur la base de [46], je suis de l'avis qu'un pas d'intégration de 100 s suffit largement pour la simulation des trajectoires interplanétaires : la raison pour laquelle le pas d'intégration devrait être diminué est que la direction de l'accélération change très rapidement dans une orbite planétaire. La simulation doit donc prendre davantage de points en considération pour représenter correctement ce phénomène. Toutefois, étant donné que la fusée passe la plupart du temps presque uniquement sous l'influence du Soleil, dont la distance est très grande, et que la simulation des planètes était très réussie avec un pas d'intégration de 1000 s, les imprécisions résultant de la courte durée d'influence d'une planète devraient être tolérables.

4.3 Trajectoire simple

Je ne peux pas faire ici de comparaison directe, car cette trajectoire ne représente qu'une des nombreuses possibilités d'atteindre Vénus. Cependant, en examinant le graphique 9, nous pouvons constater que cette trajectoire est relativement proche d'un transfert de Hohmann, un concept souvent discuté dans la littérature. En utilisant les données et la procédure décrites dans [47], on peut montrer qu'un Δv de 3.52 km/s est nécessaire pour initier le transfert de Hohmann, si l'on part d'une orbite d'attente de rayon 6563.137 km (ce qui correspond à une altitude de 185 km). Cela correspond à la valeur obtenue. Un transfert de Hohmann, comme

décrit dans [47], prend environ quatre jours de plus que la trajectoire trouvée grâce à l'aide de mon programme. Selon une comparaison de [48] entre une approximation par tronçons de coniques et un transfert à n corps modélisant une trajectoire jusqu'à Mars, une déviation de 200 000 km est possible et normalement pas significative. Avec une déviation de 171 395 km nous sommes en deçà de cette frontière, mais évidemment une comparaison directe avec un transfert jusqu'à Mars n'est possible que qualitativement.

4.4 Assistance Gravitationnelle

Je vais encore une fois comparer la trajectoire calculée à un transfert de Hohmann. La vitesse nécessaire pour initier un transfert de Hohmann est de 7.273 km/s [47]. Nous sommes donc très proches, voire un peu moins bons que le transfert de Hohmann. Cela peut étonner, car la manœuvre d'assistance gravitationnelle devrait être énergétiquement beaucoup mieux. Cela est probablement dû au fait que la trajectoire doit aussi prendre en considération la position de Jupiter. Je suppose qu'il n'y a pas assez d'alignements entre les trois planètes dans la plage de recherche pour des départs de la Terre pour choisir un transfert entre la Terre et Jupiter optimal.

Comme nous l'avons constaté précédemment, nous ratons la planète d'une distance assez importante. La comparaison de la distance minimale à la planète, $r_p = 1.941547 \cdot 10^{11}$ m, avec le rayon de la sphère d'influence à l'arrivée, $r_{\text{SOI}} = 5.2597 \cdot 10^{10}$ m, souligne assez bien ce fait. Je pense que cela est principalement dû au fait que les deux excès de vitesse hyperbolique ne sont plus de la même taille après avoir pris en considération le mouvement de Jupiter pendant que la fusée est dans son influence. Cela expliquerait aussi pourquoi on voit sur l'image 10 que la simulation ne se termine pas à la hauteur de Saturne : en allant plus vite que prévu, la fusée dépasse la planète Saturne avant la fin de la simulation. En théorie, si l'on connaissait la durée moyenne pendant laquelle la fusée se trouve dans la sphère d'influence de Jupiter, on pourrait tenir compte de ce facteur pour choisir la trajectoire, afin de minimiser l'effet de la différence de taille des excès de vitesse hyperboliques par rapport à l'algorithme qui détermine l'intérieur des sphères d'influence. Il aurait été intéressant de tester les effets d'un tel changement.

5 Résumé

Différentes méthodes de calcul d'orbites ont été comparées. Parmi celles-ci figuraient les méthodes d'intégration numérique d'Euler, de Runge-Kutta d'ordre quatre et l'intégrateur de Gauss-Jackson. Ce dernier a été utilisé pour simuler les orbites des planètes pour les 165 prochaines années, et ce, avec d'excellents résultats. La comparaison de ces résultats avec la solution du problème de Kepler a été effectuée.

En utilisant la solution du problème de Gauss, souvent appelé problème de Lambert, il est

possible de calculer des tableaux avec l'excès de vitesse hyperbolique. Ils peuvent être utilisés pour trouver une date de départ et d'arrivée pour une trajectoire simple ou pour indiquer une trajectoire utilisant l'assistance gravitationnelle. En utilisant ces dates et le système de coordonnées perifocal, un algorithme permettant d'obtenir une approximation par tronçons de coniques complète a été inventé. Le résultat peut être comparé à une simulation à N corps. Dans ce travail, les masses de toutes les planètes ainsi que leurs satellites naturels sont considérées comme concentrées en un seul point. Toute accélération perturbatrice non liée à la gravitation newtonienne a été négligée.

Table des figures

	Page de couverture : Image de Saturne fait par Voyager I. [49]	i
1	Le système périfocal (Adapté de [16])	6
2	Composante horizontale de v . Tiré de [17]	7
3	Chemin long et court (Tiré de [29])	13
4	Représentation schématique des deux cas possibles	14
5	Une visualisation de deux assistances gravitationnelles (Tiré de : [36])	16
6	Visualisation de l'ensemble des trajectoires de départ possibles. Tiré de [37]	18
7	Les orbites des planètes visualisées avec ma simulation	20
8	Orbite stable autour de la Terre	21
9	Trajectoire vers Vénus	22
10	Trajectoire d'assistance gravitationnelle vers Saturne	23

Liste des tableaux

1	Comparaison des méthodes d'intégration numérique proposées	11
2	L'écart de la position après 165 ans	104
3	L'écart de la vitesse après 165 ans	104
4	Données atmosphériques	105
5	Périapse minimal des planètes	105
6	La masse de Jupiter avec ses lunes galiléennes	106

Références

- [1] J. D. GIORGINI et J. S. S. D. GROUP. « NASA/JPL Horizons On-Line Ephemeris System. » (2022), adresse : <https://ssd.jpl.nasa.gov/horizons/>.
- [2] « NumPy, Powerful N-dimensional arrays. » (2024), adresse : <https://numpy.org/> (visité le 17/11/2024).
- [3] J. D. HUNTER, « Matplotlib : A 2D graphics environment, » *Computing in Science & Engineering*, t. 9, n° 3, p. 90-95, 2007. DOI : [10.1109/MCSE.2007.55](https://doi.org/10.1109/MCSE.2007.55).
- [4] T. M. D. TEAM, *Matplotlib : Visualization with Python*, version v3.8.4, avr. 2024. DOI : [10.5281/zenodo.10916799](https://doi.org/10.5281/zenodo.10916799). adresse : <https://doi.org/10.5281/zenodo.10916799>.
- [5] C. da COSTA-LUIS, *tqdm : A fast, Extensible Progress Bar for Python and CLI*, version v4.66.4, mai 2024. DOI : [10.5281/zenodo.11107065](https://doi.org/10.5281/zenodo.11107065). adresse : <https://doi.org/10.5281/zenodo.11107065>.
- [6] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, p. 891-921, ISBN : 9780486497044.
- [7] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, chap. 7.4, ISBN : 9780486497044.
- [8] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, chap. 1.3.2, ISBN : 9780486497044.
- [9] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, chap. 1.4.1, ISBN : 9780486497044.
- [10] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, p. 14-15, ISBN : 9780486497044.
- [11] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, chap. 1.6, ISBN : 9780486497044.
- [12] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, p. 16-17, ISBN : 9780486497044.

- [13] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, chap. 1.5.3, ISBN : 9780486497044.
- [14] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, p. 21, ISBN : 9780486497044.
- [15] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, chap. 2.5.1, ISBN : 9780486497044.
- [16] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, chap. 2.2.4, p. 46, ISBN : 9780486497044.
- [17] R. BATE, D. MUELLER et J. WHITE, *Fundamentals of Astrodynamics*. Dover Publications, 1971, chap. 1.8, p. 32, ISBN : 0-486-60061-0.
- [18] D. VALLADO, *Fundamentals of Astrodynamics and Applications : Fourth Edition* (Space Technology Library). Microcosm Press, 2013, chap. 12.2.1, p. 945-948, ISBN : 978-1881883180.
- [19] « n-body problem. » (2024), adresse : https://en.wikipedia.org/wiki/Three-body_problem#n-body_problem (visité le 17/11/2024).
- [20] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, chap. 9.5.2, ISBN : 9780486497044.
- [21] W. DURANDI, B. D. WONG, M. KRIENER, H. KÜNSCH, A. VOGELSANGER et J. WALDVOGEL, « Mathematik, » in *Formeln, Tabellen, Begriffe*, 2019, chap. 7.8, p. 142, ISBN : 978-3-280-04193-2.
- [22] M. ZELTKEVIC. « Runge-Kutta Methods. » (15.04.1998), adresse : https://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node5.html.
- [23] I. C. LONDON. « Coupled ODEs integration. » (2022), adresse : https://primer-computational-mathematics.github.io/book/c_mathematics/numerical_methods/5_Runge_Kutta_method.html#coupled-odes-integration.
- [24] M. M. BERRY et L. M. HEALY, « Implementation of Gauss-Jackson Integration for Orbit Propagation, » *The Journal of the Astronautical Sciences*, t. 52, n° 3, p. 331-357, 2004. DOI : <https://doi.org/10.1007/BF03546367>.
- [25] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, chap. 9.6.5, p. 352, ISBN : 9780486497044.

- [26] M. M. BERRY et L. M. HEALY. « Implementation of Gauss-Jackson Integration for Orbit Propagation. » (2004), adresse : <http://hdl.handle.net/1903/2202>.
- [27] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, chap. 4, ISBN : 9780486497044.
- [28] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, chap. 5, p. 191-194, ISBN : 9780486497044.
- [29] R. BATE, D. MUELLER et J. WHITE, *Fundamentals of Astrodynamics*. Dover Publications, 1971, chap. 5.2, p. 229, ISBN : 0-486-60061-0.
- [30] D. VALLADO, *Fundamentals of Astrodynamics and Applications : Fourth Edition* (Space Technology Library). Microcosm Press, 2013, chap. 12.2.2, p. 948, ISBN : 978-1881883180.
- [31] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, chap. 5.3, p. 201, ISBN : 9780486497044.
- [32] D. VALLADO, *Fundamentals of Astrodynamics and Applications : Fourth Edition* (Space Technology Library). Microcosm Press, 2013, chap. 12.3, p. 955, ISBN : 978-1881883180.
- [33] D. VALLADO, *Fundamentals of Astrodynamics and Applications : Fourth Edition* (Space Technology Library). Microcosm Press, 2013, chap. 12.4, p. 957-958, ISBN : 978-1881883180.
- [34] P. J. WESTWICK, *Into the Black : JPL and the American Space Program, 1976-2004*. Yale University Press, 2008, chap. 2, p. 21, ISBN : 978-0-300-11075-3.
- [35] D. VALLADO, *Fundamentals of Astrodynamics and Applications : Fourth Edition* (Space Technology Library). Microcosm Press, 2013, chap. 12.4, p. 957-960, ISBN : 978-1881883180.
- [36] D. VALLADO, *Fundamentals of Astrodynamics and Applications : Fourth Edition* (Space Technology Library). Microcosm Press, 2013, chap. 12.4, p. 959, ISBN : 978-1881883180.
- [37] R. BATE, D. MUELLER et J. WHITE, *Fundamentals of Astrodynamics*. Dover Publications, 1971, chap. 8.3, p. 372, ISBN : 0-486-60061-0.
- [38] U. WALTER, *Astronautics : The Physics of Space Flight : Third Edition*. Springer International Publishing, 2019, chap. 6.3, p. 145, ISBN : 978-3-319-74372-1. DOI : <https://doi.org/10.1007/978-3-319-74373-8>.
- [39] B. A. ARCHINAL, M. A'HEARN et E. e. a. BOWELL, « Report of the IAU Working Group on Cartographic Coordinates and Rotational Elements : 2009, » *Celestial Mechanics and Dynamical Astronomy*, t. 109, p. 105, 2011. DOI : <https://doi.org/10.1007/s10569-010-9320-4>.

- [40] B. A. ARCHINAL, M. A'HEARN et E. e. a. BOWELL, « Report of the IAU Working Group on Cartographic Coordinates and Rotational Elements : 2009, » *Celestial Mechanics and Dynamical Astronomy*, t. 109, p. 101-135, 2011. DOI : <https://doi.org/10.1007/s10569-010-9320-4>.
- [41] B. ARCHINAL, C. ACTON et M. e. a. A'HEARN, « Report of the IAU Working Group on Cartographic Coordinates and Rotational Elements : 2015, » *Celestial Mechanics and Dynamical Astronomy*, t. 130, p. 22-130, 2018. DOI : <https://doi.org/10.1007/s10569-017-9805-5>.
- [42] H. ROTH, « Astronomie, » in Formeln, Tabellen, Begriffe, 2019, chap. 1, p. 207, ISBN : 978-3-280-04193-2.
- [43] I. WOLFRAM RESEARCH, *Mathematica*, version 14.1, Utilisé pour résoudre le système d'équations avec la fonction "Solve[]", 2024. adresse : <https://www.wolfram.com/mathematica>.
- [44] J. D. GIORGINI et J. S. S. D. GROUP. « NASA/JPL Horizons On-Line Ephemeris System. » (2022), adresse : https://ssd.jpl.nasa.gov/orbits_doc.html.
- [45] M. M. BERRY et L. M. HEALY, « Implementation of Gauss-Jackson Integration for Orbit Propagation, » *The Journal of the Astronautical Sciences*, t. 52, n° 3, p. 353-356, 2004. DOI : <https://doi.org/10.1007/BF03546367>.
- [46] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, chap. 9.3, p. 332, ISBN : 9780486497044.
- [47] R. BATE, D. MUELLER, J. WHITE et W. SAYLOR, *Fundamentals of Astrodynamics : Second Edition* (Dover Books on Physics). Dover Publications, 2020, chap. 8.3, p. 303-308, ISBN : 9780486497044.
- [48] D. VALLADO, *Fundamentals of Astrodynamics and Applications : Fourth Edition* (Space Technology Library). Microcosm Press, 2013, chap. 12.2.5, p. 954, ISBN : 978-1881883180.
- [49] NASA/JPL-CALTECH. « Saturn and Moons Tethys, Enceladus, and Mimas. » (oct. 2024), adresse : <https://science.nasa.gov/image-detail/pia01383-3/>.
- [50] A. GLEAVES, R. ALLEN, A. TUPIS et al., « A Survey of Mission Opportunities to Trans-Neptunian Objects - Part II, Orbital Capture, » in *AIAA/AAS Astrodynamics Specialist Conference*, Minneapolis, Minnesota, août 2012, p. 4. DOI : <https://doi.org/10.2514/6.2012-5066>.
- [51] D. R. WILLIAMS. « Jupiter Fact Sheet. » (2024), adresse : <https://nssdc.gsfc.nasa.gov/planetary/factsheet/jupiterfact.html>.

- [52] D. R. WILLIAMS. « Saturnian Rings Fact Sheet. » (2022), adresse : <https://nssdc.gsfc.nasa.gov/planetary/factsheet/satringfact.html>.
- [53] D. R. WILLIAMS. « Mercury Fact Sheet. » (2024), adresse : <https://nssdc.gsfc.nasa.gov/planetary/factsheet/mercuryfact.html>.
- [54] U. WALTER, *Astronautics : The Physics of Space Flight : Third Edition*. Springer International Publishing, 2019, chap. 10.1.2, p. 431, ISBN : 978-3-319-74372-1. DOI : <https://doi.org/10.1007/978-3-319-74373-8>.
- [55] U. WALTER, *Astronautics : The Physics of Space Flight : Third Edition*. Springer International Publishing, 2019, chap. 6.1.3, p. 129, ISBN : 978-3-319-74372-1. DOI : <https://doi.org/10.1007/978-3-319-74373-8>.
- [56] U. WALTER, *Astronautics : The Physics of Space Flight : Third Edition*. Springer International Publishing, 2019, chap. 6.1.2, p. 125, ISBN : 978-3-319-74372-1. DOI : <https://doi.org/10.1007/978-3-319-74373-8>.
- [57] D. R. WILLIAMS. « Venus Fact Sheet. » (2024), adresse : <https://nssdc.gsfc.nasa.gov/planetary/factsheet/venusfact.html>.
- [58] D. R. WILLIAMS. « Mars Fact Sheet. » (2024), adresse : <https://nssdc.gsfc.nasa.gov/planetary/factsheet/marsfact.html>.
- [59] D. R. WILLIAMS. « Uranus Fact Sheet. » (2024), adresse : <https://nssdc.gsfc.nasa.gov/planetary/factsheet/uranusfact.html>.
- [60] D. R. WILLIAMS. « Neptune Fact Sheet. » (2024), adresse : <https://nssdc.gsfc.nasa.gov/planetary/factsheet/neptunefact.html>.
- [61] J. D. GIORGINI et J. S. S. D. GROUP. « NASA/JPL Horizons On-Line Ephemeris System. » (2022), adresse : <https://ssd.jpl.nasa.gov/horizons/>.
- [62] D. R. WILLIAMS. « Jupiter Fact Sheet. » (2024), adresse : <https://nssdc.gsfc.nasa.gov/planetary/factsheet/jupiterfact.html>.
- [63] D. R. WILLIAMS. « Jovian Satellite Fact Sheet. » (2023), adresse : <https://nssdc.gsfc.nasa.gov/planetary/factsheet/joviansatfact.html>.
- [64] D. R. WILLIAMS. « Saturnian Satellite Fact Sheet. » (2023), adresse : <https://nssdc.gsfc.nasa.gov/planetary/factsheet/saturniansatfact.html>.
- [65] D. R. WILLIAMS. « Uranian Satellite Fact Sheet. » (2023), adresse : <https://nssdc.gsfc.nasa.gov/planetary/factsheet/uraniansatfact.html>.
- [66] D. R. WILLIAMS. « Neptunian Satellite Fact Sheet. » (2024), adresse : <https://nssdc.gsfc.nasa.gov/planetary/factsheet/neptuniansatfact.html>.

Ce texte a été révisé linguistiquement à l'aide de l'intelligence artificielle. Le contenu n'a pas été modifié. Les outils utilisés sont DeepL Write, Chat GPT et Google Gemini. La plupart des termes techniques ont été traduits à l'aide du dictionnaire en ligne Tureng.

Une version numérique de ce document ainsi que le programme sont disponibles à l'adresse suivante : <https://github.com/gabrielleonabegg/En-route-vers-les-Planetes>

Annexe

A Le programme

A.1 kepler.py

```
1 from math import sqrt, factorial, pi
2 import numpy as np
3
4 def vect(a,b,c):
5     return np.array([a,b,c], dtype=np.longdouble)
6
7 def mag(a:np.ndarray):
8     return np.sqrt(a.dot(a)) #https://stackoverflow.com/questions
   /9171158/how-do-you-get-the-magnitude-of-a-vector-in-numpy
9
10 def C(z):
11     s = 0
12     a = 0.5
13     k = 1
14     while abs(a) > 0.000000000001:
15         s += a
16         a = (-z)**k / np.longdouble(factorial(2*k+2))
17         k +=1
18     return s
19
20
21 def S(z):
22     s = 0
23     a = 1/6
24     k = 1
25     while abs(a) > 0.000000000001:
26         s += a
27         a = (-z)**k / np.longdouble(factorial(2*k+3))
28         k +=1
29     return s
```

```

30
31 def kepler(r0v: np.ndarray, v0v: np.ndarray, t, mu, xn = None):
32     if t == 0: #enke's method might make use of this
33         return r0v, v0v
34     r0 = mag(r0v)
35     v0 = mag(v0v)
36     E = 0.5*(v0)**2 - mu/r0
37     a = - mu/(2*E)
38     h = np.cross(r0v, v0v)
39     alpha = (2*mu/r0 - v0**2)/mu #a^-1 to make sure we don't divide by 0, as
        suggested on page 169, eq (4-74)
40     e = sqrt(1+(2*E*mag(h)**2)/(mu**2))
41     if 0 < e < 1 and mu != 0:
42         TP = 2*np.pi*np.sqrt(a**3/mu)
43         while t > TP:
44             t = t - TP
45         while t < 0:
46             t += TP
47 #solve for x when time is known: section 4.4.2
48     if xn == None:
49         #select starting value
50         if 0 < e < 1:
51             xn = sqrt(mu)*t*alpha
52         else:
53             xn = np.sign(t)*np.sqrt(-a)*np.log((-2*mu*t)/(a*(r0v.dot(v0v) + np.
                sign(t)*np.sqrt(-mu*a)*(1-r0/a))))
54             tn = t+20
55 #determining x from initial conditions and time (Newton iteration scheme)
56     while abs((t-tn)/t) > 1e-8: #in our elliptic example, it doesn't even get
        executed once!
57         #print("Goal: {}".format(t))
58         z = xn**2 *alpha
59         tn = np.dot(r0v, v0v)/mu * xn**2 * C(z) + (1-r0*alpha)* xn**3*S(z) /
            sqrt(mu) + r0*xn/sqrt(mu)
60         #print(tn)
61         dtdx = (xn**2*C(z) + np.dot(r0v, v0v)/sqrt(mu) * xn*(1 - z*S(z)) + r0*(1
            - z*C(z)))/sqrt(mu)
62         xn += (t-tn)/dtdx
63     x = xn
64     z = x**2 *alpha
65     # the f and g expression
66     f = 1 - x**2/r0 * C(z)
67     g = t - x**3/sqrt(mu) * S(z)
68     # getting position vector rv and its absolute value r (for further
        computation of v)
69     rv = r0v*f + v0v*g

```

```

70     r = mag(rv)
71     dg = 1 - x**2/r * C(z)
72     df = sqrt(mu)/r0/r * x*(z*S(z) - 1)
73     #getting vector v
74     v = r0v*df + v0v*dg
75     #precision should be 1
76     precision = f*dg + df * g
77     print("Precision in the Kepler equation: {}".format(precision))
78     return rv, v

```

A.2 compare.py

```

1  import numpy as np
2  from helpers1 import mag, G
3  from general_definitions import masses
4  from tqdm import tqdm
5  from math import sqrt
6  from time import time
7  mu = G*masses[3]
8  def f(r):
9      return - mu/mag(r)**3 * r
10
11
12  r0 = np.array([6478100,0, 0], dtype=np.longdouble)
13  v0 = np.array([0, 10000, 0], dtype=np.longdouble)
14
15  E = 0.5 * mag(v0)**2 - mu/mag(r0)
16  h = np.cross(r0, v0)
17  a = - mu/(2*E)
18  p = mag(h)**2/(mu)
19  e = sqrt(1+(2*E*mag(h)**2)/(mu)**2)
20  rp = p/(1+e)
21  ra = p/(1 - e)
22  # Diskussion der Daten: Siehe Arbeitsjournal
23
24  dt = 30      #0.1
25  TP = 2*np.pi*pow(a, 3/2)/sqrt(mu)
26  nrev = 100
27
28  stepn = int(TP*nrev/dt + 2)
29
30  r = np.empty([stepn,3], dtype=np.longdouble)
31  r[0] = r0
32  v = np.empty([stepn,3], dtype=np.longdouble)
33  v[0] = v0

```

```

34
35
36 start = time()
37 i = 0
38 with tqdm(total=(stepn - 1)) as pbar:
39     while i*dt < TP*nrev:
40         a = f(r[i])
41         r[i + 1] = r[i] + v[i]*dt
42         v[i + 1] = v[i] + a*dt
43         i += 1
44         pbar.update(1)
45
46 print("Time elapsed {}s".format(time() - start))
47 print(r[-1])
48 print("Forward Euler with step-size {}s after {} revolutions has an imprecision
      of {}% & {}%".format(dt, nrev, mag(r[-1]-r0)/mag(r0), mag(v[-1]-v0)/mag(v0)))
49
50 #Runge-Kutta du quatrième ordre :
51
52 dt = 30 #1
53 stepn = int(TP*nrev/dt + 2)
54
55 r = np.empty([stepn,3], dtype=np.longdouble)
56 r[0] = r0
57 v = np.empty([stepn,3], dtype=np.longdouble)
58 v[0] = v0
59
60 start = time()
61 i = 0
62 with tqdm(total=(stepn - 1)) as pbar:
63     while i*dt < TP*nrev:
64         m0 = v[i]
65         k0 = f(r[i])
66         m1 = v[i] + k0*dt/2
67         k1 = f(r[i] + m0*dt/2)
68         m2 = v[i] + k1*dt/2
69         k2 = f(r[i] + m1*dt/2)
70         m3 = v[i] + k2*dt
71         k3 = f(r[i] + m2*dt)
72         r[i + 1] = r[i] + ((m0+ 2*m1+ 2*m2+ m3)*dt/6)
73         v[i + 1] = v[i] + (k0 + k1*2 + k2*2 + k3)*dt/6
74         i+=1
75         pbar.update(1)
76 print("Time elapsed {}s".format(time() - start))
77 print(r[-1])
78 print("RK4 with step-size {}s after {} revolutions has an imprecision of {}% &

```

```

    {}%".format(dt, nrev, mag(r[-1] - r0)/mag(r0), mag(v[-1] - v0)/mag(v0)))
79
80
81
82 from general_definitions import N2, N, a, b
83
84 #Gauss-Jackson avec
85
86 dt = 30
87 stepn = int(TP*nrev/dt + 2)
88
89
90 r = np.empty([stepn + N2,3], dtype=np.longdouble)
91 r[N2] = r0
92 v = np.empty([stepn + N2,3], dtype=np.longdouble)
93 v[N2] = v0
94
95
96 def fs(r):
97     return - mu/mag(r)**3 * r
98
99 def f(n):
100     return - mu/mag(r[n])**3 * r[n]
101
102
103
104
105 dt = -dt
106 for k in range(N2):
107     m0 = v[N2 - k]
108     k0 = fs(r[N2 - k])
109     m1 = v[N2 - k] + k0*dt/2
110     k1 = fs(r[N2 - k] + m0*dt/2)
111     m2 = v[N2 - k] + k1*dt/2
112     k2 = fs(r[N2 - k] + m1*dt/2)
113     m3 = v[N2 - k] + k2*dt
114     k3 = fs(r[N2 - k] + m2*dt)
115     r[N2 - k - 1] = r[N2 - k] + (m0 + m1*2 + m2*2 + m3)*dt/6
116     v[N2 - k - 1] = v[N2 - k] + (k0 + k1*2 + k2*2 + k3)*dt/6
117
118 dt = -dt
119 for k in range(N2):
120     m0 = v[N2 + k]
121     k0 = fs(r[N2 + k])
122     m1 = v[N2 + k] + k0*dt/2
123     k1 = fs(r[N2 + k] + m0*dt/2)

```

```

124     m2 = v[N2 + k] + k1*dt/2
125     k2 = fs(r[N2 + k] + m1*dt/2)
126     m3 = v[N2 + k] + k2*dt
127     k3 = fs(r[N2 + k] + m2*dt)
128     r[N2 + k + 1] = r[N2 + k] + (m0 + m1*2 + m2*2 + m3)*dt/6
129     v[N2 + k + 1] = v[N2 + k] + (k0 + k1*2 + k2*2 + k3)*dt/6
130
131     C1s = np.empty([3], dtype=np.longdouble)
132     S0 = np.empty([3], dtype=np.longdouble)
133
134     def resets():
135         global C1s, S0, Sn, sn
136         #defining C1s
137         sum1 = np.array([0,0,0], dtype=np.longdouble)
138         for k in range(N + 1):
139             sum1 += f(k)*b[N2][k]
140         C1s = v[N2]/dt - sum1
141         #Defining S0:
142         sum2 = np.array([0,0,0], dtype=np.longdouble)
143         for k in range(N + 1):
144             sum2 += f(k)*a[N2][k]
145         S0 = r[N2]/dt**2 - sum2
146         sn = C1s
147         Sn = S0
148     resets()
149
150     def getss(n):
151         global Sn, sn
152         if n == N2:
153             resets()
154             return Sn
155         elif -1 < n < N2:
156             resets()
157             for j in range(N2 - n):
158                 Sn = Sn - sn + f(N2 - j)*0.5
159                 sn -= (f(N2 - j) + f(N2 - j - 1))*0.5
160             return sn, Sn
161         elif n > N2:
162             resets()
163             for j in range(n - N2):
164                 Sn += sn + f(N2 + j)*0.5
165                 sn += (f(N2 + j) + f(N2 + j + 1))*0.5
166             return sn, Sn
167
168     def getsr(n):
169         global Sn, sn

```

```

170     if n == N + 1:
171         resets()
172         for j in range(n - N2 - 1):
173             if j != 0:
174                 sn += (f(N2 + j - 1) + f(N2 + j)) * 0.5
175                 Sn += sn + f(N2 + j) * 0.5
176
177     sn += (f(n - 2) + f(n - 1)) * 0.5
178     Sn += sn + f(n - 1) * 0.5
179     return sn, Sn
180
181 def getssr(n):
182     global sn
183     return sn + (f(n - 1) + f(n)) * 0.5
184
185 maxa = 1
186 while maxa > 0.000000000001:
187     maxa = 0
188     for n in range(N + 1):
189         if n != N2:
190             s, S = getss(n)
191             aold = f(n)
192             #correct starting value
193             sum3r = np.array([0, 0, 0], dtype=np.longdouble)
194             sum3v = np.array([0, 0, 0], dtype=np.longdouble)
195             for k in range(N + 1):
196                 a3 = f(k)
197                 sum3r += a3 * a[n][k]
198                 sum3v += a3 * b[n][k]
199             r[n] = (S + sum3r) * dt ** 2
200             v[n] = (s + sum3v) * dt
201             #check convergence of accelerations
202             anew = f(n)
203             magdif = mag(aold - anew)
204             if magdif > maxa:
205                 maxa = magdif
206
207
208 start = time()
209 #Commencing PEC cycle:
210 n = N
211 t = N2 * dt
212
213 corrsum = np.empty([2, 3], dtype=np.longdouble)
214 with tqdm(total=(stepn - N2 - 1)) as pbar:
215     while t < TP * nrev: #T is defined in general_definitions

```

```

216     #Predict:
217     s, S = getsr(n + 1)
218     psum = np.array([0,0,0], dtype=np.longdouble)
219     psumv = np.array([0,0,0], dtype=np.longdouble)
220     for k in range(N + 1):
221         ap = f(n-N+k)
222         psum += ap*a[N + 1][k]
223         psumv += ap*b[N + 1][k]
224     r[n + 1] = (psum + S)*dt**2
225     v[n + 1] = (psumv + f(n)/2 + psumv)*dt
226     n += 1
227     corrsum.fill(0)
228     #Evaluate-Correct:
229     for k in range(N):
230         ac = f(n + k - N)
231         corrsum[0] += ac*a[N][k]
232         corrsum[1] += ac*b[N][k]
233
234     for _ in range(200):
235         max = 0
236         rold = r[n]
237         r[n] = (f(n)*a[N][N] + corrsum[0] + S)*dt**2
238         v[n] = (f(n)*b[N][N] + corrsum[1] + s)*dt
239         diff = mag(rold - r[n])
240         if diff > max:
241             max = diff
242         if max < 0.00000000001:
243             break
244     t += dt
245     pbar.update(1)
246
247
248
249     print("Time elapsed {}s".format(time() - start))
250     print(r[-1])
251     print("Gauss-Jackson with step-size {}s after {} revolutions has an imprecision
        of {}, {}".format(dt, nrev, mag(r[-1] - r0)/mag(r0), mag(v[-1] - v0)/mag(v0)
        )))

```

A.3 helpers1.py

```

1  from math import sqrt, pi
2  import numpy as np
3
4  G = np.longdouble(6.6743015e-11)

```

```

5
6 def vect(a,b,c):
7     return np.array([a,b,c], dtype=np.longdouble)
8
9 def mag(array:np.ndarray):
10    if array.shape[-1] != 3:
11        print("Warning: An array of 3D vectors instead of {}D vectors expected".
12              format(array.shape[-1]))
13    return np.sqrt(np.sum(array*array, axis=-1))
14
15 def unsignedAngle(array1:np.ndarray, array2:np.ndarray):    #takes two arrays of
16    vectors as inputs, speeds up work with large datasets as we don't need a for
17    loop (about 60 times faster for large arrays, but since both operations are
18    rather fast, we don't actually care)
19    if array1.shape != array2.shape:
20        print("Arrays are not of the same shape")
21        return None
22    if array1.shape[-1] != 3:
23        print("Warning: 3-dimensional vectors are expected")
24    dot = np.sum(array1*array2, axis=-1)
25    mag = np.sqrt(np.sum(array1**2, -1)*np.sum(array2**2, axis=-1))
26    return np.arccos(dot/mag)
27
28 class object:
29     def __init__(self, r: vect, v: vect, m):
30         self.r = r
31         self.v = v
32         self.m = m
33
34 class planet(object):
35     def __init__(self, r: vect, v: vect, m, rayon):
36         super().__init__(r, v, m)
37         self.rayon = rayon
38
39 def norm(v):
40     return v/mag(v)

```

A.4 general_definitions.py

```

1 import numpy as np
2 import datetime
3 import os
4 from helpers1 import mag
5
6 a = np.array([(3250433/53222400, 572741/5702400, -8701681/39916800,
7               4026311/13305600, -917039/3193344, 7370669/39916800, -1025779/13305600,

```

```

754331/39916800, -330157/159667200],
7 [-330157/159667200, 530113/6652800, 518887/19958400, -27631/623700,
44773/1064448, -531521/19958400, 109343/9979200, -1261/475200,
45911/159667200],
8 [45911/159667200, -185839/39916800, 171137/1900800, 73643/39916800,
-25775/3193344, 77597/13305600, -98911/39916800, 24173/39916800,
-3499/53222400],
9 [-3499/53222400, 4387/4989600, -35039/4989600, 90817/950400, -20561/3193344,
2117/9979200, 2059/6652800, -317/2851200, 317/22809600],
10 [317/22809600, -2539/13305600, 55067/39916800, -326911/39916800, 14797/152064,
-326911/39916800, 55067/39916800, -2539/13305600, 317/22809600],
11 [317/22809600, -317/2851200, 2059/6652800, 2117/9979200, -20561/3193344,
90817/950400, -35039/4989600, 4387/4989600, -3499/53222400],
12 [-3499/53222400, 24173/39916800, -98911/39916800, 77597/13305600,
-25775/3193344, 73643/39916800, 171137/1900800, -185839/39916800,
45911/159667200],
13 [45911/159667200, -1261/475200, 109343/9979200, -531521/19958400,
44773/1064448, -27631/623700, 518887/19958400, 530113/6652800,
-330157/159667200],
14 [-330157/159667200, 754331/39916800, -1025779/13305600, 7370669/39916800,
-917039/3193344, 4026311/13305600, -8701681/39916800, 572741/5702400,
3250433/53222400],
15 [3250433/53222400, -11011481/19958400, 6322573/2851200, -8660609/1663200,
25162927/3193344, -159314453/19958400, 18071351/3326400, -24115843/9979200,
103798439/159667200]], dtype=np.longdouble)
16
17 b = np.asarray([[19087/89600, -427487/725760, 3498217/3628800, -500327/403200,
6467/5670, -2616161/3628800, 24019/80640, -263077/3628800, 8183/1036800],
18 [8183/1036800, 57251/403200, -1106377/3628800, 218483/725760, -69/280,
530177/3628800, -210359/3628800, 5533/403200, -425/290304],
19 [-425/290304, 76453/3628800, 5143/57600, -660127/3628800, 661/5670,
-4997/80640, 83927/3628800, -19109/3628800, 7/12800],
20 [7/12800, -23173/3628800, 29579/725760, 2497/57600, -2563/22680,
172993/3628800, -6463/403200, 2497/725760, -2497/7257600],
21 [-2497/7257600, 1469/403200, -68119/3628800, 252769/3628800, 0,
-252769/3628800, 68119/3628800, -1469/403200, 2497/7257600],
22 [2497/7257600, -2497/725760, 6463/403200, -172993/3628800, 2563/22680,
-2497/57600, -29579/725760, 23173/3628800, -7/12800],
23 [-7/12800, 19109/3628800, -83927/3628800, 4997/80640, -661/5670,
660127/3628800, -5143/57600, -76453/3628800, 425/290304],
24 [425/290304, -5533/403200, 210359/3628800, -530177/3628800, 69/280,
-218483/725760, 1106377/3628800, -57251/403200, -8183/1036800],
25 [-8183/1036800, 263077/3628800, -24019/80640, 2616161/3628800, -6467/5670,
500327/403200, -3498217/3628800, 427487/725760, -19087/89600],
26 [25713/89600, -9401029/3628800, 5393233/518400, -9839609/403200, 167287/4536,
-135352319/3628800, 10219841/403200, -40987771/3628800, 3288521/1036800]],

```

```

dtype=np.longdouble)
27
28 N = 8
29 N2 = int(N/2)
30
31 T = 5200848000
32 dt = 1000          #2 Stunden
33
34 # 1 Jahr: 31558140
35 # 165 Jahre: 5200848000
36
37 steps = int(T/dt) + N2 + 2 #+1 for the initial value + 1 because int always
    rounds off. If it doesnt, because T is a multiple of dt, then "while t <= T"
    will be true for the last iteration, as we needn't worry for floatingpoint
    imprecision (dt is an int)
38 masses = np.array([1.988410e30, 3.302e23, 4.8685e24, 6.04568e24, 6.4171e23,
    1.89858032e27, 5.684805395e26, 8.6821876e25, 1.0243e26], dtype=np.longdouble)
    # Für Jupiter baryzentrum
    : https://nssdc.gsfc.nasa.gov/planetary/factsheet/galileanfact\_table.html;
    https://nssdc.gsfc.nasa.gov/planetary/factsheet/saturniansatfact.html
39 objcount = len(masses)
40
41
42 rad_of_closest_approach = np.array([np.nan, 2.4405e6, 6.400248e6, 6.500137e6,
    3.5497e6, 7.86412e7, 4.8e8, 2.6026390e7, 2.5107930e7], dtype=np.longdouble)#
    Voir le tableau \ref
43 tick_rate = [None, 5*60*60, 1*86400, 2*86400, 5*86400, 6*86400, 7*86400,
    10*86400, 14*86400] #Arbitrary, earth and mars inspired by Wikipedia
44
45 names = ["Sun", "Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "
    Uranus", "Neptune"]
46
47 def pEpoch():
48     with open("log.txt", "r") as file:
49         ep = float(file.read())
50     return datetime.datetime.fromtimestamp(ep)
51
52
53 def unitRotationVector(t, planet:int): #Report of the IAU Working Group on
    Cartographic Coordinates and Rotational Elements: 2015
54     if planet == 3:
55         return earthUnitRotationVector(t)
56     t = float(t)
57     T = (datetime.timedelta(seconds=t) + pEpoch() - datetime.datetime(year=2000,
        month=1, day=1, hour=12, minute=0, second=0, microsecond=0)).
        total_seconds()/(36525*24*60*60)

```

```

58     if planet == 1:
59         alpha0 = 281.0103 - 0.0328*T
60         gamma0 = 61.4155 - 0.0049*T
61     elif planet == 2:
62         alpha0 = 272.76
63         gamma0 = 67.16
64     elif planet == 4:
65         alpha0 = 317.269202 - 0.10927547*T + 0.000068 * np.sin((198.991226 +
        19139.4819985*T)*np.pi/180) + 0.000238 * np.sin((226.292679 +
        38280.8511281*T)*np.pi/180) + 0.000052 * np.sin((249.663391 +
        57420.7251593*T)*np.pi/180) + 0.000009 * np.sin((266.183510 +
        76560.6367950*T)*np.pi/180) + 0.419057 * np.sin((79.398797 +
        0.5042615*T)*np.pi/180)
66         gamma0 = 54.432516 - 0.05827105*T + 0.000051 * np.cos((122.433576 +
        19139.9407476*T)*np.pi/180) + 0.000141 * np.cos((43.058401 +
        38280.8753272*T)*np.pi/180) + 0.000031 * np.cos((57.663379 +
        57420.7517205*T)*np.pi/180) + 0.000005 * np.cos((79.476401 +
        76560.6495004*T)*np.pi/180) + 1.591274 * np.cos((166.325722 +
        0.5042615*T)*np.pi/180)
67     elif planet == 5:
68         J = np.array([99.360714 + 4850.4046*T, 175.895369 + 1191.9605*T,
        300.323162 + 262.5475*T, 114.012305 + 6070.2476*T, 49.511251 +
        64.3000*T])*np.pi/180
69         alpha0 = 268.056595 - 0.006499*T + 0.000117*np.sin(J[0]) + 0.000938 * np
        .sin(J[1]) + 0.001432* np.sin(J[2]) + 0.000030* np.sin(J[3]) +
        0.002150 * np.sin(J[4])
70         gamma0 = 64.495303 + 0.002413*T + 0.000050* np.cos(J[0]) + 0.000404 * np
        .cos(J[1]) + 0.000617 * np.cos(J[2]) - 0.000013 * np.cos(J[3]) +
        0.000926 * np.cos(J[4])
71     elif planet == 6:
72         alpha0 = 40.589 - 0.036*T
73         gamma0 = 83.537 - 0.004*T
74     elif planet == 7:
75         alpha0 = 257.311
76         gamma0 = -15.175
77     elif planet == 8:
78         N = (357.85 + 52.316*T)*np.pi/180
79         alpha0 = 299.36 + 0.70 * np.sin(N)
80         gamma0 = 43.46 - 0.51 * np.cos(N)
81
82     alpha0 = alpha0*np.pi/180
83     gamma0 = gamma0*np.pi/180
84     if planet != 2 and planet != 7: #Voir la distinction entre le mouvement prograde ou
        rétrograde fait dans IAU p.
85         w = np.array([np.cos(alpha0)*np.cos(gamma0), np.sin(alpha0)*np.cos(
        gamma0), np.sin(gamma0)], dtype=np.longdouble) #\cite p.45 FoA

```

```

86     else:
87         w = - np.array([np.cos(alpha0)*np.cos(gamma0), np.sin(alpha0)*np.cos(
            gamma0), np.sin(gamma0)], dtype=np.longdouble)
88     return w/mag(w)
89
90 def earthUnitRotationVector(t1):          #Report of the IAU Working Group on
        Cartographic Coordinates and Rotational Elements: 2009
91     t1 = float(t1)
92     T = (datetime.timedelta(seconds=t1) + pEpoch() - datetime.datetime(year
        =2000, month=1, day=1, hour=12, minute=0, second=0, microsecond=0)).
        total_seconds()/(36525*24*60*60)
93     alpha = np.longdouble(0.00 - 0.641*T)*(np.pi/180)
94     gamma = np.longdouble(90 - 0.557*T)*np.pi/180
95     w = np.array([np.cos(alpha)*np.cos(gamma), np.sin(alpha)*np.cos(gamma), np.
        sin(gamma)], dtype=np.longdouble)#\cite p.45 FoA
96     return w/mag(w)

```

A.5 horizons.py

Ce programme a été adapté de <https://ssd-api.jpl.nasa.gov/doc/horizons.html>

```

1  import json
2  import requests
3  import datetime
4  import numpy as np
5  from general_definitions import pEpoch
6
7  # https://stackoverflow.com/questions/100210/what-is-the-standard-way-to-add-n-
    seconds-to-datetime-time-in-python
8
9  def horizons(id, dtime=None):
10     # Define API URL and SPK filename:
11     url = 'https://ssd.jpl.nasa.gov/api/horizons.api'
12
13     # Define the time span:
14
15     if dtime is None:
16         epoch = datetime.datetime.now()
17         start_time = epoch.strftime('%Y-%b-%d')
18         start_time += "%2000:00:00"
19         stop_time = epoch.strftime('%Y-%b-%d') + "%2000:15"
20     elif dtime!=0:
21         epoch = pEpoch()
22         start_time_d = (epoch + datetime.timedelta(0,dtime))
23         start_time = start_time_d.strftime('%Y-%b-%d') + "%20" + start_time_d.
            strftime('%H:%M:%S')

```

```

24     stop_time_d = (epoch + datetime.timedelta(0, dtime) + datetime.timedelta
25                     (0, 900))
26     stop_time = stop_time_d.strftime('%Y-%b-%d') + "%20" + stop_time_d.strftime(
27                     '%H:%M:%S')
28 elif dtime == 0:
29     epoch = pEpoch()
30     start_time = epoch.strftime('%Y-%b-%d')
31     start_time += "%2000:00:00"
32     stop_time = epoch.strftime('%Y-%b-%d') + "%2000:15"
33
34 # Build the appropriate URL for this API request:
35 # IMPORTANT: You must encode the "=" as "%3D" and the ";" as "%3B" in the
36 # Horizons COMMAND parameter specification.
37 url += "?format=json&OBJ_DATA=NO&MAKE_EPHEM='YES'&EPHEM_TYPE='VECTORS'&CENTER
38         ='%400'&VEC_TABLE='2x'&CAL_TYPE='GREGORIAN'&CSV_FORMAT=YES&TIME_DIGITS='
39         SECONDS'"
40 url += "&START_TIME='{ }'&STOP_TIME='{ }'".format(start_time, stop_time)
41 url += "&COMMAND='{ }'".format(id)
42
43 # Submit the API request and decode the JSON-response:
44 response = requests.get(url)
45 try:
46     data = json.loads(response.text)
47 except ValueError:
48     print("Unable to decode JSON results")
49
50 # If the request was valid...
51 if (response.status_code == 200):
52     data = data["result"]
53     # print(data)
54     s = data.find("$$SOE")
55     e = data.find("n.a.",)
56     data = data[s + 6:e].split(', ')
57     data = data[2:8]
58     data = list(map(str.lower, data))
59     r = np.asarray(data[:3], dtype=np.longdouble)
60     v = np.asarray(data[3:], dtype=np.longdouble)
61     return r*1000, v*1000
62
63 # If the request was invalid, extract error content and display it:
64 if (response.status_code == 400):
65     data = json.loads(response.text)
66     if "message" in data:
67         print("MESSAGE: {}".format(data["message"]))
68     else:

```

```

66         print(json.dumps(data, indent=2))
67
68     # Otherwise, some other error occurred:
69     print("response code: {0}".format(response.status_code))
70     return None

```

A.6 planetary_movement.py

```

1     import numpy as np
2     from helpers1 import *
3     from horizons import horizons
4     from vis import *
5     from general_definitions import *
6     from tqdm import tqdm
7
8     r = np.empty([steps,9,3], dtype=np.longdouble)
9     v = np.empty([steps,9,3], dtype=np.longdouble)
10
11     rinit, vinit = horizons(10)
12     r[N2][0] = rinit
13     v[N2][0] = vinit
14
15     #get initial conditions:
16     for i in range(8):
17         rh, vh = horizons(i + 1)
18         r[N2][i + 1] = rh
19         v[N2][i + 1] = vh
20     print("downloaded data from horizons api")
21
22     def ft(posvects: np.ndarray, n: int):
23         a = np.array([0,0,0], dtype=np.longdouble)
24         for i in range(objcount):
25             if i != n:
26                 rl = posvects[i] - posvects[n]    #make sure that the local r doesn't
27                 override the global one
28                 if mag(rl) == 0:
29                     print("Don't divide by 0 you idiot!")
30                 a += rl * G * (masses[i]) / mag(rl)**3
31     return a
32
33     #for all planets: calculate m1 and k1, use the vectors obtained to recalculate
34     the force field
35     #then on the basis of this, calculate m2 and k2 and so on.
36
37     #for the startup procedure of the rocket, we need to have the intermediate

```

```

    positions of the planets:
36 startup = np.empty([N + 1, 3 , objcount , 3], dtype=np.longdouble)
37
38 stepli = np.empty([4,2,objcount,3], dtype=np.longdouble)
39
40 #how to access r and v for a given step:
41     # r = r[N2 - k]
42     # v = v[N2 - k]
43 dt = -dt
44 for k in range(N2):
45     stepli[0][0] = v[N2 - k]                                #[0][0] represents m and
        therefore a velocity
46     for i in range(objcount):
47         stepli[0][1][i] = ft(r[N2 - k], i)                #[0][1] represents k (rungekutta
        .py) and therefore an acceleration
48
49     for i in range(1,4):
50         #calculate m
51         for j in range(objcount):
52             if i < 3:
53                 stepli[i][0][j] = v[N2 - k][j] + stepli[i - 1][1][j]*dt/2    #
        m1 & m2 (see rungekutta.py)
54             else:
55                 stepli[i][0][j] = v[N2 - k][j] + stepli[i - 1][1][j]*dt
56         #calculate k (see rungekutta.py), has nothing to do with the local for-
        loop variable k
57         # instead of recalculating this vector for every mass, it is saved in
        stepvectm
58         if i < 3:
59             stepvectm = r[N2 - k] + stepli[i][0]*dt/2
60         else:
61             stepvectm = r[N2 - k] + stepli[i][0]*dt
62         startup[N2 - k - 1][i - 1] = stepvectm
63         for j in range(objcount):                            #only after calculating all m's can we
        calculate all k's
64             stepli[i][1][j] = ft(stepvectm, j)
65     #finally calculate new r and v
66     r[N2 - k - 1] = r[N2 - k] + (stepli[0][0] + stepli[1][0]*2 + stepli[2][0]*2
        + stepli[3][0])*dt/6
67     v[N2 - k - 1] = v[N2 - k] + (stepli[0][1] + stepli[1][1]*2 + stepli[2][1]*2
        + stepli[3][1])*dt/6
68
69 dt = -dt
70 for k in range(N2):
71     stepli[0][0] = v[N2 + k]                                #[0][0] represents m and
        therefore a velocity

```

```

72     for i in range(objcount):
73         stepli[0][1][i] = ft(r[N2 + k], i)      #[0][1] represents k (rungekutta
            .py) and therefore an acceleration
74
75     for i in range(1,4):
76         #calculate m
77         for j in range(objcount):
78             if i < 3:
79                 stepli[i][0][j] = v[N2 + k][j] + stepli[i - 1][1][j]*dt/2      #
                    m1 & m2 (see rungekutta.py)
80             else:
81                 stepli[i][0][j] = v[N2 + k][j] + stepli[i - 1][1][j]*dt
82         #calculate k (see rungekutta.py), has nothing to do with the local for-
            loop variable k
83         # instead of recalculating this vector for every mass, it is saved in
            stepvectm
84         if i < 3:
85             stepvectm = r[N2 + k] + stepli[i][0]*dt/2
86         else:
87             stepvectm = r[N2 + k] + stepli[i][0]*dt
88         startup[N2 + k][i - 1] = stepvectm
89         for j in range(objcount):      #only after calculating all m's can we
            calculate all k's
90             stepli[i][1][j] = ft(stepvectm, j)
91         #finally calculate new r and v
92         r[N2 + k + 1] = r[N2 + k] + (stepli[0][0] + stepli[1][0]*2 + stepli[2][0]*2
            + stepli[3][0])*dt/6
93         v[N2 + k + 1] = v[N2 + k] + (stepli[0][1] + stepli[1][1]*2 + stepli[2][1]*2
            + stepli[3][1])*dt/6
94     print("Starting procedure complete")
95
96     #now that the startvalues are initialized, we no longer need stepli and can free
        its memory:
97     stepli = None
98     del stepli
99
100    #the ordinate coefficients are defined in general_definitions
101
102    C1s = np.empty([objcount,3], dtype=np.longdouble)
103    S0 = np.empty([objcount,3], dtype=np.longdouble)
104
105    def resets():
106        global C1s, S0, Sn, sn
107        #defining C1s
108        for i in range(objcount):
109            sum1 = np.array([0,0,0], dtype=np.longdouble)

```

```

110         for k in range(N + 1):
111             sum1 += ft(r[k], i)*b[N2][k]
112             C1s[i] = v[N2][i]/dt - sum1
113     #Defining S0:
114     for i in range(objcount):
115         sum2 = np.array([0,0,0], dtype=np.longdouble)
116         for k in range(N + 1):
117             sum2 += ft(r[k], i)*a[N2][k]
118         S0[i] = r[N2][i]/dt**2 - sum2
119     sn = C1s #since S0 and C1s never actually get used, passing on the
120             pointer to the array instead of copying it doesn't constitute a bug
121     Sn = S0
122     resets()
123
124 def getss(n):
125     global Sn, sn
126     if n == N2:
127         resets()
128         return Sn
129     elif -1 < n < N2:
130         resets()
131         for i in range(objcount):
132             for j in range(N2 - n):
133                 Sn[i] = Sn[i] - sn[i] + ft(r[N2 - j], i)*0.5
134                 sn[i] -= (ft(r[N2 - j], i) + ft(r[N2 - j - 1], i))*0.5
135             return sn, Sn
136     elif n > N2:
137         resets()
138         for i in range(objcount):
139             for j in range(n - N2):
140                 Sn[i] += sn[i] + ft(r[N2 + j], i)*0.5
141                 sn[i] += (ft(r[N2 + j], i) + ft(r[N2 + j + 1], i))*0.5
142             return sn, Sn
143
144 a_1 = np.empty_like(sn)
145 def getsr(n):
146     global Sn, sn, a_1
147     if n == N + 1:
148         resets()
149         for i in range(len(masses)):
150             for j in range(n - N2 - 1):
151                 a_1[i] = ft(r[N2 + j], i)
152                 if j != 0:
153                     sn[i] += (ft(r[N2 + j - 1], i) + a_1[i])*0.5
154                     Sn[i] += sn[i] + a_1[i]*0.5
155         for i in range(len(masses)):

```

```

155         a_1[i] = ft(r[n - 1], i)
156         sn[i] += (ft(r[n - 2], i) + a_1[i]) * 0.5
157         Sn[i] += sn[i] + a_1[i] * 0.5
158     return sn, Sn
159
160 def getssr(n):
161     global sn, a_1
162     cpy = sn.copy()
163     for i in range(len(masses)):
164         cpy[i] += (a_1[i] + ft(r[n], i)) * 0.5
165     return cpy
166 #Correct starting values !!!!!!!!!!!
167
168 #while max change in acceleration is higher than ...
169 #for all points:
170     #for all planets:
171         #apply corrector
172 #get max change in acceleration
173
174 maxa = 1
175 while maxa > 0.000000000001:
176     maxa = 0
177     for n in range(N + 1):
178         if n != N2:
179             s, S = getss(n)
180             oldr = r[n]
181             for o in range(len(masses)):
182                 sum3r = np.array([0, 0, 0], dtype=np.longdouble)
183                 sum3v = np.array([0, 0, 0], dtype=np.longdouble)
184                 for k in range(N + 1):
185                     ao = ft(r[k], o)
186                     sum3r += ao * a[n][k]
187                     sum3v += ao * b[n][k]
188                 r[n][o] = (S[o] + sum3r) * dt ** 2
189                 v[n][o] = (s[o] + sum3v) * dt
190             for o in range(len(masses)):
191                 aold = ft(oldr, o)
192                 anew = ft(r[n], o)
193                 magdif = mag(aold - anew)
194                 if magdif > maxa:
195                     maxa = magdif
196
197
198 #Commencing PEC cycle:
199 n = N
200 t = N2 * dt

```

```

201
202 corrsun = np.empty([2, objcount, 3], dtype=np.longdouble)           #corrsun[0]:
    position, corrsun[1]: velocity
203 with tqdm(total=steps-9) as pbar:
204     while t <= T:           #T is defined in general_definitions
205         #Predict:
206         s, S = getsr(n + 1)           #returns sn and Sn+1, sn is used for the
            predictor
207         for o in range(objcount):
208             psumr = np.array([0, 0, 0], dtype=np.longdouble)
209             psumv = np.array([0, 0, 0], dtype=np.longdouble)
210             for k in range(N + 1):
211                 pa = ft(r[n-N+k], o)
212                 psumr += pa*a[N + 1][k]
213                 psumv += pa*b[N + 1][k]
214             r[n + 1][o] = (psumr + S[o])*dt**2
215             v[n + 1][o] = (s[o] + ft(r[n], o)/2 + psumv)
216         n += 1
217         corrsun.fill(0)
218         #Evaluate-Correct:# May not make a difference for Gauss-Jackson, but
            summed Adams becomes unstable very quickly when not corrected.
219         for o in range(objcount):
220             for k in range(N):
221                 ac = ft(r[n + k - N], o)
222                 corrsun[0][o] += ac*a[N][k]
223                 corrsun[1][o] += ac*b[N][k]
224         for _ in range(200):
225             max = 0
226             s = getssr(n)
227             for o in range(objcount):
228                 rold = r[n][o]
229                 vold = v[n][o]
230                 aco = ft(r[n], o)
231                 r[n][o] = (aco*a[N][N] + corrsun[0][o] + S[o])*dt**2
232                 v[n][o] = (aco*b[N][N] + corrsun[1][o] + s[o])*dt
233                 diff = mag(rold - r[n][o])
234                 diffv = mag(vold - v[n][o])
235                 if diff > max:
236                     max = diff
237                 if diffv > max:
238                     max = diffv
239             if max < 0.0000000001:
240                 break
241         t += dt
242         pbar.update(1)
243

```

```

244 print("data calculated")
245 print(t)
246
247 with open("r.npy", "wb") as file:  #'wb': write as binary
248     np.save(file, r)
249 with open("v.npy", "wb") as file:
250     np.save(file, v)
251
252 epoch = datetime.datetime.now().replace(hour=0, minute=0, second=0, microsecond
    =0)
253 with open("log.txt", "w") as file:
254     file.write(str(epoch.timestamp()))
255
256 import os
257 os.system("shutdown.exe /s")

```

A.7 planetary_movement2.py

```

1     import numpy as np
2 from helpers1 import *
3 from horizons import horizons
4 from vis import *
5 from general_definitions import *
6 from tqdm import tqdm
7
8 r = np.empty([steps,9,3], dtype=np.longdouble)
9 v = np.empty([steps,9,3], dtype=np.longdouble)
10
11 rinit, vinit = horizons(10)
12 r[N2][0] = rinit
13 v[N2][0] = vinit
14
15 #get initial conditions:
16 for i in range(8):
17     rh, vh = horizons(i + 1)
18     r[N2][i + 1] = rh
19     v[N2][i + 1] = vh
20 print("Download from Horizons API complete!")
21
22 masses1 = masses[np.newaxis, :, np.newaxis]
23 def aG(rAllPlanets):
24     deltar = rAllPlanets[np.newaxis, :] - rAllPlanets[:, np.newaxis]
25     distanceFactor = (np.sqrt(np.sum(deltar**2, axis=-1))**3)[: ,:, np.newaxis]
26     return np.sum(np.divide(deltar, distanceFactor, out=np.zeros_like(deltar),
        where=np.rint(distanceFactor)!=0)*G*masses1, axis=1)

```

```

27
28 masses2 = masses[np.newaxis, np.newaxis, :, np.newaxis]
29 def aGeneral2(rAllPlanets):
30     deltar = rAllPlanets[:, np.newaxis, :] - rAllPlanets[:, :, np.newaxis]
31     distanceFactor = (np.sqrt(np.sum(deltar**2, axis=-1))**3)[:, :, :, np.newaxis]
32     return np.sum(np.divide(deltar, distanceFactor, out=np.zeros_like(deltar),
33                             where=np.rint(distanceFactor)!=0)*G*masses1, axis=2)
34
35 stepli = np.empty([4, 2, objcount, 3], dtype=np.longdouble)
36
37 dt = -dt
38 for k in range(N2):
39     stepli[0, 0] = v[N2 - k]
40     stepli[0, 1] = aG(r[N2 - k])
41     stepli[1, 0] = v[N2 - k] + stepli[0, 1]*dt/2
42     stepli[1, 1] = aG(r[N2 - k] + stepli[0, 0]*dt/2)
43     stepli[2, 0] = v[N2 - k] + stepli[1, 1]*dt/2
44     stepli[2, 1] = aG(r[N2 - k] + stepli[1, 0]*dt/2)
45     stepli[3, 0] = v[N2 - k] + stepli[2, 1]*dt
46     stepli[3, 1] = aG(r[N2 - k] + stepli[2, 0]*dt)
47     r[N2 - k - 1] = r[N2 - k] + (stepli[0, 0] + stepli[1, 0]*2 + stepli[2, 0]*2
48         + stepli[3, 0])*dt/6
49     v[N2 - k - 1] = v[N2 - k] + (stepli[0, 1] + stepli[1, 1]*2 + stepli[2, 1]*2
50         + stepli[3, 1])*dt/6
51
52 dt = -dt
53 for k in range(N2):
54     stepli[0, 0] = v[N2 + k]
55     stepli[0, 1] = aG(r[N2 + k])
56     stepli[1, 0] = v[N2 + k] + stepli[0, 1]*dt/2
57     stepli[1, 1] = aG(r[N2 + k] + stepli[0, 0]*dt/2)
58     stepli[2, 0] = v[N2 + k] + stepli[1, 1]*dt/2
59     stepli[2, 1] = aG(r[N2 + k] + stepli[1, 0]*dt/2)
60     stepli[3, 0] = v[N2 + k] + stepli[2, 1]*dt
61     stepli[3, 1] = aG(r[N2 + k] + stepli[2, 0]*dt)
62     r[N2 + k + 1] = r[N2 + k] + (stepli[0, 0] + stepli[1, 0]*2 + stepli[2, 0]*2
63         + stepli[3, 0])*dt/6
64     v[N2 + k + 1] = v[N2 + k] + (stepli[0, 1] + stepli[1, 1]*2 + stepli[2, 1]*2
65         + stepli[3, 1])*dt/6
66
67 #now that the startvalues are initialized, we no longer need stepli and can free
68 its memory:
69 stepli = None
70 del stepli
71

```

```

67 C1s = np.empty([objcount,3], dtype=np.longdouble)
68 S0 = np.empty([objcount,3], dtype=np.longdouble)
69
70 def ft(posvects: np.ndarray, n: int):
71     a = np.array([0,0,0], dtype=np.longdouble)
72     for i in range(objcount):
73         if i != n:
74             rl = posvects[i] - posvects[n] #make sure that the local r doesn't
75                 override the global one
76             if mag(rl) == 0:
77                 print("Don't divide by 0 you idiot!")
78             a += rl * G * (masses[i]) / mag(rl) ** 3
79
80     return a
81
82 def resets():
83     #defining C1s
84     C1s[:] = v[N2]/dt - np.sum(aGeneral2(r[:N+1]) * b[N2][:, np.newaxis, np.
85         newaxis], axis=0)
86     #Defining S0:
87     S0[:] = r[N2]/dt**2 - np.sum(aGeneral2(r[:N+1]) * a[N2][:, np.newaxis, np.
88         newaxis], axis=0)
89     sn = C1s #since S0 and C1s never actually get used, passing on the
90         pointer to the array instead of copying it doesn't constitute a bug
91     Sn = S0
92     resets()
93
94 def getss(n):
95     global Sn, sn
96     if n == N2:
97         resets()
98         return Sn
99     elif -1 < n < N2:
100         resets()
101         for i in range(N2 - n):
102             Sn = Sn - sn + aG(r[N2 - i]) * 0.5
103             sn -= (aG(r[N2 - i]) + aG(r[N2 - i - 1])) * 0.5
104         return sn, Sn
105     elif n > N2:
106         resets()
107         for i in range(n - N2):
108             Sn += sn + aG(r[N2 + i]) * 0.5
109             sn += (aG(r[N2 + i]) + aG(r[N2 + i + 1])) * 0.5
110         return sn, Sn

```

```

109 a_1 = np.empty_like(sn)
110 def getsr(n):
111     global Sn, sn, a_1
112     if n == N + 1:
113         resets()
114         for j in range(n - N2 - 1):
115             a_1 = aG(r[N2 + j])
116             if j != 0:
117                 sn += (aG(r[N2 + j - 1]) + a_1)*0.5
118                 Sn += sn + a_1*0.5
119         a_1 = aG(r[n - 1])
120         sn += (aG(r[n - 2]) + a_1)*0.5
121         Sn += sn + a_1*0.5
122     return sn, Sn
123
124 def getssr(n):
125     global sn, a_1
126     cpy = sn.copy()
127     cpy += (a_1 + aG(r[n]))*0.5
128     return cpy
129
130 maxa = 1
131 while maxa > 0.000000000001:
132     maxa = 0
133     for n in range(N + 1):
134         if n != N2:
135             s, S = getss(n)
136             oldr = r[n]
137             a0 = aGeneral2(r[:N + 1])
138             sum3r = np.sum(a0 * a[n][:, np.newaxis, np.newaxis], axis=0)
139             sum3v = np.sum(a0 * b[n][:, np.newaxis, np.newaxis], axis=0)
140             r[n] = (S + sum3r)*dt**2
141             v[n] = (s + sum3v)*dt
142             for o in range(len(masses)):
143                 aold = aG(oldr)
144                 anew = aG(r[n])
145                 maxa = np.max(mag(aold - anew))
146
147 #Commencing PEC cycle:
148 n = N
149 t = N2*dt
150
151 corrsum = np.empty([2, objcount, 3], dtype=np.longdouble) #corrsum[0]:
152     position, corrsum[1]: velocity
153 with tqdm(total=steps-9) as pbar:
154     while t <= T: #T is defined in general_definitions

```

```

154     #Predict:
155     s, S = getsr(n + 1)           #returns sn and Sn+1, sn is used for the
                                   predictor
156     pa = aGeneral2(r[n - N:n + 1])
157     psumr = np.sum(pa * a[N + 1][:, np.newaxis, np.newaxis], axis=0)
158     psumv = np.sum(pa * b[N + 1][:, np.newaxis, np.newaxis], axis=0)
159     r[n + 1] = (psumr + S)*dt**2
160     v[n + 1] = (s + aG(r[n])/2 + psumv)
161     n += 1
162     corrsum.fill(0)
163     #Evaluate-Correct:# May not make a difference for Gauss-Jackson, but
                                   summed Adams becomes unstable very quickly when not corrected.
164     ac = aGeneral2(r[n - N:n])
165     corrsum[0] = np.sum(ac * a[N][: -1, np.newaxis, np.newaxis], axis=0)
166     corrsum[1] = np.sum(ac * b[N][: -1, np.newaxis, np.newaxis], axis=0)
167
168     for _ in range(200):
169         max = 0
170         s = getssr(n)
171         rold = r[n]
172         vold = v[n]
173         aco = aG(r[n])
174         r[n] = (aco*a[:, :, np.newaxis, np.newaxis][N, N] + corrsum[0] + S)*
                dt**2
175         v[n] = (aco*b[:, :, np.newaxis, np.newaxis][N, N] + corrsum[1] + s)*
                dt
176         maxr = np.max(mag(rold - r[n]))
177         maxv = np.max(mag(vold - v[n]))
178         if maxr < 0.0000000001 and maxv < 0.0000000001:
179             break
180         t += dt
181         pbar.update(1)
182
183     print("Data calculated")
184     print(t)
185
186     with open("r.npy", "wb") as file:    #'wb': write as binary
187         np.save(file, r)
188     with open("v.npy", "wb") as file:
189         np.save(file, v)
190
191     epoch = datetime.datetime.now().replace(hour=0, minute=0, second=0, microsecond
        =0)
192     with open("log.txt", "w") as file:
193         file.write(str(epoch.timestamp()))
194

```

```

195 import os
196 os.system("shutdown.exe /s")

```

A.8 test2.py

```

1 from kepler import kepler
2 from helpers1 import G, mag
3 import numpy as np
4 from horizons import horizons
5 from general_definitions import objcount, masses, dt, N2
6 import general_definitions
7 import sys
8
9 def timeToIndex(t):
10     return int(round(t/dt)) + N2
11
12 try:
13     T = float(sys.argv[1])
14     print(T)
15
16     if T > general_definitions.T:
17         print("Time t larger than specified in general_definitions\nTerminating
            program")
18         sys.exit(1)
19     elif T < 0:
20         print("No negative values accepted")
21         sys.exit(1)
22
23     n = timeToIndex(T)
24     #read into memory position vectors:
25     #Gauss-Jackson
26     print(n)
27     with open("r.npy", "rb") as file:
28         myr = np.load(file)
29         myr = myr[n]
30     #read into memory velocity vectors for comparison:
31     with open("v.npy", "rb") as file:
32         myv = np.load(file)
33         myv = myv[n]
34 except IndexError:
35     from general_definitions import T
36     with open("r.npy", "rb") as file:
37         myr = np.load(file)
38         myr = myr[-1]
39     #read into memory velocity vectors for comparison:

```

```

40     with open("v.npy", "rb") as file:
41         myv = np.load(file)
42         myv = myv[-1]
43
44
45 def direction(r:np.ndarray):
46     #get solely the direction of the vectors:
47     direction = np.empty([len(r), 3], dtype=np.longdouble)
48     for i in range(len(r)):
49         direction[i] = r[i]/mag(r[i])
50     return direction
51
52 def dist(r:np.ndarray):
53     #calculate the distance from the sun:
54     dist = np.empty([len(r)], dtype=np.longdouble)
55     for i in range(objcount):
56         dist = mag(r[i])
57     return dist
58
59 def heliocentric(r:np.ndarray):
60     #calculate position with respect to the sun
61     helior = np.empty([objcount - 1,3], dtype=np.longdouble)
62     for i in range(objcount - 1):
63         helior[i] = r[i + 1] - r[0]
64     return helior
65
66 def error(r1:np.ndarray, r2:np.ndarray):
67     return "{:.4}".format(mag(r1 - r2)/mag(r1)*100) + " %"
68
69 def errdist(r1:np.ndarray, r2:np.ndarray):
70     return "{:.4}".format(abs(mag(r1) - mag(r2))/mag(r1)*100) + " %"
71
72 """
73 #read into memory position vectors:
74 #Gauss-Jackson
75 with open("r.npy", "rb") as file:
76     myr = np.load(file)
77     myr = myr[-1]
78 #read into memory velocity vectors for comparison:
79 with open("v.npy", "rb") as file:
80     myv = np.load(file)
81     myv = myv[-1]
82 """
83
84
85 #JPL's Horizons:

```

```

86 horizonsr = np.empty([objcount,3], dtype=np.longdouble)
87 horizonsv = np.empty([objcount,3], dtype=np.longdouble)
88 horizonsr[0], horizonsv[0] = horizons(10, T)
89 for i in range(objcount - 1):
90     horizonsr[i + 1], horizonsv[i + 1] = horizons(i + 1, T)
91
92 #Kepler Problem:
93 keplerr = np.empty([objcount - 1,3], dtype=np.longdouble)
94 keplerv = np.empty([objcount - 1,3], dtype=np.longdouble)
95 helios_r, helios_v = horizons(10, 0)
96 for i in range(objcount - 1):
97     r1, v1 = horizons(i + 1, 0)
98     keplerr[i], keplerv[i] = kepler(r1 - helios_r, v1 - helios_v, T, (masses[0]
        + masses[i + 1])*G)
99
100 #Idea with the origin
101 print("Test new origin idea : ", end="")
102 bary = np.array([0,0,0], dtype=np.longdouble)
103 for i in range(objcount):
104     bary += masses[i] * myr[i]
105 print(bary)
106 print("As a reference the same calculation for horizons: ", end="")
107 bary.fill(0)
108 for i in range(objcount):
109     bary += masses[i] * horizonsr[i]
110 print(bary)
111
112 #convert to heliocentric coodrinates:
113 myr = heliocentric(myr)
114 myv = heliocentric(myv)
115 horizonsr = heliocentric(horizonsr)
116 horizonsv = heliocentric(horizonsv)
117
118 print("PM to Horizons: ")
119 for i in range(objcount - 1):
120     print(error(horizonsr[i], myr[i]))
121 print("Kepler to Horizons: ")
122 for i in range(objcount - 1):
123     print(error(horizonsr[i], keplerr[i]))
124 print("Kepler to PM")
125 for i in range(objcount - 1):
126     print(error(myr[i], keplerr[i]))
127
128 print("Distance; PM to Horizons: ")
129 for i in range(objcount - 1):
130     print(errdist(myr[i], horizonsr[i]))

```

```

131 print("Distance; Kepler to Horizons: ")
132 for i in range(objcount - 1):
133     print(errdist(keplerr[i], horizonsr[i]))
134 print("Distance; Kepler to PM: ")
135 for i in range(objcount - 1):
136     print(errdist(keplerr[i], myr[i]))
137
138 print("Velocity tests :\n\n")
139 print("PM to Horizons: ")
140 for i in range(objcount - 1):
141     print(error(horizonsv[i], myv[i]))
142 print("Kepler to Horizons: ")
143 for i in range(objcount - 1):
144     print(error(horizonsv[i], keplerv[i]))
145 print("Kepler to PM")
146 for i in range(objcount - 1):
147     print(error(myv[i], keplerv[i]))
148
149 print("Distance; PM to Horizons: ")
150 for i in range(objcount - 1):
151     print(errdist(horizonsv[i], myv[i]))
152 print("Distance; Kepler to Horizons: ")
153 for i in range(objcount - 1):
154     print(errdist(horizonsv[i], keplerv[i]))
155 print("Distance; Kepler to PM: ")
156 for i in range(objcount - 1):
157     print(errdist(myv[i], keplerv[i]))

```

A.9 vis.py

```

1 # [1]: https://www.youtube.com/watch?v=fAztJg9oi7s
2 # [2]: https://matplotlib.org/stable/gallery/mplot3d/lorenz\_attractor.html#sphx-  
glr-gallery-mplot3d-lorenz-attractor-py
3 # [3]: https://stackoverflow.com/questions/45148704/how-to-hide-axes-and-  
gridlines
4 # [4]: https://stackoverflow.com/questions/11140163/plotting-a-3d-cube-a-sphere-  
and-a-vector
5
6
7 import numpy as np
8 import matplotlib.pyplot as plt
9 from mpl_toolkits import mplot3d
10 ax = plt.axes(projection="3d")
11 ax.set_aspect(aspect="equal")
12 ax.set_box_aspect(aspect=(1,1,1))

```

```

13
14 def reset():
15     global ax
16     ax = plt.axes(projection="3d")
17     ax.set_aspect(aspect="equal")
18
19 def point(pos:np.ndarray, col:str = 'g'):
20     x,y,z = pos
21     ax.scatter([x], [y], [z], color=col, s=2)
22
23 def lign(arr:np.ndarray, col:str):
24     ax.grid(False)
25     ax.set_box_aspect([1,1,1])
26     ax.plot(*arr.T, color=col, linewidth='0.5')    #[2]
27
28 def sphere(pos:np.ndarray, r, col:str):
29     x,y,z = pos
30     print(x,y,z)
31     u, v = np.mgrid[0:2*np.pi:30j, 0:np.pi:30j]
32     x = np.cos(u)*np.sin(v)*r + x
33     y = np.sin(u)*np.sin(v)*r + y
34     z = np.cos(v)*r + z
35     ax.plot_wireframe(x, y, z, color=col)
36
37 def setlim(m):
38     ax.set_xlim(-m, m)
39     ax.set_ylim(-m, m)
40     ax.set_zlim(-m, m)
41
42 def plot():
43     ax.set_aspect(aspect="equal")
44     ax.set_box_aspect(aspect=(1,1,1))
45     plt.show()

```

A.10 pmvisualization.py

```

1 import numpy as np
2 from vis import *
3 from datetime import datetime
4 from general_definitions import dt, N2
5
6 def timeToIndex(t):
7     return abs(int(round(t/dt)) + N2)
8 idx = timeToIndex(31535800)
9

```

```

10 with open("r.npy", "rb") as file:
11     r = np.load(file)
12
13 t0 = 373246737.99170226
14 t1 = 578499117.7331004
15 target = 6
16
17 IDX0 = timeToIndex(t0)
18 IDX1 = timeToIndex(t1)
19 Sun = r[:,50,0]
20 lign(Sun, '#fbb543')
21 Mercury = r[:,IDX1:50,1]
22 lign(Mercury, '#585858')
23 Venus = r[:,IDX1:50,2]
24 lign(Venus, '#b7711c')
25 Earth = r[:,timeToIndex(31536000):50,3]
26 lign(Earth, '#3d4782')
27 Mars = r[:,IDX1:50,4]
28 lign(Mars, '#ed795c')
29 Jupiter = r[:,IDX1:50,5]
30 lign(Jupiter, '#b8a48c')
31 Saturn = r[:,50,6]
32 lign(Saturn, '#c4ad8d')
33 Uranus = r[:,50,7]
34 lign(Uranus, '#c5ebee')
35 Neptune = r[:,50,8]
36 lign(Neptune, '#497bfe')
37
38 idx = timeToIndex(31535800)
39 try:
40     with open("rocketdata.npy", "rb") as file:
41         rr=np.load(file)
42         lign(rr, 'black')
43         point(r[IDX0, 3])
44         point(r[IDX1, target])
45 except FileNotFoundError:
46     print("Simulate a trajectory for it to be displayed")
47
48
49
50 m = max(r.min(), r.max(), key=abs)
51 setlim(m)
52
53 plot()

```

A.11 simpletraj.py

```
1 import numpy as np
2 from helpers1 import G, mag, norm
3 from general_definitions import dt, N2, masses, rad_of_closest_approach,
   earthUnitRotationVector, tick_rate, unitRotationVector, names, pEpoch
4 from kepler import C, S
5 from math import factorial, cos
6
7
8 import warnings
9 from sys import exit
10 import datetime
11 import tqdm
12 from tqdm import trange
13
14
15 def unsigned_angle(a:np.ndarray, b:np.ndarray):
16     return np.arccos(np.dot(a,b)/(mag(a)*mag(b)))
17
18 def unsignedAngle(array1:np.ndarray, array2:np.ndarray):    #takes two arrays of
   vectors as inputs, speeds up work with large datasets as we don't need a for
   loop (about 60 times faster for large arrays, but since both operations are
   rather fast, we don't actually care)
19     if array1.shape != array2.shape:
20         print("Arrays are not of the same shape")
21         return None
22     if array1.shape[-1] != 3:
23         print("Warning: 3-dimensional vectors are expected")
24     dot = np.sum(array1*array2, axis=-1)
25     mag = np.sqrt(np.sum(array1**2, -1)*np.sum(array2**2, -1))
26     return np.arccos(dot/mag)
27
28
29 def signed_angle(a:np.ndarray, b:np.ndarray, N:np.ndarray):    #https://
   stackoverflow.com/questions/5188561/signed-angle-between-two-3d-vectors-with-
   same-origin-within-the-same-plane
30     alpha = np.arctan2(np.dot(np.cross(a,b), N), np.dot(a, b))
31     if alpha > 0:
32         return alpha
33     else:
34         return 2*np.pi + alpha
35
36 def derC(z):#Checked
37     fact = 4
38     k = 1
```

```

39     s = 0
40     deltas = 100
41     while abs(deltas) > 1e-7:
42         # print("{}*z^{}/{}/{}!".format((-1)**k * k, k - 1, fact))
43         deltas = (-1)**k*k/factorial(fact)*z**(k - 1)
44         s += deltas
45         k +=1
46         fact+=2
47     return s
48
49
50
51 def derS(z):#Checked
52     fact = 5
53     k = 1
54     s = 0
55     deltas = 100
56     while abs(deltas) > 1e-7:
57         # print("{}*z^{}/{}/{}!".format((-1)**k*k, (k - 1), fact))
58         deltas = (-1)**k*k/factorial(fact)*z**(k - 1)
59         s += deltas
60         k +=1
61         fact+=2
62     return s
63
64 def timeToIndex(t):
65     return abs(int(round(t/dt)) + N2)
66
67 def hyperbolicTOF(mu, a, e, true_anomaly):#BUG CHECK
68     print("a: {}".format((-a)**3))
69     F = np.arccosh((e + np.cos(true_anomaly))/(1 + e*np.cos(true_anomaly)))
70     print(e)
71     print((e + np.cos(true_anomaly))/(1 + e*np.cos(true_anomaly)))
72     print(F)
73     print(e*np.sinh(F) - F)
74     return abs(np.sqrt((-a)**3/mu)*(e*np.sinh(F) - F))
75
76 AHHcount = 0
77 def porkchop(t1, t2, p1_index, p2_index, r1v=None, r2v=None):#assuming
78     time from ephemeris epoch; CHECKED
79     global AHHcount
80     time_1_index = timeToIndex(t1)
81     time_2_index = timeToIndex(t2)
82     case0 = False
83     if r1v is None or r2v is None:
84         r1v = rplanets[time_1_index, p1_index] - rplanets[time_1_index, 0]

```

```

84         r2v = rplanets[time_2_index, p2_index] - rplanets[time_2_index, 0]
85         case0 = True
86     v1v = vplanets[time_1_index, p1_index] - vplanets[time_1_index, 0]
87     v2v = vplanets[time_2_index, p2_index] - vplanets[time_2_index, 0]
88     mu = G*masses[0]
89     t = t2 - t1
90     n = np.cross(r1v, r2v)
91     h = np.cross(r1v, v1v)
92     dv = unsigned_angle(r1v, r2v)
93     if unsigned_angle(n, h) > np.pi/2:          #Genaue überlegungen : siehe arbeitsjournal
94         DM = -1
95         dv = 2*np.pi - dv
96     else:
97         DM = 1
98
99     sqrtmu = np.sqrt(mu)
100    if abs(dv - np.pi) < 1e-3:
101        tqdm.tqdm.write("Collinear state vectors: the plane is not unequely
102                           defined")
103        return None
104    if abs(dv - 2*np.pi) < 1e-3 or dv < 1e-3:
105        tqdm.tqdm.write("Collinear state vectors that point in the same
106                           direction; isn't optimal anyways")
107        return None
108    #CHANGE
109    r1 = mag(r1v)
110    r2 = mag(r2v)
111    A = DM*(np.sqrt(r1*r2)*np.sin(dv))/(np.sqrt(1 - np.cos(dv)))
112    zn = dv**2
113    convergence = False
114    for _ in range(50):
115        Sn = S(zn)
116        Cn = C(zn)
117        y = r1 + r2 - A*(1 - zn*Sn)/np.sqrt(Cn)
118        if y < 0:
119            AHHcount += 1
120            return None
121        xn = np.sqrt(y/Cn)
122        tn = xn**3*Sn/sqrtmu + A*np.sqrt(y)/sqrtmu
123        if t < 1 or t < 1e6:
124            if abs(t - tn) < 1e-4:
125                convergence = True
126                break
127        elif abs((t - tn)/t) < 1e-4:          #These values no longer correspond to
128            one another (new and old edition)
129            convergence = True

```

```

127         break
128     if abs(zn) < 0.5:
129         dCdz = derC(zn)
130         dSdz = derS(zn)
131     else:
132         dCdz = 1/(2*zn)*(1 - zn*Sn - 2*Cn)
133         dSdz = 1/(2*zn)*(Cn - 3*Sn)
134
135         dtdz = xn**3/sqrtmu*(dSdz - (3*Sn*dCdz)/(2*Cn)) + A/(8*sqrtmu)*((3*Sn*np
            .sqrt(y))/Cn + A/xn)
136         zn = zn + (t - tn)/dtdz
137     if not convergence:
138         tqdm.tqdm.write("The method hasn't converged, returning approximate
            values")
139     f = 1 - y/r1
140     g = A*np.sqrt(y/mu)
141     dg = 1 - y/r2
142     v1 = (r2v - f*r1v)/g
143     v2 = (dg*r2v - r1v)/g
144     vinf_departure = v1 - v1v
145     vinf_arrival = v2 - v2v
146     if case0:
147         C3 = mag(vinf_departure)**2
148         return C3, vinf_departure, vinf_arrival
149     else:
150         return vinf_departure, vinf_arrival
151
152
153
154 #Load ephemerides of all Planets:
155 with open("r.npy", "rb") as file:
156     rplanets = np.load(file)
157 with open("v.npy", "rb") as file:
158     vplanets = np.load(file)
159 now = datetime.datetime.now().replace(hour=0, minute=0, second=0, microsecond=0)
160 # epoch = datetime.datetime.fromtimestamp(os.path.getmtime('r.npy')).replace(
    hour=0, minute=0, second=0, microsecond=0) #NOTE: outdated
161 epoch = pEpoch()
162 deltat = (now - epoch).total_seconds() #Time in seconds since ephemeris "
    epoch"
163
164 transfer_time = np.array([[0, 0], [0.4, 1], [2, 3], [3, 7], [10, 17], [10, 31]],
    dtype=np.longdouble)*31536000 #transfer time from earth [min, max] in
    years; for mars a range of 1 year is used (which is more than a Hohmann
    transfer) and 0.4 is slightly lower than the record
165

```

```

166
167
168
169
170 mode0 = True
171
172
173
174 check = True
175 while check:
176     print("What is your target ?\n1)  Mercury\n2)  Venus\n4)  Mars\n5)  Jupiter "
177         )
178     x = input()
179     try:
180         x = int(x)
181         if x < 1 or x > 5:
182             print("Invalid index {} cannot be chosen".format(x))
183         else:
184             check = False
185     except ValueError:
186         print("Please provide an integer")
187
188 periapsisEarthOrbit = 185000 + 6378137          #According to Vallado:
189 if mode0:
190     departure = 3
191     arrival = x
192     mu0 = G*masses[departure]
193     mu1 = G*masses[arrival]
194     transfer_time = np.array([[np.nan, np.nan],[np.nan, np.nan],[100, 160],[0,
195         0], [120, 270], [2*365, 3*365]], dtype=np.longdouble)*86400
196     transfer_range = transfer_time[x]
197     search_range = transfer_range[1] - transfer_range[0]
198     #Up to what time in the future do we want to check for possible launch dates
199     ?
200     range_of_analysis = 20*31536000
201     sd1 = int(range_of_analysis/tick_rate[departure])          #TEST FOR POSSIBLE
202     BUG
203     sd2 = int(int(search_range)/int(tick_rate[arrival])) + 1
204     pchp1 = np.full([sd1, sd2, 1], np.nan, dtype=np.longdouble)
205     jmax = 0
206     i = 0
207     for t1 in range(int(deltat), int(deltat + range_of_analysis), int(tick_rate
208         [departure])):
209         j = 0
210         for t2 in range(int(t1) + int(transfer_range[0]), int(t1) + int(
211             transfer_range[1]), int(tick_rate[x])):

```

```

206         tmp = porkchop(t1, t2, departure, arrival)
207         if tmp is not None:
208             C3, vinf_departure, vinf_arrival = tmp
209             vinf_departure = mag(vinf_departure)
210             vinf_arrival = mag(vinf_arrival)
211             pchp1[i][j] = vinf_departure # + vinf_arrival
212         else:
213             tqdm.tqdm.write("oh")
214             pchp1[i][j] = np.nan
215         j += 1
216     if i == 0:
217         jmax = j - 1
218     i += 1
219
220
221 minindices = np.unravel_index(np.nanargmin(pchp1), pchp1.shape)
222
223
224 departureTime = minindices[0]*tick_rate[departure] + deltat
225 arrivalTime = minindices[1]*tick_rate[arrival] + transfer_range[0] +
226     departureTime
227 departureIndex = timeToIndex(departureTime)
228 arrivalIndex = timeToIndex(arrivalTime)
229 _, vinf_departure, vinf_arrival = porkchop(departureTime, arrivalTime,
230     departure, arrival)
231
232 departureSOI = mag(rplanets[departureIndex, departure] - rplanets[
233     departureIndex, 0])*(masses[departure]/masses[0])**2/5)
234 arrivalSOI = mag(rplanets[arrivalIndex, arrival] - rplanets[arrivalIndex,
235     0])*(masses[arrival]/masses[0])**2/5)
236 print("Input arrival orbit radius")
237 while True:
238     arrivalOrbitRadius = input()
239     try:
240         arrivalOrbitRadius = np.longdouble(arrivalOrbitRadius)
241         if arrivalOrbitRadius > arrivalSOI:
242             print("Specified radius greater than the SOI of {}".format(names
243                 [arrival]))
244         elif arrivalOrbitRadius < rad_of_closest_approach[arrival]:
245             print("Specified radius is less than the lower limit. Do you
246                 want to use this radius nonetheless? (y/n)")
247             if input().capitalize() == "Y":
248                 break
249         else:
250             break
251     except ValueError:

```

```

246         print("Please input a number")
247
248
249     rotationVectorEarth = earthUnitRotationVector(departureTime)
250     rotationVectorTarget = unitRotationVector(arrivalTime, x)
251     earthAtDeparture = rplanets[departureIndex, departure] - rplanets[
        departureIndex, 0]
252     targetAtArrival = rplanets[arrivalIndex, arrival] - rplanets[arrivalIndex,
        0]
253     #TODO
254     s = - np.sum(vinf_departure*rotationVectorEarth)/np.sum(vinf_departure**2)
255     N = norm(rotationVectorEarth + s*vinf_departure)
256     e = mag(vinf_departure)**2*periapsisEarthOrbit/(mu0) + 1          #See notes
        made in Vallado
257     p = (mag(vinf_departure)**2*periapsisEarthOrbit**2/(mu0) + 2*
        periapsisEarthOrbit)          #See notes made in Vallado
258     true_anomaly = np.arccos(-1/e)          #@infinity
259     P = np.array([1/(1 + e**2 + 2 * e * np.cos(true_anomaly)) * np.sqrt(p/mu0) *
        (N[2] * vinf_departure[1] * (e + np.cos(true_anomaly)) - N[1] *
        vinf_departure[2] * (e + np.cos(true_anomaly)) - 2 * e * vinf_departure
        [0] / np.tan(true_anomaly) - vinf_departure[0] / np.sin(true_anomaly) - e
        **2 * vinf_departure[0] / np.sin(true_anomaly) + N[1]**2 * vinf_departure
        [0] * (e + np.cos(true_anomaly))**2 / np.sin(true_anomaly) + N[2]**2 *
        vinf_departure[0] * (e + np.cos(true_anomaly))**2 / np.sin(true_anomaly)
        - N[0] * N[1] * vinf_departure[1] * (e + np.cos(true_anomaly))**2 / np.
        sin(true_anomaly) - N[0] * N[2] * vinf_departure[2] * (e + np.cos(
        true_anomaly))**2 / np.sin(true_anomaly)), -((np.sqrt(p/mu0) * (2 * N[2]
        * vinf_departure[0] * (e + np.cos(true_anomaly)) - 2 * N[0] *
        vinf_departure[2] * (e + np.cos(true_anomaly)) + vinf_departure[1] / np.
        sin(true_anomaly) + 2 * N[0] * N[1] * vinf_departure[0] * (e + np.cos(
        true_anomaly))**2 / np.sin(true_anomaly) + 2 * N[1]**2 * vinf_departure
        [1] * (e + np.cos(true_anomaly))**2 / np.sin(true_anomaly) + 2 * N[1] * N
        [2] * vinf_departure[2] * (e + np.cos(true_anomaly))**2 / np.sin(
        true_anomaly) - vinf_departure[1] * np.cos(2 * true_anomaly) / np.sin(
        true_anomaly)))/(2 * (1 + e**2 + 2 * e * np.cos(true_anomaly)))), -((np.
        sqrt(p/mu0) * (vinf_departure[2] - N[1] * vinf_departure[0] * (e + np.cos
        (true_anomaly)) / np.sin(true_anomaly) + N[0] * vinf_departure[1] * (e +
        np.cos(true_anomaly)) / np.sin(true_anomaly) + N[0] * N[2] *
        vinf_departure[0] * (1/np.tan(true_anomaly) + e / np.sin(true_anomaly))
        **2 + N[1] * N[2] * vinf_departure[1] * (1/np.tan(true_anomaly) + e / np.
        sin(true_anomaly))**2 + N[2]**2 * vinf_departure[2] * (1/np.tan(
        true_anomaly) + e / np.sin(true_anomaly))**2) * np.sin(true_anomaly))/(1
        + e**2 + 2 * e * np.cos(true_anomaly))], dtype=np.longdouble)
260     print("TEST 1: VALUE OUGHT TO BE NEAR 0: {}".format(N.dot(P)))
261     print("TEST 2: Value ought to be near 1: {}".format(mag(P)))
262     Q = np.cross(N, P)

```

```

263 true_anomalySOI = np.arccos((p/departureSOI - 1)/e)
264 departureVector = (p - departureSOI)/e*P + departureSOI*np.sin(
    true_anomalySOI)*Q + rplanets[departureIndex, departure] - rplanets[
    departureIndex, 0]
265
266 s = - np.sum(vinf_arrival*rotationVectorTarget)/np.sum(vinf_arrival**2)
267 N = (rotationVectorTarget + s*vinf_arrival)/mag(rotationVectorTarget + s*
    vinf_arrival)
268 print("Test: Value ought to be near 0: {}".format(N.dot(vinf_arrival)))
269 e = mag(vinf_arrival)**2*arrivalOrbitRadius/(mu1) + 1
270 p = (mag(vinf_arrival)**2*arrivalOrbitRadius**2/(mu1) + 2*arrivalOrbitRadius
    )
271 true_anomaly = -np.arccos(-1/e)
272 P = np.array([1/(1 + e**2 + 2 * e * np.cos(true_anomaly)) * np.sqrt(p/mu1) *
    (N[2] * vinf_arrival[1] * (e + np.cos(true_anomaly)) - N[1] *
    vinf_arrival[2] * (e + np.cos(true_anomaly)) - 2 * e * vinf_arrival[0] /
    np.tan(true_anomaly) - vinf_arrival[0] / np.sin(true_anomaly) - e**2 *
    vinf_arrival[0] / np.sin(true_anomaly) + N[1]**2 * vinf_arrival[0] * (e +
    np.cos(true_anomaly))**2 / np.sin(true_anomaly) + N[2]**2 * vinf_arrival
    [0] * (e + np.cos(true_anomaly))**2 / np.sin(true_anomaly) - N[0] * N[1]
    * vinf_arrival[1] * (e + np.cos(true_anomaly))**2 / np.sin(true_anomaly)
    - N[0] * N[2] * vinf_arrival[2] * (e + np.cos(true_anomaly))**2 / np.sin(
    true_anomaly)), -((np.sqrt(p/mu1) * (2 * N[2] * vinf_arrival[0] * (e + np
    .cos(true_anomaly)) - 2 * N[0] * vinf_arrival[2] * (e + np.cos(
    true_anomaly)) + vinf_arrival[1] / np.sin(true_anomaly) + 2 * N[0] * N[1]
    * vinf_arrival[0] * (e + np.cos(true_anomaly))**2 / np.sin(true_anomaly)
    + 2 * N[1]**2 * vinf_arrival[1] * (e + np.cos(true_anomaly))**2 / np.sin
    (true_anomaly) + 2 * N[1] * N[2] * vinf_arrival[2] * (e + np.cos(
    true_anomaly))**2 / np.sin(true_anomaly) - vinf_arrival[1] * np.cos(2 *
    true_anomaly) / np.sin(true_anomaly)))/(2 * (1 + e**2 + 2 * e * np.cos(
    true_anomaly)))), -((np.sqrt(p/mu1) * (vinf_arrival[2] - N[1] *
    vinf_arrival[0] * (e + np.cos(true_anomaly)) / np.sin(true_anomaly) + N
    [0] * vinf_arrival[1] * (e + np.cos(true_anomaly)) / np.sin(true_anomaly)
    + N[0] * N[2] * vinf_arrival[0] * (1/np.tan(true_anomaly) + e / np.sin(
    true_anomaly))**2 + N[1] * N[2] * vinf_arrival[1] * (1/np.tan(
    true_anomaly) + e / np.sin(true_anomaly))**2 + N[2]**2 * vinf_arrival[2]
    * (1/np.tan(true_anomaly) + e / np.sin(true_anomaly))**2) * np.sin(
    true_anomaly))/(1 + e**2 + 2 * e * np.cos(true_anomaly))], dtype=np.
    longdouble)
273 Q = np.cross(N, P)
274 true_anomalySOI = -np.arccos((p/arrivalSOI - 1)/e)
275 arrivalVector = (p - arrivalSOI)/e*P + arrivalSOI*np.sin(true_anomalySOI)*Q
    + rplanets[arrivalIndex, arrival] - rplanets[arrivalIndex, 0]
276 while True:
277     vinf_departure, vinf_arrival = porkchop(departureTime, arrivalTime,
        departure, arrival, departureVector, arrivalVector)

```

```

278     s = - np.sum(vinf_departure*rotationVectorEarth)/np.sum(vinf_departure
279                      **2)
280     N0 = (rotationVectorEarth + s*vinf_departure)/mag(rotationVectorEarth +
281                  s*vinf_departure)
282     e0 = mag(vinf_departure)**2*periapsisEarthOrbit/(mu0) + 1
283     p0 = (mag(vinf_departure)**2*periapsisEarthOrbit**2/(mu0) + 2*
284           periapsisEarthOrbit)
285     true_anomaly = np.arccos(-1/e0)
286     P0 = np.array([1/(1 + e0**2 + 2 * e0 * np.cos(true_anomaly)) * np.sqrt(
287         p0/mu0) * (N0[2] * vinf_departure[1] * (e0 + np.cos(true_anomaly)) -
288         N0[1] * vinf_departure[2] * (e0 + np.cos(true_anomaly)) - 2 * e0 *
289         vinf_departure[0] / np.tan(true_anomaly) - vinf_departure[0] / np.sin(
290         true_anomaly) - e0**2 * vinf_departure[0] / np.sin(true_anomaly) +
291         N0[1]**2 * vinf_departure[0] * (e0 + np.cos(true_anomaly))**2 / np.
292         sin(true_anomaly) + N0[2]**2 * vinf_departure[0] * (e0 + np.cos(
293         true_anomaly))**2 / np.sin(true_anomaly) - N0[0] * N0[1] *
294         vinf_departure[1] * (e0 + np.cos(true_anomaly))**2 / np.sin(
295         true_anomaly) - N0[0] * N0[2] * vinf_departure[2] * (e0 + np.cos(
296         true_anomaly))**2 / np.sin(true_anomaly)), -((np.sqrt(p0/mu0) * (2 *
297         N0[2] * vinf_departure[0] * (e0 + np.cos(true_anomaly)) - 2 * N0[0] *
298         vinf_departure[2] * (e0 + np.cos(true_anomaly)) + vinf_departure[1]
299         / np.sin(true_anomaly) + 2 * N0[0] * N0[1] * vinf_departure[0] * (e0
300         + np.cos(true_anomaly))**2 / np.sin(true_anomaly) + 2 * N0[1]**2 *
301         vinf_departure[1] * (e0 + np.cos(true_anomaly))**2 / np.sin(
302         true_anomaly) + 2 * N0[1] * N0[2] * vinf_departure[2] * (e0 + np.cos(
303         true_anomaly))**2 / np.sin(true_anomaly) - vinf_departure[1] * np.cos(
304         2 * true_anomaly) / np.sin(true_anomaly)))/(2 * (1 + e0**2 + 2 * e0
305         * np.cos(true_anomaly)))), -((np.sqrt(p0/mu0) * (vinf_departure[2] -
306         N0[1] * vinf_departure[0] * (e0 + np.cos(true_anomaly)) / np.sin(
307         true_anomaly) + N0[0] * vinf_departure[1] * (e0 + np.cos(true_anomaly)
308         )) / np.sin(true_anomaly) + N0[0] * N0[2] * vinf_departure[0] * (1/np
309         .tan(true_anomaly) + e0 / np.sin(true_anomaly))**2 + N0[1] * N0[2] *
310         vinf_departure[1] * (1/np.tan(true_anomaly) + e0 / np.sin(
311         true_anomaly))**2 + N0[2]**2 * vinf_departure[2] * (1/np.tan(
312         true_anomaly) + e0 / np.sin(true_anomaly))**2) * np.sin(true_anomaly)
313         )/(1 + e0**2 + 2 * e0 * np.cos(true_anomaly))], dtype=np.longdouble)
314     print("TEST 1: VALUE OUGHT TO BE NEAR 0: {}".format(N0.dot(P0)))
315     Q0 = np.cross(N0, P0)
316     true_anomalySOI0 = np.arccos((p0/departureSOI - 1)/e0)
317     diff1 = departureVector
318     geocentricV = (p0 - departureSOI)/e0*P0 + departureSOI*np.sin(
319         true_anomalySOI0)*Q0
320     departureVector = geocentricV + rplanets[departureIndex, departure] -
321         rplanets[departureIndex, 0]
322     diff1 = mag(diff1 - departureVector)

```

```

292     s = - np.sum(vinf_arrival*rotationVectorTarget)/np.sum(vinf_arrival**2)
293     N1 = (rotationVectorTarget + s*vinf_arrival)/mag(rotationVectorTarget +
        s*vinf_arrival)
294     e1 = mag(vinf_arrival)**2*arrivalOrbitRadius/(mu1) + 1
295     p1 = (mag(vinf_arrival)**2*arrivalOrbitRadius**2/(mu1) + 2*
        arrivalOrbitRadius)
296     true_anomaly = -np.arccos(-1/e1)
297     P1 = np.array([1/(1 + e1**2 + 2 * e1 * np.cos(true_anomaly)) * np.sqrt(
        p1/mu1) * (N1[2] * vinf_arrival[1] * (e1 + np.cos(true_anomaly)) - N1
        [1] * vinf_arrival[2] * (e1 + np.cos(true_anomaly)) - 2 * e1 *
        vinf_arrival[0] / np.tan(true_anomaly) - vinf_arrival[0] / np.sin(
        true_anomaly) - e1**2 * vinf_arrival[0] / np.sin(true_anomaly) + N1
        [1]**2 * vinf_arrival[0] * (e1 + np.cos(true_anomaly))**2 / np.sin(
        true_anomaly) + N1[2]**2 * vinf_arrival[0] * (e1 + np.cos(
        true_anomaly))**2 / np.sin(true_anomaly) - N1[0] * N1[1] *
        vinf_arrival[1] * (e1 + np.cos(true_anomaly))**2 / np.sin(
        true_anomaly) - N1[0] * N1[2] * vinf_arrival[2] * (e1 + np.cos(
        true_anomaly))**2 / np.sin(true_anomaly)), -((np.sqrt(p1/mu1) * (2 *
        N1[2] * vinf_arrival[0] * (e1 + np.cos(true_anomaly)) - 2 * N1[0] *
        vinf_arrival[2] * (e1 + np.cos(true_anomaly)) + vinf_arrival[1] / np.
        sin(true_anomaly) + 2 * N1[0] * N1[1] * vinf_arrival[0] * (e1 + np.
        cos(true_anomaly))**2 / np.sin(true_anomaly) + 2 * N1[1]**2 *
        vinf_arrival[1] * (e1 + np.cos(true_anomaly))**2 / np.sin(
        true_anomaly) + 2 * N1[1] * N1[2] * vinf_arrival[2] * (e1 + np.cos(
        true_anomaly))**2 / np.sin(true_anomaly) - vinf_arrival[1] * np.cos(2
        * true_anomaly) / np.sin(true_anomaly)))/(2 * (1 + e1**2 + 2 * e1 *
        np.cos(true_anomaly)))), -((np.sqrt(p1/mu1) * (vinf_arrival[2] - N1
        [1] * vinf_arrival[0] * (e1 + np.cos(true_anomaly)) / np.sin(
        true_anomaly) + N1[0] * vinf_arrival[1] * (e1 + np.cos(true_anomaly))
        / np.sin(true_anomaly) + N1[0] * N1[2] * vinf_arrival[0] * (1/np.tan
        (true_anomaly) + e1 / np.sin(true_anomaly))**2 + N1[1] * N1[2] *
        vinf_arrival[1] * (1/np.tan(true_anomaly) + e1 / np.sin(true_anomaly)
        )**2 + N1[2]**2 * vinf_arrival[2] * (1/np.tan(true_anomaly) + e1 / np
        .sin(true_anomaly))**2) * np.sin(true_anomaly))/(1 + e1**2 + 2 * e1 *
        np.cos(true_anomaly)))] , dtype=np.longdouble)
298     Q1 = np.cross(N1, P1)
299     true_anomalySOI1 = -np.arccos((p1/arrivalSOI - 1)/e1)
300     diff2 = arrivalVector
301     planetocentricV = (p1 - arrivalSOI)/e1*P1 + arrivalSOI*np.sin(
        true_anomalySOI1)*Q1
302     arrivalVector = planetocentricV + rplanets[arrivalIndex, arrival] -
        rplanets[arrivalIndex, 0]
303     if np.isnan(departureVector).any() or np.isnan(arrivalVector).any():
304         print("NaN encountered. Terminating program")
305         exit(1)
306     diff2 = mag(arrivalVector - diff2)

```

```

307         if diff1 + diff2 < 0.5:
308             break
309         tof0 = hyperbolicTOF(mu0, periapsisEarthOrbit/(1 - e0), e0, true_anomalySOI0
310             )
311         tof1 = abs(hyperbolicTOF(mu1, arrivalOrbitRadius/(1 - e1), e1,
312             true_anomalySOI1))
313         DepartureTime = epoch + datetime.timedelta(seconds=float(departureTime -
314             tof0))
315         ArrivalTime = epoch + datetime.timedelta(seconds=float(arrivalTime + tof1))
316         vdep = np.sqrt(mu0/p0)*(e0 + np.cos(0))*Q0
317         print("Thrust1:\nTime: {}\nPlanetocentric position: {}\nTarget velocity: {}\n
318             \nu0394v: {} ({} km/s)".format(DepartureTime.strftime("%d.%m.%y. %H:%M
319             :%S"), periapsisEarthOrbit*P0, np.sqrt(mu0/p0)*(e0 + np.cos(0))*Q0, (np.
320             sqrt(mu0/p0)*(e0 + 1) - np.sqrt(mu0/periapsisEarthOrbit))*Q0, (np.sqrt(
321             mu0/p0)*(e0 + 1) - np.sqrt(mu0/periapsisEarthOrbit))/1000))
322         print("Thrust2:\nTime: {}\nPlanetocentric position: {}\nTarget velocity: {}\n
323             \nu0394v: {} ({} km/s)".format(ArrivalTime.strftime("%d.%m.%y. %H:%M:%
324             S"), arrivalOrbitRadius*P1, np.sqrt(mu1/p1)*(e1 + np.cos(0))*Q1, (np.sqrt
325             (mu1/p1)*(e1 + 1) - np.sqrt(mu1/arrivalOrbitRadius))*Q1, (np.sqrt(mu1/p1)
326             *(e1 + 1) - np.sqrt(mu1/arrivalOrbitRadius))/1000))
327         print("Total \nu0394v required: {} km/s".format((np.sqrt(mu0/p0)*(e0 + 1) -
328             np.sqrt(mu0/periapsisEarthOrbit) + np.sqrt(mu1/p1)*(e1 + 1) - np.sqrt(mu1
329             /arrivalOrbitRadius))/1000))
330         print("Data for simulation :")
331         print("t0 = {}".format(departureTime - tof0))
332         print("t1 = {}".format(arrivalTime + tof1))
333         print("rinit = np.array([\"{}\", \"{}\", \"{}\"], dtype=np.longdouble)".
334             format(periapsisEarthOrbit*P0[0], periapsisEarthOrbit*P0[1],
335             periapsisEarthOrbit*P0[2]))
336         rv = np.sqrt(mu0/p0)*(e0 + 1)*Q0
337         print("vinit = np.array([\"{}\", \"{}\", \"{}\"], dtype=np.longdouble)".
338             format(rv[0], rv[1], rv[2]))
339         print("refpos = np.array([\"{}\", \"{}\", \"{}\"], dtype=np.longdouble)".
340             format(arrivalOrbitRadius*P1[0], arrivalOrbitRadius*P1[1],
341             arrivalOrbitRadius*P1[2]))
342         print("target = {}".format(x))

```

A.12 swingby.py

```

1 import numpy as np
2 from helpers1 import G, mag, norm
3 from general_definitions import dt, N2, masses, rad_of_closest_approach,
4     earthUnitRotationVector, tick_rate, unitRotationVector, names, pEpoch
5 from kepler import C, S
6 from math import factorial, cos

```

```

6
7
8 import warnings
9 from sys import exit
10 import datetime
11 import tqdm
12 from tqdm import trange
13
14
15 def unsigned_angle(a:np.ndarray, b:np.ndarray):
16     return np.arccos(np.dot(a,b)/(mag(a)*mag(b)))
17
18 def unsignedAngle(array1:np.ndarray, array2:np.ndarray):    #takes two arrays of
    vectors as inputs, speeds up work with large datasets as we don't need a for
    loop (about 60 times faster for large arrays, but since both operations are
    rather fast, we don't actually care)
19     if array1.shape != array2.shape:
20         print("Arrays are not of the same shape")
21         return None
22     if array1.shape[-1] != 3:
23         print("Warning: 3-dimentional vectors are expected")
24     dot = np.sum(array1*array2, axis=-1)
25     mag = np.sqrt(np.sum(array1**2, -1)*np.sum(array2**2, -1))
26     return np.arccos(dot/mag)
27
28
29 def signed_angle(a:np.ndarray, b:np.ndarray, N:np.ndarray):    #https://
    stackoverflow.com/questions/5188561/signed-angle-between-two-3d-vectors-with-
    same-origin-within-the-same-plane
30     alpha = np.arctan2(np.dot(np.cross(a,b), N), np.dot(a, b))
31     if alpha > 0:
32         return alpha
33     else:
34         return 2*np.pi + alpha
35
36 def derC(z):#Checked
37     fact = 4
38     k = 1
39     s = 0
40     deltas = 100
41     while abs(deltas) > 1e-7:
42         # print("{}*z^{}/{}!".format((-1)**k * k, k - 1, fact))
43         deltas = (-1)**k*k/factorial(fact)*z**(k - 1)
44         s += deltas
45         k +=1
46         fact+=2

```

```

47     return s
48
49
50
51 def derS(z):#Checked
52     fact = 5
53     k = 1
54     s = 0
55     deltas = 100
56     while abs(deltas) > 1e-7:
57         # print("{z}^{k}/{k}!".format((-1)**k*k, (k - 1), fact))
58         deltas = (-1)**k*k/factorial(fact)*z**(k - 1)
59         s += deltas
60         k +=1
61         fact+=2
62     return s
63
64 N2order = N2    #later used for the normal vector
65 def timeToIndex(t):
66     return abs(int(round(t/dt)) + N2order)
67
68 def hyperbolicTOF(mu, a, e, true_anomaly):#BUG CHECK
69     # print("a: {}".format((-a)**3))
70     F = np.arccosh((e + np.cos(true_anomaly))/(1 + e*np.cos(true_anomaly)))
71     # if true_anomaly > np.pi and true_anomaly < 2*np.pi:
72     #     F = -F
73     return abs(np.sqrt((-a)**3/mu)*(e*np.sinh(F) - F))
74
75 AHHcount = 0
76 def porkchop(t1, t2, p1_index, p2_index, r1v=None, r2v=None):    #assuming
    time from ephemeris epoch; CHECKED
77     global AHHcount
78     time_1_index = timeToIndex(t1)
79     time_2_index = timeToIndex(t2)
80     case0 = False
81     if r1v is None or r2v is None:
82         r1v = rplanets[time_1_index, p1_index] - rplanets[time_1_index, 0]
83         r2v = rplanets[time_2_index, p2_index] - rplanets[time_2_index, 0]
84         case0 = True
85     v1v = vplanets[time_1_index, p1_index] - vplanets[time_1_index, 0]
86     v2v = vplanets[time_2_index, p2_index] - vplanets[time_2_index, 0]
87     mu = G*masses[0]
88     t = t2 - t1
89     n = np.cross(r1v, r2v)
90     h = np.cross(r1v, v1v)
91     dv = unsigned_angle(r1v, r2v)

```

```

92     if unsigned_angle(n, h) > np.pi/2:          #Genaue überlegungen : siehe arbeitsjournal
93         DM = -1
94         dv = 2*np.pi - dv
95     else:
96         DM = 1
97
98     sqrtmu = np.sqrt(mu)
99     if abs(dv - np.pi) < 1e-3:
100         tqdm.tqdm.write("Collinear state vectors: the plane is not unequely
101                             defined")
102         return None
103     if abs(dv - 2*np.pi) < 1e-3 or dv < 1e-3:
104         tqdm.tqdm.write("Collinear state vectors that point in the same
105                             direction; isn't optimal anyways")
106         return None
107     #CHANGE
108     r1 = mag(r1v)
109     r2 = mag(r2v)
110     A = DM*(np.sqrt(r1*r2)*np.sin(dv))/(np.sqrt(1 - np.cos(dv)))
111     zn = dv**2
112     convergence = False
113     for _ in range(50):
114         Sn = S(zn)
115         Cn = C(zn)
116         y = r1 + r2 - A*(1 - zn*Sn)/np.sqrt(Cn)
117         if y < 0:
118             AHHcount += 1
119             return None
120         xn = np.sqrt(y/Cn)
121         tn = xn**3*Sn/sqrtmu + A*np.sqrt(y)/sqrtmu
122         if t < 1 or t < 1e6:
123             if abs(t - tn) < 1e-4:
124                 convergence = True
125                 break
126             elif abs((t - tn)/t) < 1e-4:          #These values no longer correspond to
127                                                         one another (new and old edition) WHY ARE THESE VALUES NORMALIZED
128                 convergence = True
129                 break
130         if abs(zn) < 0.5:
131             dCdz = derC(zn)
132             dSdz = derS(zn)
133         else:
134             dCdz = 1/(2*zn)*(1 - zn*Sn - 2*Cn)
135             dSdz = 1/(2*zn)*(Cn - 3*Sn)
136
137     dtdz = xn**3/sqrtmu*(dSdz - (3*Sn*dCdz)/(2*Cn)) + A/(8*sqrtmu)*((3*Sn*np

```

```

        .sqrt(y))/Cn + A/xn)
135     zn = zn + (t - tn)/dtdz
136     if not convergence:
137         tqdm.tqdm.write("The method hasn't converged, returning approximate
            values")
138     f = 1 - y/r1
139     g = A*np.sqrt(y/mu)
140     dg = 1 - y/r2
141     v1 = (r2v - f*r1v)/g
142     v2 = (dg*r2v - r1v)/g
143     vinf_departure = v1 - v1v
144     vinf_arrival = v2 - v2v
145     if case0:
146         C3 = mag(vinf_departure)**2
147         return C3, vinf_departure, vinf_arrival
148     else:
149         return vinf_departure, vinf_arrival
150
151
152 #Load ephemerides of all Planets:
153 with open("r.npy", "rb") as file:
154     rplanets = np.load(file)
155 with open("v.npy", "rb") as file:
156     vplanets = np.load(file)
157 now = datetime.datetime.now().replace(hour=0, minute=0, second=0, microsecond=0)
158 # epoch = datetime.datetime.fromtimestamp(os.path.getmtime('r.npy')).replace(
    hour=0, minute=0, second=0, microsecond=0) #NOTE: outdated
159 epoch = pEpoch()
160 deltat = (now - epoch).total_seconds() #Time in seconds since ephemeris "
    epoch"
161
162 transfer_time = np.array([[0, 0], [0.4, 1], [2, 3], [3, 7], [10, 17], [10, 31]],
    dtype=np.longdouble)*31536000 #transfer time from earth [min, max] in
    years; for mars a range of 1 year is used (which is more than a Hohmann
    transfer) and 0.4 is slightly lower than the record
163
164 periapsisEarthOrbit = 185000 + 6378137 #According to Vallado:
165
166
167
168 mode0 = False
169 mode1 = False
170 mode2 = False
171
172
173 check = True

```

```

174 while check:
175     print("What is your target ?\n6) Saturn\n7) Uranus\n8) Neptune")
176     x = input()
177     try:
178         x = int(x)
179         if x < 6 or x > 8:
180             print("Invalid index {} cannot be chosen".format(x))
181         else:
182             check = False
183     except ValueError:
184         print("Please provide an integer")
185
186
187 departure1 = 3#Earth
188 arrival1 = 5#Jupiter
189 departure2 = 5
190 arrival2 = x#Arrival
191 range_of_analysis = 12*31536000          #NOTE: Arbitrary choice, to be verified
192 mu1 = G*masses[departure1]
193 mu2 = G*masses[arrival1]
194 mu3 = G*masses[arrival2]
195
196 maxRadiusAtArrival = np.min(mag(rplanets[:,100, arrival2] - rplanets[:,100, 0]))
197     *(masses[arrival2]/masses[0])**2/5          #perihelion is used; I only use
198     every 100th point. With a step-size of 1000s this is around 1.2 days, still
199     totally acceptable while a lot faster
200
201 print("Input the arrival orbit's radius (min. : {})".format(
202     rad_of_closest_approach[arrival2]))
203
204 while True:
205     arrivalOrbitRadius = input()
206     try:
207         arrivalOrbitRadius = np.longdouble(arrivalOrbitRadius)
208         if arrivalOrbitRadius > maxRadiusAtArrival:
209             print("Specified radius greater than the SOI of {}".format(names[
210                 arrival2]))
211         elif arrivalOrbitRadius < rad_of_closest_approach[arrival2]:
212             print("Radius is less than the specified lower limit. Do you want to
213                 use this radius nonetheless? (y/n)")
214             if input().capitalize() == "Y":
215                 break
216         else:
217             break
218     except ValueError:
219         print("Please input a number")
220
221
222 transfer_time = np.array([[np.nan,np.nan],[0.1, 3], [0.2, 0.7], [0, 0], [0.4,

```

```

1], [2, 3], [3, 7], [10, 17], [10, 31]], dtype=np.longdouble)*31536000
    #transfer time from earth [min, max] in years; for mars a range of 1
year is used (which is more than a Hohmann transfer) and 0.4 is slightly
lower than the record; rough estimates are used for mercury and venus: In
order to establish an orbit around mercury, one would have to make multiple
venus flybys; for venus: lower limit roughly four months upper limit: more
than the hohmann transfer
214 transfer_range1 = [transfer_time[arrival1, 0] - transfer_time[departure1, 1],
    transfer_time[arrival1, 1] - transfer_time[departure1, 0]]
215 transfer_range2 = [transfer_time[arrival2, 0] - transfer_time[departure2, 1],
    transfer_time[arrival2, 1] - transfer_time[departure2, 0]]
216 search_ranget1 = transfer_range1[1] - transfer_range1[0]
217 search_ranget2 = transfer_range2[1] - transfer_range2[0]
218
219 i = 0
220 sd1 = int(range_of_analysis/tick_rate[departure1])
221 sd2 = int(search_ranget1/tick_rate[arrival1]) + 1
222 pchp1 = np.empty([sd1, sd2, 5], dtype=np.longdouble)
223 jmax = 0
224 for t1 in trange(int(deltat), int(deltat + range_of_analysis), int(tick_rate[
    departure1])):
225     j = 0
226     for t2 in range(int(t1) + int(transfer_range1[0]), int(t1) + int(
        transfer_range1[1]), int(tick_rate[arrival1])):
227         tmp = porkchop(t1, t2, departure1, arrival1)
228         if tmp is not None:
229             C3, vinf_departure, vinf_arrival = tmp
230             vinf_departure = mag(vinf_departure)
231             pchp1[i, j] = C3, vinf_departure, *vinf_arrival
232         else:
233             pchp1[i, j] = np.nan, np.nan, np.nan, np.nan, np.nan
234         j += 1
235     if i == 0:
236         jmax = j - 1
237     i += 1
238
239
240
241 sd1 = int((range_of_analysis + transfer_range1[1] - transfer_range1[0])/
    tick_rate[departure2]) + 1
242 sd2 = int(search_ranget2/tick_rate[arrival2]) + 1
243 pchp2 = np.empty([sd1, sd2, 4])
244 i = 0
245 for t1 in trange(int(deltat + transfer_range1[0]), int(deltat + transfer_range1
    [1] + range_of_analysis), int(tick_rate[departure2])):
246     j = 0

```

```

247     for t2 in range(t1 + int(transfer_range2[0]), int(t1 + transfer_range2[1]),
248                     int(tick_rate[arrival2])):
249         tmp = porkchop(t1, t2, departure2, arrival2)
250         if tmp is not None:
251             _, vinf_departure, vinf_arrival = tmp
252             vinf_arrival = mag(vinf_arrival)
253             pchp2[i, j] = *vinf_departure, vinf_arrival
254         else:
255             pchp2[i, j] = np.nan, np.nan, np.nan, np.nan
256         j += 1
257     i += 1
258     tqdm.tqdm.write("AHHHH*({})".format(AHHcount))
259
260 def swingbyTimeIndices(t):#Checked; returns the all indices that indicate an
261     arrival time at target of t
262     start = int(round((t - (transfer_range1[0] + jmax*tick_rate[arrival1]) -
263                       deltat)/tick_rate[departure1]))
264     stop = int(round((t - transfer_range1[0] - deltat)/tick_rate[departure1])) +
265         1
266     i = np.arange(start if start > 0 else 0, stop if stop < int(
267         range_of_analysis/tick_rate[departure1]) else int(range_of_analysis/
268         tick_rate[departure1])) # + 1 because the last element should be
269     inclusive; the starting index for i cannot be lower than 0 and the last
270     index mustn't exceed the highest possible index for i
271     pchp1Indices = (i, np rint((t - i*tick_rate[departure1] - deltat -
272                                transfer_range1[0])/tick_rate[arrival1]).astype(np.int64),)
273     return pchp1Indices
274
275 k = 0
276 compare = np.empty([sd1], dtype=np.dtype([('dvinf', np.longdouble), ('indices',
277     np.int64, (2, 2))]))
278 compare[:, 'dvinf'] = np.nan
279 for swingby_time in trange(int(deltat + transfer_range1[0]), int(deltat +
280     transfer_range1[1] + range_of_analysis), int(tick_rate[departure2])): #
281     for every possible swingby date, select the departure and arrival dates that
282     are best
283     pchp1Indices = swingbyTimeIndices(swingby_time)
284     if not np.size(pchp1Indices[0]):
285         break
286     jarray1 = pchp1[pchp1Indices]
287     jarray2 = pchp2[k]
288     min = np.inf
289     m = None #add safety if the two conditions below should be unfulfilled
290     for i in range(len(jarray1)):
291         for j in range(len(jarray2)):

```

```

280         if mu2/mag(jarray1[i, 2:5])**2 * (1/cos((np.pi - unsignedAngle(
            jarray1[i, 2:5], jarray2[j, 0:3]))/2) - 1) >
            rad_of_closest_approach[arrival1]: #Check if physically possible
                without crashing
281         magdiff = abs(mag(jarray1[i, 2:5]) - mag(jarray2[j, 0:3]))
282         if magdiff < 1: #minimise the difference in magnitude of the
            ingoing and outgoint vinf vectors at the swingby planet
283             if jarray1[i, 1] < min:
284                 min = jarray1[i, 1]
285                 m = i, j
286     if m is not None:
287         i, j = m
288         compare[k]['indices'] = (pchip1Indices[0][i], pchip1Indices[1][i]), (k, j)
289         compare[k]['dvinf'] = abs(mag(jarray1[i, 2:5]) - mag(jarray2[j, 0:3]))
290     else:
291         compare[k]['indices'] = (0, 0), (0, 0)
292         compare[k]['dvinf'] = np.nan
293     k += 1
294
295 #first we will choose those with low velocity difference @swingby
296 indices = np.where(compare['dvinf'] < 1)
297 # print(indices)
298 compare = compare[indices]
299 # print(compare)
300 #we have now chosen those favorable at swingby. We also want to minimize Δv at departure and arrival. We
    will do this as follows :
301 #first: make indices for pchip1&2
302 pchip1OptimizedIndices = tuple([compare['indices'][:,0, 0], compare['indices']
   [:,0, 1]])
303 pchip2OptimizedIndices = tuple([compare['indices'][:,1, 0], compare['indices']
   [:,1, 1]])
304
305 # print(pchip1[pchip1OptimizedIndices][:, 1])
306 # print(pchip2[pchip2OptimizedIndices])
307
308 cvdep = np.sqrt(mu1/periapsisEarthOrbit) #circular speed at departure
309 cvarr = np.sqrt(mu3/arrivalOrbitRadius)
310
311 #From the Energy equation follows that at distance r = infinity; E=vinf^2/2 and
    at periapsis = vp^2 / 2 - mu/rp
312 #1-Dimentional array, np.nanargmin is ok!
313 minindex = np.nanargmin(np.sqrt(pchip1[pchip1OptimizedIndices][:, 1]**2 + 2*mu1/
    periapsisEarthOrbit) - np.sqrt(mu1/periapsisEarthOrbit) + np.sqrt(pchip2[
    pchip2OptimizedIndices][:, 3]**2 + 2*mu3/arrivalOrbitRadius) - np.sqrt(mu3/
    arrivalOrbitRadius))
314

```

```

315
316 pchp1Indices = pchp1OptimizedIndices[0][minindex], pchp1OptimizedIndices[1][
    minindex]
317 pchp2Indices = pchp2OptimizedIndices[0][minindex], pchp2OptimizedIndices[1][
    minindex]
318
319 #NOTE: Add tests here
320 t1 = pchp1Indices[0]*tick_rate[departure1] + deltat
321 t2 = pchp2Indices[0]*tick_rate[departure2] + transfer_range1[0] + deltat
322 t3 = t2 + transfer_range2[0] + pchp2Indices[1]*tick_rate[arrival2]
323
324
325 # print(t1)
326 # print(t2)
327 # print(t3)
328 departureIndex = timeToIndex(t1)
329 flybyIndex = timeToIndex(t2)
330 arrivalIndex = timeToIndex(t3)
331
332 _, vinf_dep1, vinf_arr1 = porkchop(t1, t2, departure1, arrival1)
333 _, vinf_dep2, vinf_arr2 = porkchop(t2, t3, departure2, arrival2)
334
335 # print(mag(vinf_arr1) - mag(vinf_dep2))
336 # print(mag(vinf_dep1))
337 # print(mag(vinf_arr2))
338
339 # rdep = rplanets[departureIndex, departure1] - rplanets[departureIndex, 0]
340 # rarr = rplanets[flybyIndex, arrival1] - rplanets[flybyIndex, 0]
341 # vdep = vplanets[departureIndex, departure1] - vplanets[departureIndex, 0] +
    vinf_dep1
342 # varr = vplanets[flybyIndex, arrival1] - vplanets[flybyIndex, 0] + vinf_arr1
343
344 # muS = G*masses[0]
345 # evect = (mag(vdep)**2/muS - 1/mag(rdep))*rdep - np.dot(rdep, vdep)/muS*vdep
346 # hvect = np.cross(rdep, vdep)
347 # true_anomaly_dep = signed_angle(evect, rdep, norm(hvect))
348 # true_anomaly_arr = signed_angle(evect, rarr, norm(hvect))
349 # print("T1: Value ought to be near 0: {}".format(mag(evect - ((mag(varr)**2/muS
    - 1/mag(rarr))*rarr - np.dot(rarr, varr)/muS*varr)))) #Do the same thing
    with arrival and departure to verify that they describe the same thing
350 # print("T2: Value ought to be near 0: {}".format(mag(hvect - np.cross(rarr,
    varr))))
351 # p = mag(hvect)**2/muS
352 # e = mag(evect)
353 # print(e)
354 # P = norm(evect)

```

```

355 # Q = norm(np.cross(hvect, evect))
356 # def plotter(true_anomaly):
357 #     return p/(1 + e*np.cos(true_anomaly))*np.cos(true_anomaly)*P + p/(1 + e*np
        .cos(true_anomaly))*np.sin(true_anomaly)*Q
358 # trajectory = np.empty([10000, 3], dtype=np.longdouble)
359 # i = 0
360 # bool1 = True
361 # for true_anomaly in np.arange(true_anomaly_dep, true_anomaly_arr, (
        true_anomaly_arr - true_anomaly_dep)/10000):
362 #     if bool1:
363 #         print(true_anomaly)
364 #         bool1 = False
365 #         trajectory[i] = plotter(true_anomaly)
366 #         i += 1
367 # print(true_anomaly_arr, "2.549362280593937506")
368 # print(rdep, plotter(1.0563201566314143629))
369 # print(rarr, plotter(2.549362280593937506))
370 # import matplotlib.pyplot as plt
371 # from mpl_toolkits.mplot3d import Axes3D
372 # ax = plt.axes(projection="3d")
373 # ax.scatter(*rdep)
374 # ax.scatter(*rarr)
375 # ax.plot(*trajectory.T)
376 # plt.show()
377
378
379 periapsisEarthOrbit = 185000 + 6378137          #According to Vallado:
380
381 departureSOI = mag(rplanets[departureIndex, departure1] - rplanets[
        departureIndex, 0])*(masses[departure1]/masses[0])**2/5)
382 swingbySOI = mag(rplanets[flybyIndex, departure2] - rplanets[flybyIndex, 0])*(
        masses[departure2]/masses[0])**2/5)
383 arrivalSOI = mag(rplanets[arrivalIndex, arrival2] - rplanets[arrivalIndex, 0])*(
        masses[arrival2]/masses[0])**2/5)
384
385
386 #NOTE : To be verified
387 rotationVectorEarth = earthUnitRotationVector(t1)          #Not going to change a
        lot
388 rotationVectorTarget = unitRotationVector(t3, x)
389 #SOI of earth
390 s = - np.sum(vinf_depl*rotationVectorEarth)/np.sum(vinf_depl**2)
391 N = norm(rotationVectorEarth + s*vinf_depl)
392 # print("Important new test 1")
393 # print(np.dot(N, vinf_depl))
394 e = mag(vinf_depl)**2*periapsisEarthOrbit/(mu1) + 1          #See notes made in

```

Vallado

```

395 p = (mag(vinf_dep1)**2*periapsisEarthOrbit**2/(mu1) + 2*periapsisEarthOrbit)
      #See notes made in Vallado
396 true_anomaly = abs(np.arccos(-1/e))      #@infinity
397 # print(true_anomaly)
398 P = np.array([1/(1 + e**2 + 2 * e * np.cos(true_anomaly)) * np.sqrt(p/mu1) * (N
      [2] * vinf_dep1[1] * (e + np.cos(true_anomaly)) - N[1] * vinf_dep1[2] * (e +
      np.cos(true_anomaly)) - 2 * e * vinf_dep1[0] / np.tan(true_anomaly) -
      vinf_dep1[0] / np.sin(true_anomaly) - e**2 * vinf_dep1[0] / np.sin(
      true_anomaly) + N[1]**2 * vinf_dep1[0] * (e + np.cos(true_anomaly))**2 / np.
      sin(true_anomaly) + N[2]**2 * vinf_dep1[0] * (e + np.cos(true_anomaly))**2 /
      np.sin(true_anomaly) - N[0] * N[1] * vinf_dep1[1] * (e + np.cos(true_anomaly)
      )**2 / np.sin(true_anomaly) - N[0] * N[2] * vinf_dep1[2] * (e + np.cos(
      true_anomaly))**2 / np.sin(true_anomaly)), -((np.sqrt(p/mu1) * (2 * N[2] *
      vinf_dep1[0] * (e + np.cos(true_anomaly)) - 2 * N[0] * vinf_dep1[2] * (e + np
      .cos(true_anomaly)) + vinf_dep1[1] / np.sin(true_anomaly) + 2 * N[0] * N[1] *
      vinf_dep1[0] * (e + np.cos(true_anomaly))**2 / np.sin(true_anomaly) + 2 * N
      [1]**2 * vinf_dep1[1] * (e + np.cos(true_anomaly))**2 / np.sin(true_anomaly)
      + 2 * N[1] * N[2] * vinf_dep1[2] * (e + np.cos(true_anomaly))**2 / np.sin(
      true_anomaly) - vinf_dep1[1] * np.cos(2 * true_anomaly) / np.sin(true_anomaly
      )))/(2 * (1 + e**2 + 2 * e * np.cos(true_anomaly)))), -((np.sqrt(p/mu1) * (
      vinf_dep1[2] - N[1] * vinf_dep1[0] * (e + np.cos(true_anomaly)) / np.sin(
      true_anomaly) + N[0] * vinf_dep1[1] * (e + np.cos(true_anomaly)) / np.sin(
      true_anomaly) + N[0] * N[2] * vinf_dep1[0] * (1/np.tan(true_anomaly) + e / np
      .sin(true_anomaly))**2 + N[1] * N[2] * vinf_dep1[1] * (1/np.tan(true_anomaly)
      + e / np.sin(true_anomaly))**2 + N[2]**2 * vinf_dep1[2] * (1/np.tan(
      true_anomaly) + e / np.sin(true_anomaly))**2) * np.sin(true_anomaly))/(1 + e
      **2 + 2 * e * np.cos(true_anomaly))], dtype=np.longdouble)
399 # print("TEST 1: VALUE OUGHT TO BE NEAR 0: {}".format(N.dot(P)))
400 # print("TEST 2: Value ought to be near 1: {}".format(mag(P)))
401 Q = np.cross(N, P)
402 vtest = np.sqrt(mu1/p)*(-np.sin(true_anomaly)*P + (e + np.cos(true_anomaly))*Q)
403 # print("Important new test 2")
404 # print(mag(vinf_dep1 - vtest))
405 true_anomalySOI = np.arccos((p/departureSOI - 1)/e)
406 departureVector1 = (p - departureSOI)/e*P + departureSOI*np.sin(true_anomalySOI)
      *Q + rplanets[departureIndex, departure1] - rplanets[departureIndex, 0]
407 #Swingby SOI
408 N = norm(np.cross(vinf_arr1, vinf_dep2))
409 P = norm(norm(vinf_arr1) - norm(vinf_dep2))
410 Q = norm(norm(vinf_arr1) + norm(vinf_dep2))
411 # print("Test1: Value ought to be near 0: {}".format(N.dot(P)))
412 # print("Test2: Value ought to be near 0: {}".format(mag(np.cross(P, Q) - N)))
413 turning_angle = unsigned_angle(vinf_arr1, vinf_dep2)
414 rp = mu2/mag(vinf_arr1)**2*(1/cos((np.pi - turning_angle)/2) - 1)
415 e = 1/np.sin(turning_angle/2)

```

```

416 p = mag(vinf_arr1)**2 * rp**2 /mu2 + 2*rp
417 true_anomalySOIout = np.arccos((p/swingbySOI - 1)/e)
418 true_anomalySOIin = 2*np.pi - true_anomalySOIout
419 swingbyTOF2 = hyperbolicTOF(mu2, p/(1 - e**2), e, true_anomalySOIout)
420 TimeIn = t2 - swingbyTOF2
421 TimeInIndex = timeToIndex(TimeIn)
422 TimeOut = t2 + swingbyTOF2
423 TimeOutIndex = timeToIndex(TimeOut)
424 rvSOIin = swingbySOI*cos(true_anomalySOIin)*P + swingbySOI*np.sin(
    true_anomalySOIin)*Q
425 rvSOIout = swingbySOI*cos(true_anomalySOIout)*P + swingbySOI*np.sin(
    true_anomalySOIout)*Q
426 arrivalVector1 = rvSOIin + rplanets[TimeInIndex, arrival1] - rplanets[
    TimeInIndex, 0]
427 departureVector2 = rvSOIout + rplanets[TimeOutIndex, departure2] - rplanets[
    TimeOutIndex, 0]
428 #Define arrival SOI:
429 s = - np.sum(vinf_arr2*rotationVectorTarget)/np.sum(vinf_arr2**2)
430 N = (rotationVectorTarget + s*vinf_arr2)/mag(rotationVectorTarget + s*vinf_arr2)
431 # print("Test: Value ought to be near 0: {}".format(N.dot(vinf_arr2)))
432 e = mag(vinf_arr2)**2*arrivalOrbitRadius/(mu3) + 1
433 p = (mag(vinf_arr2)**2*arrivalOrbitRadius**2/(mu3) + 2*arrivalOrbitRadius)
434 true_anomaly = abs(np.arccos(-1/e))
435 P = np.array([1/(1 + e**2 + 2 * e * np.cos(true_anomaly)) * np.sqrt(p/mu3) * (N
    [2] * vinf_arr2[1] * (e + np.cos(true_anomaly)) - N[1] * vinf_arr2[2] * (e +
    np.cos(true_anomaly)) - 2 * e * vinf_arr2[0] / np.tan(true_anomaly) -
    vinf_arr2[0] / np.sin(true_anomaly) - e**2 * vinf_arr2[0] / np.sin(
    true_anomaly) + N[1]**2 * vinf_arr2[0] * (e + np.cos(true_anomaly))**2 / np.
    sin(true_anomaly) + N[2]**2 * vinf_arr2[0] * (e + np.cos(true_anomaly))**2 /
    np.sin(true_anomaly) - N[0] * N[1] * vinf_arr2[1] * (e + np.cos(true_anomaly)
    )**2 / np.sin(true_anomaly) - N[0] * N[2] * vinf_arr2[2] * (e + np.cos(
    true_anomaly))**2 / np.sin(true_anomaly)), -((np.sqrt(p/mu3) * (2 * N[2] *
    vinf_arr2[0] * (e + np.cos(true_anomaly)) - 2 * N[0] * vinf_arr2[2] * (e + np
    .cos(true_anomaly)) + vinf_arr2[1] / np.sin(true_anomaly) + 2 * N[0] * N[1] *
    vinf_arr2[0] * (e + np.cos(true_anomaly))**2 / np.sin(true_anomaly) + 2 * N
    [1]**2 * vinf_arr2[1] * (e + np.cos(true_anomaly))**2 / np.sin(true_anomaly)
    + 2 * N[1] * N[2] * vinf_arr2[2] * (e + np.cos(true_anomaly))**2 / np.sin(
    true_anomaly) - vinf_arr2[1] * np.cos(2 * true_anomaly) / np.sin(true_anomaly
    )))/(2 * (1 + e**2 + 2 * e * np.cos(true_anomaly)))), -((np.sqrt(p/mu3) * (
    vinf_arr2[2] - N[1] * vinf_arr2[0] * (e + np.cos(true_anomaly)) / np.sin(
    true_anomaly) + N[0] * vinf_arr2[1] * (e + np.cos(true_anomaly)) / np.sin(
    true_anomaly) + N[0] * N[2] * vinf_arr2[0] * (1/np.tan(true_anomaly) + e / np
    .sin(true_anomaly))**2 + N[1] * N[2] * vinf_arr2[1] * (1/np.tan(true_anomaly)
    + e / np.sin(true_anomaly))**2 + N[2]**2 * vinf_arr2[2] * (1/np.tan(
    true_anomaly) + e / np.sin(true_anomaly))**2) * np.sin(true_anomaly))/(1 + e
    **2 + 2 * e * np.cos(true_anomaly))], dtype=np.longdouble)

```

```

436 Q = np.cross(N, P)
437 true_anomalySOI = np.arccos((p/arrivalSOI - 1)/e)
438 arrivalVector2 = (p - arrivalSOI)/e*P + arrivalSOI*np.sin(true_anomalySOI)*Q +
    rplanets[arrivalIndex, arrival2] - rplanets[arrivalIndex, 0]
439 while True:
440     vinf_dep1, vinf_arr1 = porkchop(t1, TimeIn, departure1, arrival1,
        departureVector1, arrivalVector1)
441     vinf_dep2, vinf_arr2 = porkchop(TimeOut, t3, departure2, arrival2,
        departureVector2, arrivalVector2)
442     s = - np.sum(vinf_dep1*rotationVectorEarth)/np.sum(vinf_dep1**2)
443     N1 = norm(rotationVectorEarth + s*vinf_dep1)
444     e1 = mag(vinf_dep1)**2*periapsisEarthOrbit/(mu1) + 1          #See notes made
        in Vallado
445     p1 = (mag(vinf_dep1)**2*periapsisEarthOrbit**2/(mu1) + 2*periapsisEarthOrbit
        )          #See notes made in Vallado
446     true_anomaly = abs(np.arccos(-1/e1))          #@infinity
447     P1 = np.array([1/(1 + e1**2 + 2 * e1 * np.cos(true_anomaly)) * np.sqrt(p1/
        mu1) * (N1[2] * vinf_dep1[1] * (e1 + np.cos(true_anomaly)) - N1[1] *
        vinf_dep1[2] * (e1 + np.cos(true_anomaly)) - 2 * e1 * vinf_dep1[0] / np.
        tan(true_anomaly) - vinf_dep1[0] / np.sin(true_anomaly) - e1**2 *
        vinf_dep1[0] / np.sin(true_anomaly) + N1[1]**2 * vinf_dep1[0] * (e1 + np.
        cos(true_anomaly))**2 / np.sin(true_anomaly) + N1[2]**2 * vinf_dep1[0] *
        (e1 + np.cos(true_anomaly))**2 / np.sin(true_anomaly) - N1[0] * N1[1] *
        vinf_dep1[1] * (e1 + np.cos(true_anomaly))**2 / np.sin(true_anomaly) - N1
        [0] * N1[2] * vinf_dep1[2] * (e1 + np.cos(true_anomaly))**2 / np.sin(
        true_anomaly)), -((np.sqrt(p1/mu1) * (2 * N1[2] * vinf_dep1[0] * (e1 + np
        .cos(true_anomaly)) - 2 * N1[0] * vinf_dep1[2] * (e1 + np.cos(
        true_anomaly)) + vinf_dep1[1] / np.sin(true_anomaly) + 2 * N1[0] * N1[1]
        * vinf_dep1[0] * (e1 + np.cos(true_anomaly))**2 / np.sin(true_anomaly) +
        2 * N1[1]**2 * vinf_dep1[1] * (e1 + np.cos(true_anomaly))**2 / np.sin(
        true_anomaly) + 2 * N1[1] * N1[2] * vinf_dep1[2] * (e1 + np.cos(
        true_anomaly))**2 / np.sin(true_anomaly) - vinf_dep1[1] * np.cos(2 *
        true_anomaly) / np.sin(true_anomaly)))/(2 * (1 + e1**2 + 2 * e1 * np.cos(
        true_anomaly)))), -((np.sqrt(p1/mu1) * (vinf_dep1[2] - N1[1] * vinf_dep1
        [0] * (e1 + np.cos(true_anomaly)) / np.sin(true_anomaly) + N1[0] *
        vinf_dep1[1] * (e1 + np.cos(true_anomaly)) / np.sin(true_anomaly) + N1[0]
        * N1[2] * vinf_dep1[0] * (1/np.tan(true_anomaly) + e1 / np.sin(
        true_anomaly))**2 + N1[1] * N1[2] * vinf_dep1[1] * (1/np.tan(true_anomaly)
        ) + e1 / np.sin(true_anomaly))**2 + N1[2]**2 * vinf_dep1[2] * (1/np.tan(
        true_anomaly) + e1 / np.sin(true_anomaly))**2) * np.sin(true_anomaly))/(1
        + e1**2 + 2 * e1 * np.cos(true_anomaly))], dtype=np.longdouble)
448     # print("TEST 1: VALUE OUGHT TO BE NEAR 0: {}".format(N1.dot(P1)))
449     # print("TEST 2: Value ought to be near 1: {}".format(mag(P1)))
450     Q1 = np.cross(N1, P1)
451     true_anomalySOI1 = np.arccos((p1/departureSOI - 1)/e1)
452     diff0 = departureVector1

```

```

453 departureVector1 = (p1 - departureSOI)/e1*P1 + departureSOI*np.sin(
      true_anomalySOI1)*Q1 + rplanets[departureIndex, departure1] - rplanets[
      departureIndex, 0]
454 diff0 = mag(departureVector1 - diff0)
455 #Swingby SOI
456 N2 = norm(np.cross(vinf_arr1, vinf_dep2))
457 P2 = norm(norm(vinf_arr1) - norm(vinf_dep2))
458 Q2 = norm(norm(vinf_arr1) + norm(vinf_dep2))
459 # print("Test1: Value ought to be near 0: {}".format(N2.dot(P2)))
460 # print("Test2: Value ought to be near 0: {}".format(mag(np.cross(P2, Q2) -
      N2)))
461 turning_angle = unsigned_angle(vinf_arr1, vinf_dep2)
462 rp = mu2/mag(vinf_arr1)**2*(1/cos((np.pi - turning_angle)/2) - 1)
463 e2 = 1/np.sin(turning_angle/2)
464 p2 = mag(vinf_arr1)**2 * rp**2 /mu2 + 2*rp
465 true_anomalySOIout = np.arccos((p2/swingbySOI - 1)/e2)
466 true_anomalySOIin = 2*np.pi - true_anomalySOIout
467
468 swingbyTOF2 = hyperbolicTOF(mu2, p2/(1 - e2**2), e2, true_anomalySOIout)
469 TimeIn = t2 - swingbyTOF2
470 TimeInIndex = timeToIndex(TimeIn)
471 TimeOut = t2 + swingbyTOF2
472 TimeOutIndex = timeToIndex(TimeOut)
473 rvSOIin = swingbySOI*cos(true_anomalySOIin)*P2 + swingbySOI*np.sin(
      true_anomalySOIin)*Q2
474 rvSOIout = swingbySOI*cos(true_anomalySOIout)*P2 + swingbySOI*np.sin(
      true_anomalySOIout)*Q2
475 diff1 = arrivalVector1
476 arrivalVector1 = rvSOIin + rplanets[TimeInIndex, arrival1] - rplanets[
      TimeInIndex, 0]
477 diff1 = mag(arrivalVector1 - diff1)
478 diff2 = departureVector2
479 departureVector2 = rvSOIout + rplanets[TimeOutIndex, departure2] - rplanets[
      TimeOutIndex, 0]
480 diff2 = mag(departureVector2 - diff2)
481 #Define arrival SOI:
482 s = - np.sum(vinf_arr2*rotationVectorTarget)/np.sum(vinf_arr2**2)
483 N3 = (rotationVectorTarget + s*vinf_arr2)/mag(rotationVectorTarget + s*
      vinf_arr2)
484 # print("Test: Value ought to be near 0: {}".format(N3.dot(vinf_arr2)))
485 e3 = mag(vinf_arr2)**2*arrivalOrbitRadius/(mu3) + 1
486 p3 = (mag(vinf_arr2)**2*arrivalOrbitRadius**2/(mu3) + 2*arrivalOrbitRadius)
487 true_anomaly = abs(np.arccos(-1/e3))#see p 17 notes
488 P3 = np.array([1/(1 + e3**2 + 2 * e3 * np.cos(true_anomaly)) * np.sqrt(p3/
      mu3) * (N3[2] * vinf_arr2[1] * (e3 + np.cos(true_anomaly)) - N3[1] *
      vinf_arr2[2] * (e3 + np.cos(true_anomaly)) - 2 * e3 * vinf_arr2[0] / np.

```

```

tan(true_anomaly) - vinf_arr2[0] / np.sin(true_anomaly) - e3**2 *
vinf_arr2[0] / np.sin(true_anomaly) + N3[1]**2 * vinf_arr2[0] * (e3 + np.
cos(true_anomaly))**2 / np.sin(true_anomaly) + N3[2]**2 * vinf_arr2[0] *
(e3 + np.cos(true_anomaly))**2 / np.sin(true_anomaly) - N3[0] * N3[1] *
vinf_arr2[1] * (e3 + np.cos(true_anomaly))**2 / np.sin(true_anomaly) - N3
[0] * N3[2] * vinf_arr2[2] * (e3 + np.cos(true_anomaly))**2 / np.sin(
true_anomaly)), -((np.sqrt(p3/mu3) * (2 * N3[2] * vinf_arr2[0] * (e3 + np
.cos(true_anomaly)) - 2 * N3[0] * vinf_arr2[2] * (e3 + np.cos(
true_anomaly)) + vinf_arr2[1] / np.sin(true_anomaly) + 2 * N3[0] * N3[1]
* vinf_arr2[0] * (e3 + np.cos(true_anomaly))**2 / np.sin(true_anomaly) +
2 * N3[1]**2 * vinf_arr2[1] * (e3 + np.cos(true_anomaly))**2 / np.sin(
true_anomaly) + 2 * N3[1] * N3[2] * vinf_arr2[2] * (e3 + np.cos(
true_anomaly))**2 / np.sin(true_anomaly) - vinf_arr2[1] * np.cos(2 *
true_anomaly) / np.sin(true_anomaly)))/(2 * (1 + e3**2 + 2 * e3 * np.cos(
true_anomaly)))), -((np.sqrt(p3/mu3) * (vinf_arr2[2] - N3[1] * vinf_arr2
[0] * (e3 + np.cos(true_anomaly)) / np.sin(true_anomaly) + N3[0] *
vinf_arr2[1] * (e3 + np.cos(true_anomaly)) / np.sin(true_anomaly) + N3[0]
* N3[2] * vinf_arr2[0] * (1/np.tan(true_anomaly) + e3 / np.sin(
true_anomaly))**2 + N3[1] * N3[2] * vinf_arr2[1] * (1/np.tan(true_anomaly)
) + e3 / np.sin(true_anomaly))**2 + N3[2]**2 * vinf_arr2[2] * (1/np.tan(
true_anomaly) + e3 / np.sin(true_anomaly))**2) * np.sin(true_anomaly))/(1
+ e3**2 + 2 * e3 * np.cos(true_anomaly))), dtype=np.longdouble)
489 Q3 = np.cross(N3, P3)
490 true_anomalySOI3 = np.arccos((p3/arrivalSOI - 1)/e3)
491 diff3 = arrivalVector2
492 arrivalVector2 = (p3 - arrivalSOI)/e3*P3 + arrivalSOI*np.sin(
true_anomalySOI3)*Q3 + rplanets[arrivalIndex, arrival2] - rplanets[
arrivalIndex, 0]
493 diff3 = mag(arrivalVector2 - diff3)
494 # print(diff0, diff1, diff2, diff3)
495 # print("Difference in total : {}".format(diff0 + diff1 + diff2 + diff3))
496 if diff0 + diff1 + diff2 + diff3 < 0.5:
497     break
498 #Achtung: Verschiebung um 1 bei den indices!!!
499 tof0 = hyperbolicTOF(mu1, periapsisEarthOrbit/(1 - e1), e1, true_anomalySOI1)
500 tofararrival = abs(hyperbolicTOF(mu3, arrivalOrbitRadius/(1 - e3), e3,
true_anomalySOI3))
501 DepartureTime = epoch + datetime.timedelta(seconds=float(t1 - tof0))
502 ArrivalTime = epoch + datetime.timedelta(seconds=float(t3 + tofararrival))
503 print("Thrust1:\nTime: {}\nHeliocentric position: {}\nTarget velocity: {}\n\
u0394v: {} ({} km/s)".format(DepartureTime.strftime("%d.%m.%y. %H:%M:%S"),
periapsisEarthOrbit*P1 + rplanets[departureIndex, departure1] - rplanets[
departureIndex, 0], np.sqrt(mu1/p1)*(e1 + 1)*Q1, (np.sqrt(mu1/p1)*(e1 + 1) -
np.sqrt(mu1/periapsisEarthOrbit))*Q1, (np.sqrt(mu1/p1)*(e1 + 1) - np.sqrt(mu1
/periapsisEarthOrbit))/1000))
504 print("Thrust2:\nTime: {}\nHeliocentric position: {}\nTarget velocity: {}\n\

```

```

    u0394v: {} ({} km/s)".format(ArrivalTime.strftime("%d.%m.%y.      %H:%M:%S"),
    arrivalOrbitRadius*P3 + rplanets[arrivalIndex, arrival2] - rplanets[
    arrivalIndex, 0], np.sqrt(mu3/p3)*(e3 + 1)*Q3, (np.sqrt(mu3/p3)*(e3 + 1) - np
    .sqrt(mu3/arrivalOrbitRadius))*Q3, (np.sqrt(mu3/p3)*(e3 + 1) - np.sqrt(mu3/
    arrivalOrbitRadius))/1000))
505 print("Total \u0394v required: {} km/s".format((np.sqrt(mu1/p1)*(e1 + 1) - np.
    sqrt(mu1/periapsisEarthOrbit) + np.sqrt(mu3/p3)*(e3 + 1) - np.sqrt(mu3/
    arrivalOrbitRadius))/1000))
506 print("The difference in v_infty magnitude at swingby: {}".format(abs(mag(
    vinf_dep2) - mag(vinf_arr1))))
507 print("Data for simulation :")
508 print("t0 = {}".format(t1 - tof0))
509 print("t1 = {}".format(t3 + tofarrival))
510 print("rinit = np.array([\"{}\", \"{}\", \"{}\"], dtype=np.longdouble)".format(
    periapsisEarthOrbit*P1[0], periapsisEarthOrbit*P1[1], periapsisEarthOrbit*P1
    [2]))
511 rv = np.sqrt(mu1/p1)*(e1 + 1)*Q1
512 print("vinit = np.array([\"{}\", \"{}\", \"{}\"], dtype=np.longdouble)".format(
    rv[0], rv[1], rv[2]))
513 print("refpos = np.array([\"{}\", \"{}\", \"{}\"], dtype=np.longdouble)".format(
    arrivalOrbitRadius*P3[0], arrivalOrbitRadius*P3[1], arrivalOrbitRadius*P3[2])
    )
514 print("target = {}".format(x))

```

A.13 rocket.py

```

1 import numpy as np
2 from helpers1 import *
3 from general_definitions import *
4 import datetime
5 from tqdm import tqdm
6 from math import ceil
7
8 Nold = N
9 N2old = N2
10
11 b = [[94815908183/402361344000, -307515172843/373621248000,
    2709005666077/1307674368000, -2309296746931/523069747200,
    507942835493/69742632960, -4007043002299/435891456000, 2215533/250250,
    -2816016533573/435891456000, 175102023617/49816166400,
    -144690945961/104613949440, 486772076771/1307674368000,
    -160495253651/2615348736000, 2224234463/475517952000],
12 [2224234463/475517952000, 7034932909/40236134400, -599204812637/1307674368000,
    18278957351/24908083200, -373290894521/348713164800,
    550602114107/435891456000, -4538591/3891888, 360347741893/435891456000,

```

- $-153580740679/348713164800, 89210842829/523069747200,$
 $-8468059909/186810624000, 3869513783/523069747200, -4012317/7175168000],$
13 $[-4012317/7175168000, 31245653651/2615348736000, 1199987603/9144576000,$
 $-31205504599/104613949440, 116481399481/348713164800,$
 $-21844230061/62270208000, 18461231/60810750, -90050181701/435891456000,$
 $7462997467/69742632960, -7078368623/174356582400, 13891101923/1307674368000,$
 $-4478654099/2615348736000, 13681829/106748928000],$
14 $[13681829/106748928000, -5820152083/2615348736000, 5739162887/261534873600,$
 $345913117/3657830400, -72062156729/348713164800, 73700317499/435891456000,$
 $-261983/2002000, 1041757943/12454041600, -14518999879/348713164800,$
 $8038193101/523069747200, -5153476771/1307674368000, 148748057/237758976000,$
 $-1935865/41845579776],$
15 $[-1935865/41845579776, 173463193/237758976000, -1089820997/186810624000,$
 $6133014383/174356582400, 149947703/2438553600, -12825001151/87178291200,$
 $5454343/60810750, -22437447749/435891456000, 8407070279/348713164800,$
 $-639529483/74724249600, 558737863/261534873600, -869611667/2615348736000,$
 $521303/21525504000],$
16 $[521303/21525504000, -944389651/2615348736000, 3424241827/1307674368000,$
 $-6674450381/523069747200, 522979469/9963233280, 92427157/3048192000,$
 $-51350723/486486000, 20981972677/435891456000, -7081103431/348713164800,$
 $236881763/34871316480, -2134386979/1307674368000, 92427157/373621248000,$
 $-92427157/5230697472000],$
17 $[-92427157/5230697472000, 132822967/523069747200, -2274524387/1307674368000,$
 $4013113421/523069747200, -8855328071/348713164800, 32793164357/435891456000,$
 $0, -32793164357/435891456000, 8855328071/348713164800,$
 $-4013113421/523069747200, 2274524387/1307674368000, -132822967/523069747200,$
 $92427157/5230697472000],$
18 $[92427157/5230697472000, -92427157/373621248000, 2134386979/1307674368000,$
 $-236881763/34871316480, 7081103431/348713164800, -20981972677/435891456000,$
 $51350723/486486000, -92427157/3048192000, -522979469/9963233280,$
 $6674450381/523069747200, -3424241827/1307674368000, 944389651/2615348736000,$
 $-521303/21525504000],$
19 $[-521303/21525504000, 869611667/2615348736000, -558737863/261534873600,$
 $639529483/74724249600, -8407070279/348713164800, 22437447749/435891456000,$
 $-5454343/60810750, 12825001151/87178291200, -149947703/2438553600,$
 $-6133014383/174356582400, 1089820997/186810624000, -173463193/237758976000,$
 $1935865/41845579776],$
20 $[1935865/41845579776, -148748057/237758976000, 5153476771/1307674368000,$
 $-8038193101/523069747200, 14518999879/348713164800, -1041757943/12454041600,$
 $261983/2002000, -73700317499/435891456000, 72062156729/348713164800,$
 $-345913117/3657830400, -5739162887/261534873600, 5820152083/2615348736000,$
 $-13681829/106748928000],$
21 $[-13681829/106748928000, 4478654099/2615348736000, -13891101923/1307674368000,$
 $7078368623/174356582400, -7462997467/69742632960, 90050181701/435891456000,$
 $-18461231/60810750, 21844230061/62270208000, -116481399481/348713164800,$
 $31205504599/104613949440, -1199987603/9144576000, -31245653651/2615348736000,$

- 4012317/7175168000],
- 22 [4012317/7175168000, -3869513783/523069747200, 8468059909/186810624000,
-89210842829/523069747200, 153580740679/348713164800,
-360347741893/435891456000, 4538591/3891888, -550602114107/435891456000,
373290894521/348713164800, -18278957351/24908083200,
599204812637/1307674368000, -7034932909/40236134400,
-2224234463/475517952000],
- 23 [-2224234463/475517952000, 160495253651/2615348736000,
-486772076771/1307674368000, 144690945961/104613949440,
-175102023617/49816166400, 2816016533573/435891456000, -2215533/250250,
4007043002299/435891456000, -507942835493/69742632960,
2309296746931/523069747200, -2709005666077/1307674368000,
307515172843/373621248000, -94815908183/402361344000],
- 24 [106364763817/402361344000, -9000055832083/2615348736000,
491703913717/23775897600, -13247042672623/174356582400,
66393001798471/348713164800, -149831214658501/435891456000,
31975145483/69498000, -40318232897599/87178291200,
121844891963321/348713164800, -102675619234099/523069747200,
104639289835229/1307674368000, -59344946587373/2615348736000,
733526173/172204032]]
- 25
- 26 $a =$ [[132282840127/2414168064000, 192413017/1162377216,
-183706612697/348713164800, 133373184587/112086374400,
-467089093853/232475443200, 26637354127/10378368000,
-186038426051/74724249600, 5304463979/2905943040, -77220056327/77491814400,
308415783287/784604620800, -8800586233/83026944000, 1525695617/87178291200,
-1197622087/896690995200],
- 27 [-1197622087/896690995200, 14516634431/201180672000, 64188105383/1046139494400,
-227269902593/1569209241600, 23409520499/99632332800,
-25305946559/87178291200, 102644956367/373621248000, -492615587/2490808320,
74251645873/697426329600, -65181504263/1569209241600,
11614474687/1046139494400, -679920221/373621248000, 866474507/6276836966400],
- 28 [866474507/6276836966400, -2046617/653837184, 3033240127/36578304000,
613021979/28021593600, -4596037109/99632332800, 2439013/42567525,
-3989974979/74724249600, 2062196293/54486432000, -14026509859/697426329600,
380746409/49037788800, -12299341/5977939968, 446819/1334361600,
-72041419/2853107712000],
- 29 [-72041419/2853107712000, 58072601/124540416000, -423410459/83026944000,
989217599/10973491200, 296244089/77491814400, -1980839447/145297152000,
5218829519/373621248000, -1462665121/145297152000, 177708673/33210777600,
-65905289/32024678400, 946410977/1743565824000, -315501/3587584000,
207259963/31384184832000],
- 30 [207259963/31384184832000, -20754971/186810624000, 5133428761/5230697472000,
-5483137573/784604620800, 462681151/4877107200, -145599917/31135104000,
-859562191/373621248000, 574469327/217945728000, -156165377/99632332800,
493595561/784604620800, -885138967/5230697472000, 3292123/118879488000,

- $-9374747/4483454976000]$,
 31 $[-9374747/4483454976000, 1606609/47551795200, -40978319/149448499200,$
 $2478440803/1569209241600, -5916580259/697426329600, 297378623/3048192000,$
 $-123511513/14944849920, 2290603/1779148800, -38522503/697426329600,$
 $-16224893/224172748800, 162596491/5230697472000, -3203699/523069747200,$
 $3203699/6276836966400],$
 32 $[3203699/6276836966400, -1901831/217945728000, 5132899/69742632960,$
 $-164834207/392302310400, 452015111/232475443200, -22134407/2421619200,$
 $36740617/373248000, -22134407/2421619200, 452015111/232475443200,$
 $-164834207/392302310400, 5132899/69742632960, -1901831/217945728000,$
 $3203699/6276836966400],$
 33 $[3203699/6276836966400, -3203699/523069747200, 162596491/5230697472000,$
 $-16224893/224172748800, -38522503/697426329600, 2290603/1779148800,$
 $-123511513/14944849920, 297378623/3048192000, -5916580259/697426329600,$
 $2478440803/1569209241600, -40978319/149448499200, 1606609/47551795200,$
 $-9374747/4483454976000],$
 34 $[-9374747/4483454976000, 3292123/118879488000, -885138967/5230697472000,$
 $493595561/784604620800, -156165377/99632332800, 574469327/217945728000,$
 $-859562191/373621248000, -145599917/31135104000, 462681151/4877107200,$
 $-5483137573/784604620800, 5133428761/5230697472000, -20754971/186810624000,$
 $207259963/31384184832000],$
 35 $[207259963/31384184832000, -315501/3587584000, 946410977/1743565824000,$
 $-65905289/32024678400, 177708673/33210777600, -1462665121/145297152000,$
 $5218829519/373621248000, -1980839447/145297152000, 296244089/77491814400,$
 $989217599/10973491200, -423410459/83026944000, 58072601/124540416000,$
 $-72041419/2853107712000],$
 36 $[-72041419/2853107712000, 446819/1334361600, -12299341/5977939968,$
 $380746409/49037788800, -14026509859/697426329600, 2062196293/54486432000,$
 $-3989974979/74724249600, 2439013/42567525, -4596037109/99632332800,$
 $613021979/28021593600, 3033240127/36578304000, -2046617/653837184,$
 $866474507/6276836966400],$
 37 $[866474507/6276836966400, -679920221/373621248000, 11614474687/1046139494400,$
 $-65181504263/1569209241600, 74251645873/697426329600, -492615587/2490808320,$
 $102644956367/373621248000, -25305946559/87178291200, 23409520499/99632332800,$
 $-227269902593/1569209241600, 64188105383/1046139494400,$
 $14516634431/201180672000, -1197622087/896690995200],$
 38 $[-1197622087/896690995200, 1525695617/87178291200, -8800586233/83026944000,$
 $308415783287/784604620800, -77220056327/77491814400, 5304463979/2905943040,$
 $-186038426051/74724249600, 26637354127/10378368000,$
 $-467089093853/232475443200, 133373184587/112086374400,$
 $-183706612697/348713164800, 192413017/1162377216,$
 $132282840127/2414168064000],$
 39 $[132282840127/2414168064000, -1866476396209/2615348736000,$
 $2040667428953/475517952000, -24757711059413/1569209241600,$
 $27597902895821/697426329600, -31173587791351/435891456000,$
 $5116077905657/53374464000, -42070857451313/435891456000,$

```

50972790156553/697426329600, -64631301332531/1569209241600,
88195348546091/5230697472000, -12555699585959/2615348736000,
2504631949133/2853107712000]]
40 N=12
41 N2 = int(N/2)
42
43
44 #Faster function for the calculation of accelerations
45 masses1 = masses[np.newaxis, :, np.newaxis]
46 def aGeneral(rAllPlanets):
47     deltar = rAllPlanets[np.newaxis, :] - rAllPlanets[:, np.newaxis]
48     distanceFactor = (np.sqrt(np.sum(deltar**2, axis=-1))**3)[:, :, np.newaxis]
49     return np.sum(np.divide(deltar, distanceFactor, out=np.zeros_like(deltar),
        where=np.rint(distanceFactor)!=0)*G*masses1, axis=1)
50
51
52
53 #UI:
54 # t0 = 373246737.99170226
55 # t1 = 578499117.7331004
56 # rinit = np.array(["4144393.343943437", "-4942696.470309006",
    "-1211826.1813648157"], dtype=np.longdouble)
57 # vinit = np.array(["11127.632798784025", "7561.947853523697",
    "7212.976972498207"], dtype=np.longdouble)
58 # refpos = np.array(["116981803.91779557", "485211077.83533645",
    "-29756805.910740476"], dtype=np.longdouble)
59 # target = 6
60
61 #testing values:
62 t0 = 0
63 t1 = 31536000
64 rinit = np.array([6478100,0, 0], dtype=np.longdouble)
65 vinit = np.array([0, 10000, 0], dtype=np.longdouble)
66 refpos = np.array([0,0,0], dtype=np.longdouble)
67 target = 3
68
69 #r0, v0
70 import time
71 starttime = time.time()
72 #LOADING SELF-CALCULATED EPHEMERIS:
73 with open("r.npy", "rb") as file:
74     rplanets_load = np.load(file)
75 with open("v.npy", "rb") as file:
76     vplanets_load = np.load(file)
77
78 print("Time to load files: {:.2f}s".format(time.time() - starttime))

```

```

79
80 #Startup and Interpolation
81 planetarydt = dt
82 dt = 100
83 sl = int(planetarydt/dt) - 1 #This only works if multiples of 100 (dt)
      are used as planetarydt
84 forwards = sl//2 + sl%2
85 backwards = sl//2
86
87 #redefining sl for later purposes:
88 sl += 1
89
90
91 T = t1 - t0
92 steps = int(T/dt) + N2 + 2
93
94 rplanets = np.empty([steps,9,3], dtype=np.longdouble)
95 vplanets = np.empty([steps,9,3], dtype=np.longdouble)
96
97
98
99 b0 = t0 - dt*N2 #backpoint 0
100 IDX0 = ceil(b0/planetarydt) + N2old #used later for slicing
101 print(IDX0)
102 IDX1 = int(t1/planetarydt) + N2old
103 print(IDX1)
104 p0 = ceil(b0/planetarydt)*sl - round(b0/dt) #step in terms of dt -
      nearest calculated point = number of steps to be interpolated backwards &
      index of the first grid-point to be used
105 p1 = int(t1/planetarydt)*sl - round(b0/dt) #from b0 to t1, how many
      points are used.
106 print(p0)
107 print(p1)
108
109 rplanets[p0:p1 + 1:sl] = rplanets_load[IDX0:IDX1 + 1]
110 vplanets[p0:p1 + 1:sl] = vplanets_load[IDX0:IDX1 + 1]
111
112 rplanets_load = None
113 del rplanets_load
114 vplanets_load = None
115 del vplanets_load
116
117 masses2 = masses[np.newaxis, np.newaxis, :, np.newaxis]
118 def aGeneral2(rAllPlanets):
119     deltar = rAllPlanets[:,np.newaxis, :] - rAllPlanets[:, :, np.newaxis]
120     distanceFactor = (np.sqrt(np.sum(deltar**2, axis=-1))**3)[:,:, :, np.newaxis]

```

```

121     return np.sum(np.divide(deltar, distanceFactor, out=np.zeros_like(deltar),
122                             where=np.rint(distanceFactor)!=0)*G*masses1, axis=2)
122
123 dt = -dt
124 for i in range(p0):
125     m0 = vplanets[p0 - i]
126     k0 = aGeneral(rplanets[p0 - i])
127     m1 = vplanets[p0 - i] + k0*dt/2
128     k1 = aGeneral(rplanets[p0 - i] + m0*dt/2)
129     m2 = vplanets[p0 - i] + k1*dt/2
130     k2 = aGeneral(rplanets[p0 - i] + m1*dt/2)
131     m3 = vplanets[p0 - i] + k2*dt/2
132     k3 = aGeneral(rplanets[p0 - i] + m2*dt/2)
133     vplanets[p0 - i - 1] = vplanets[p0 - i] + (k0 + 2*k1 + 2*k2 + k3)*dt/6
134     rplanets[p0 - i - 1] = rplanets[p0 - i] + (m0 + 2*m1 + 2*m2 + m3)*dt/6
135
136 with tqdm(total=(sl - 1)*4) as pbar:
137     for i in range(backwards):
138         m0 = vplanets[p0 + sl - i:p1 + 1:sl]
139         k0 = aGeneral2(rplanets[p0 + sl - i:p1 + 1:sl])
140         pbar.update(1)
141         m1 = vplanets[p0 + sl - i:p1 + 1:sl] + k0*dt/2
142         k1 = aGeneral2(rplanets[p0 + sl - i:p1 + 1:sl] + m0*dt/2)
143         pbar.update(1)
144         m2 = vplanets[p0 + sl - i:p1 + 1:sl] + k1*dt/2
145         k2 = aGeneral2(rplanets[p0 + sl - i:p1 + 1:sl] + m1*dt/2)
146         pbar.update(1)
147         m3 = vplanets[p0 + sl - i:p1 + 1:sl] + k2*dt
148         k3 = aGeneral2(rplanets[p0 + sl - i:p1 + 1:sl] + m2*dt)
149         pbar.update(1)
150         vplanets[p0 + sl - i - 1:p1 + 1:sl] = vplanets[p0 + sl - i:p1 + 1:sl] +
151             (k0 + 2*k1 + 2*k2 + k3)*dt/6
152         rplanets[p0 + sl - i - 1:p1 + 1:sl] = rplanets[p0 + sl - i:p1 + 1:sl] +
153             (m0 + 2*m1 + 2*m2 + m3)*dt/6
154
155 dt = -dt
156 for i in range(forwards):
157     m0 = vplanets[p0 + i:p1:sl]
158     k0 = aGeneral2(rplanets[p0 + i:p1:sl]) #p1 is the last point that
159         is a multiple of sl. It is not included. This is deliberate behaviour
160         , we will extrapolate later
161     pbar.update(1)
162     m1 = vplanets[p0 + i:p1:sl] + k0*dt/2
163     k1 = aGeneral2(rplanets[p0 + i:p1:sl] + m0*dt/2)
164     pbar.update(1)
165     m2 = vplanets[p0 + i:p1:sl] + k1*dt/2

```

```

162         k2 = aGeneral2(rplanets[p0 + i:p1:s1] + m1*dt/2)
163         pbar.update(1)
164         m3 = vplanets[p0 + i:p1:s1] + k2*dt
165         k3 = aGeneral2(rplanets[p0 + i:p1:s1] + m2*dt)
166         pbar.update(1)
167         vplanets[p0 + i + 1:p1:s1] = vplanets[p0 + i:p1:s1] + (k0 + 2*k1 + 2*k2
            + k3)*dt/6
168         rplanets[p0 + i + 1:p1:s1] = rplanets[p0 + i:p1:s1] + (m0 + 2*m1 + 2*m2
            + m3)*dt/6
169
170     print(np.arange(p0 + forwards, p1, s1)[-1])
171     print(p1)
172     print(steps)
173
174     for i in range(steps - p1 - 1):
175         m0 = vplanets[p1 + i]
176         k0 = aGeneral(rplanets[p1 + i])
177         m1 = vplanets[p1 + i] + k0*dt/2
178         k1 = aGeneral(rplanets[p1 + i] + m0*dt/2)
179         m2 = vplanets[p1 + i] + k1*dt/2
180         k2 = aGeneral(rplanets[p1 + i] + m1*dt/2)
181         m3 = vplanets[p1 + i] + k2*dt/2
182         k3 = aGeneral(rplanets[p1 + i] + m2*dt/2)
183         vplanets[p1 + i + 1] = vplanets[p1 + i] + (k0 + 2*k1 + 2*k2 + k3)*dt/6
184         rplanets[p1 + i + 1] = rplanets[p1 + i] + (m0 + 2*m1 + 2*m2 + m3)*dt/6
185
186     # from vis import *
187     # Earth = rplanets[-100:,3]
188     # lign(Earth, '#3d4782')
189     # plot()
190
191     #Interpolation complete !
192     # with open("rocketdata.npy", "rb") as file:
193     #     r = np.load(file)
194     # print(np.min(mag(r - rplanets[:,target])))
195     # exit(0)
196
197     r = np.empty([steps,3], dtype=np.longdouble)
198     v = np.empty([steps,3], dtype=np.longdouble)
199
200     #DECLARING INITIAL CONDITIONS:
201     r[N2] = rplanets[N2][3] + rinit      #transformation from earth to other
        coordinates!
202     v[N2] = vplanets[N2][3] + vinit
203
204

```

```

205 def fs(rpl, rsat):
206     deltar = rpl - rsat[np.newaxis, :]
207     distanceFactor = (np.sqrt(np.sum(deltar**2, axis=-1))**3)[: , np.newaxis]
208     return np.sum(np.divide(deltar, distanceFactor, out=np.zeros_like(deltar),
        where=np.rint(distanceFactor)!=0)*(G*masses[: , np.newaxis]), axis=0)
209
210 def f(n):
211     deltar = rplanets[n] - r[n][np.newaxis, :]
212     distanceFactor = (np.sqrt(np.sum(deltar**2, axis=-1))**3)[: , np.newaxis]
213     return np.sum(np.divide(deltar, distanceFactor, out=np.zeros_like(deltar),
        where=np.rint(distanceFactor)!=0)*(G*masses[: , np.newaxis]), axis=0)
214
215
216
217 stepli = np.empty([4,2,objcount,3], dtype=np.longdouble)
218
219 dt = -dt
220 for k in range(N2):
221     stepli[0, 0] = vplanets[N2 - k]
222     stepli[0, 1] = aGeneral(rplanets[N2 - k])
223     stepli[1, 0] = vplanets[N2 - k] + stepli[0, 1]*dt/4
224     stepli[1, 1] = aGeneral(rplanets[N2 - k] + stepli[0, 0]*dt/4)
225     stepli[2, 0] = vplanets[N2 - k] + stepli[1, 1]*dt/4
226     stepli[2, 1] = aGeneral(rplanets[N2 - k] + stepli[1, 0]*dt/4)
227     stepli[3, 0] = vplanets[N2 - k] + stepli[2, 1]*dt/2
228     stepvectm = r[N2 - k] + (stepli[0, 0] + stepli[1, 0]*2 + stepli[2, 0]*2 +
        stepli[3, 0])*dt/12
229
230     m0 = v[N2 - k]
231     k0 = fs(rplanets[N2 - k], r[N2 - k])
232     m1 = v[N2 - k] + k0*dt/2
233     k1 = fs(stepvectm, m0*dt/2)
234     m2 = v[N2 - k] + k1*dt/2
235     k2 = fs(stepvectm, m1*dt/2)
236     m3 = v[N2 - k] + k2*dt
237     k3 = fs(rplanets[N2 - k - 1], m2*dt)
238     r[N2 - k - 1] = r[N2 - k] + (m0 + m1*2 + m2*2 + m3)*dt/6
239     v[N2 - k - 1] = v[N2 - k] + (k0 + k1*2 + k2*2 + k3)*dt/6
240
241 dt = -dt
242 for k in range(N2):
243     stepli[0, 0] = vplanets[N2 + k]
244     stepli[0, 1] = aGeneral(rplanets[N2 + k])
245     stepli[1, 0] = vplanets[N2 + k] + stepli[0, 1]*dt/4
246     stepli[1, 1] = aGeneral(rplanets[N2 + k] + stepli[0, 0]*dt/4)
247     stepli[2, 0] = vplanets[N2 + k] + stepli[1, 1]*dt/4

```

```

248     stepli[2, 1] = aGeneral(rplanets[N2 + k] + stepli[1, 0]*dt/4)
249     stepli[3, 0] = vplanets[N2 + k] + stepli[2, 1]*dt/2
250     stepvectm = r[N2 + k] + (stepli[0, 0] + stepli[1, 0]*2 + stepli[2, 0]*2 +
        stepli[3, 0])*dt/12
251
252     m0 = v[N2 + k]
253     k0 = fs(rplanets[N2 + k], r[N2 + k])
254     m1 = v[N2 + k] + k0*dt/2
255     k1 = fs(stepvectm, m0*dt/2)
256     m2 = v[N2 + k] + k1*dt/2
257     k2 = fs(stepvectm, m1*dt/2)
258     m3 = v[N2 + k] + k2*dt
259     k3 = fs(rplanets[N2 + k + 1], m2*dt)
260     r[N2 + k + 1] = r[N2 + k] + (m0 + m1*2 + m2*2 + m3)*dt/6
261     v[N2 + k + 1] = v[N2 + k] + (k0 + k1*2 + k2*2 + k3)*dt/6
262
263
264
265
266 C1s = np.empty([3], dtype=np.longdouble)
267 S0 = np.empty([3], dtype=np.longdouble)
268
269 def resets():
270     global C1s, S0, Sn, sn
271     #defining C1s
272     sum1 = np.array([0,0,0], dtype=np.longdouble)
273     for k in range(N + 1):
274         sum1 += f(k)*b[N2][k]
275     C1s = v[N2]/dt - sum1
276     #Defining S0:
277     sum2 = np.array([0,0,0], dtype=np.longdouble)
278     for k in range(N + 1):
279         sum2 += f(k)*a[N2][k]
280     S0 = r[N2]/dt**2 - sum2
281     sn = C1s
282     Sn = S0
283 resets()
284
285 def getss(n):
286     global Sn, sn
287     if n == N2:
288         resets()
289         return Sn
290     elif -1 < n < N2:
291         resets()
292         for j in range(N2 - n):

```

```

293         Sn = Sn - sn + f(N2 - j)*0.5
294         sn -= (f(N2 - j) + f(N2 - j - 1))*0.5
295     return sn, Sn
296 elif n > N2:
297     resets()
298     for j in range(n - N2):
299         Sn += sn + f(N2 + j)*0.5
300         sn += (f(N2 + j) + f(N2 + j + 1))*0.5
301     return sn, Sn
302
303 def getsr(n):
304     global Sn, sn
305     if n == N + 1:
306         resets()
307         for j in range(n - N2 - 1):
308             if j != 0:
309                 sn += (f(N2 + j - 1) + f(N2 + j))*0.5
310                 Sn += sn + f(N2 + j)*0.5
311
312         sn += (f(n - 2) + f(n - 1))*0.5
313         Sn += sn + f(n - 1)*0.5
314     return sn, Sn
315
316 def getssr(n):
317     global sn
318     return sn + (f(n - 1) + f(n))*0.5
319
320 maxa = 1
321 while maxa > 0.000000000001:
322     maxa = 0
323     for n in range(N + 1):
324         if n != N2:
325             s, S = getss(n)
326             aold = f(n)
327             #correct starting value
328             sum3r = np.array([0,0,0], dtype=np.longdouble)
329             sum3v = np.array([0,0,0], dtype=np.longdouble)
330             for k in range(N + 1):
331                 a3 = f(k)
332                 sum3r += a3*a[n][k]
333                 sum3v += a3*b[n][k]
334             r[n] = (S + sum3r)*dt**2
335             v[n] = (s + sum3v)*dt
336             #check convergence of accelerations
337             anew = f(n)
338             magdif = mag(aold - anew)

```

```

339         if magdif > maxa:
340             maxa = magdif
341
342     #Commencing PEC cycle:
343     n = N
344     t = N2*dt
345
346
347     corrsum = np.empty([2, 3], dtype=np.longdouble)
348     with tqdm(total=(steps - 9)) as pbar:
349         while t <= T:          #T is defined in general_definitions
350             #Predict:
351             s, S = getsr(n + 1)
352             psum = np.array([0,0,0], dtype=np.longdouble)
353             psumv = np.array([0,0,0], dtype=np.longdouble)
354             for k in range(N + 1):
355                 ap = f(n-N+k)
356                 psum += ap*a[N + 1][k]
357                 psumv += ap*b[N + 1][k]
358             r[n + 1] = (psum + S)*dt**2
359             v[n + 1] = (psumv + f(n)/2 + psumv)*dt
360             n += 1
361             corrsum.fill(0)
362             #Evaluate-Correct:
363             for k in range(N):
364                 ac = f(n + k - N)
365                 corrsum[0] += ac*a[N][k]
366                 corrsum[1] += ac*b[N][k]
367
368             for _ in range(200):
369                 max = 0
370                 rold = r[n]
371                 r[n] = (f(n)*a[N][N] + corrsum[0] + S)*dt**2
372                 v[n] = (f(n)*b[N][N] + corrsum[1] + s)*dt
373                 diff = mag(rold - r[n])
374                 if diff > max:
375                     max = diff
376                 if max < 0.0000000001:
377                     break
378             t += dt
379             pbar.update(1)
380     print("Rocket Trajectory calculated!\nSaving to file...")
381
382     with open("rocketdata.npy", "wb") as file:
383         np.save(file, r)
384

```

```
385 print(np.min(mag(r - rplanets[:,target] - refpos[np.newaxis, :])))
```

B Documents supplémentaires

B.1 Tableau des écarts

Le tableau présente les écarts observés entre différentes méthodes de calcul du mouvement des planètes et une référence obtenue à partir du système Horizons. Les notations utilisées sont : « réf. » pour référence, « sim. » pour simulation, et « K » pour la solution au problème de Kepler. Les différences sont évaluées soit en comparant la distance entre deux vecteurs (notée $|\Delta\vec{r}|$ ou $|\Delta\vec{v}|$), soit en comparant la différence de leur norme (notée $\Delta|\vec{r}|$ ou $\Delta|\vec{v}|$). Ces écarts énumérés ci-dessous sont exprimés en pourcentage de la norme du vecteur de référence. Par exemple, pour un vecteur arbitraire $\vec{w}$, l'écart est calculé selon la formule : $\frac{|\vec{w}_1 - \vec{w}_2|}{w_2} \cdot 100\%$ pour $|\Delta\vec{w}|$ ou $\frac{||\vec{w}_1| - |\vec{w}_2||}{w_2} \cdot 100\%$ pour $\Delta|\vec{w}|$. Cette procédure est automatisée avec le programme `test2.py`. Les données de la simulation sont des valeurs qui vont 165 ans dans l'avenir avec un pas d'intégration de 1000 s.

Méthode	Mercure	Vénus	Terre	Mars	Jupiter	Saturne	Uranus	Neptune
Sim. et réf. ($ \Delta \vec{r} $)	0.5787%	0.1407 %	0.07439 %	0.05042 %	0.008333 %	0.003085 %	0.0013 %	0.0006901 %
K. et réf. ($ \Delta \vec{r} $)	0.1784 %	3.113 %	2.542 %	5.826 %	2.024 %	12.26 %	11.32 %	4.464 %
K. et sim. ($ \Delta \vec{r} $)	0.4038 %	2.972 %	2.616 %	5.877 %	2.015 %	12.26 %	11.32 %	4.465 %
Sim. et réf. ($\Delta \vec{r} $)	0.004507 %	0.0002972 %	0.001167 %	0.003238 %	0.0001052 %	$8.443 \cdot 10^{-5}$ %	$5.343 \cdot 10^{-5}$ %	$6.849 \cdot 10^{-6}$ %
K. et réf. ($\Delta \vec{r} $)	0.009216 %	0.007756 %	0.02794 %	0.3035 %	0.1208 %	0.2 %	0.2106 %	0.05903 %
K. et sim. ($\Delta \vec{r} $)	0.004709 %	0.007459 %	0.0291 %	0.3067 %	0.1207 %	0.2001 %	0.2106 %	0.05904 %

TABLE 2 – L'écart de la position après 165 ans

Méthode	Mercure	Vénus	Terre	Mars	Jupiter	Saturne	Uranus	Neptune
Sim. et réf. ($ \Delta \vec{v} $)	0.4742 %	0.1417 %	0.1417 %	0.04703 %	0.007936 %	0.00314 %	0.001338 %	0.0006955 %
K. et réf. ($ \Delta \vec{v} $)	0.07818 %	3.13 %	2.548 %	5.518 %	1.899 %	12.49 %	12.2 %	4.276 %
K. et sim. ($ \Delta \vec{v} $)	0.4077 %	2.972 %	2.988 %	2.623 %	5.565 %	1.891 %	12.5 %	12.2 %
Sim. et réf. ($\Delta \vec{v} $)	0.003728 %	0.0002991 %	0.001177 %	0.003027 %	$9.861 \cdot 10^{-5}$ %	$9.611 \cdot 10^{-5}$ %	$5.615 \cdot 10^{-5}$ %	$1.31 \cdot 10^{-7}$ %
K. et réf. ($\Delta \vec{v} $)	0.00752 %	0.007901 %	0.02747 %	0.2803 %	0.1187 %	0.4186 %	0.391 %	0.09348 %
K. et sim. ($\Delta \vec{v} $)	0.003792 %	0.007602 %	0.02864 %	0.2833 %	0.1186 %	0.4187 %	0.391 %	0.09348 %

TABLE 3 – L'écart de la vitesse après 165 ans

B.2 Choix du périapse minimal

Du fait d'un grand champ magnétique et d'une forte exposition aux radiations, le périapse minimal de Jupiter était déterminé à une distance de 1,1 fois le rayon jovien [50]. Comme un rayon jovien équivaut à 71 492 000 m [51], nous pouvons définir la distance minimale comme étant 78 641 200 m.

Pour Saturne, le périapse minimal a été choisi à la fin de l'anneau E, qui vaut 480 000 000 m. [52]

En considérant que l'atmosphère de Mercure est « essentiellement un vide » [53], elle ne doit pas être prise en compte. J'ai donc déterminé la distance minimale à l'équateur de Mercure, égale à 2 440 500 m.

Pour les autres planètes, nous devons prendre en compte l'atmosphère : D'après la source [54], l'atmosphère commence à avoir un effet en dessous d'une altitude de 122 km. En utilisant le modèle exponentiel par morceaux décrit dans [55], on obtient une densité de $\rho_{122km} = 1.9740 \cdot 10^{-8}$ à cette altitude. J'utiliserai cette valeur comme référence pour déterminer l'endroit où la densité atmosphérique d'une planète devient suffisamment petite pour être négligée.

Si la densité à la surface et la hauteur de l'échelle d'une atmosphère sont connues et que nous faisons l'hypothèse simplifiée que la température de l'atmosphère ainsi que sa composition sont constantes, nous pouvons utiliser la formule barométrique $\rho = \rho_0 \exp(-\frac{h}{H})$ dérivée du livre [56] pour déterminer la hauteur à laquelle l'atmosphère devient négligeable. Les densités de surface et la hauteur de l'échelle sont présentées dans le tableau suivant :

	Vénus [57]	Mars [58]	Uranus [59]	Neptune [60]
Densité à la surface	65. kg/m ³	0.020 kg/m ³	0.42 kg/m ³	0.45 kg/m ³
La hauteur de l'échelle	15.9 km	11.1 km	27.7 km	19.1 - 20.3 km

TABLE 4 – Données atmosphériques

La hauteur h se trouvant donc avec $h = -H(\ln \rho - \ln \rho_0)$, doit être additionnée avec R , qui est le rayon de la planète à l'équateur, pour obtenir $r_{p, \min} = R + h$. En complétant avec les valeurs de Mercure, de la Terre, de Jupiter et de Saturne :

Planète	Mercure	Vénus	Terre	Mars	Jupiter	Saturne	Uranus	Neptune
$r_{p, \min}$	2 440 500 m	6 400 250 m	6 500 137 m	3 549 700 m	78 641 200 m	$4.8 \cdot 10^8$ m	26 026 390 m	25 107 930 m

TABLE 5 – Périapse minimal des planètes

B.3 Considérations pour les masses barycentriques

Le Soleil La masse du Soleil est la plus importante pour calculer les trajectoires de vol des planètes. Celle donnée par [61] a changé fin octobre, ce qui a entraîné un écart élevé dans ma

simulation. Si la lectrice ou le lecteur remarque ce changement, la masse du Soleil, qui est actuellement de $1.988410 \cdot 10^{30}$ kg selon les données actuelles, doit être mise à jour.

Mercure La planète Mercure n'ayant pas de lune [53], a une masse barycentrique de $3.302 \cdot 10^{23}$ kg [61].

Vénus Vénus n'a pas de satellite naturel non plus [57]. La masse du barycentre est donc $4.8685 \cdot 10^{24}$ kg [61].

Terre La masse de la Terre est de $5.97219 \cdot 10^{24}$ kg, et celle de la Lune est de $7.349 \cdot 10^{22}$ kg. La masse totale que nous considérons comme concentrée au barycentre du système terrestre est donc de $6.04568 \cdot 10^{24}$ kg [61].

Mars La planète rouge a les deux satellites : Phobos et Déimos. La masse de la planète elle-même est de $6.4171 \cdot 10^{23}$ kg, celle de Phobos est de $1.08 \cdot 10^{16}$ kg et celle de Déimos est de $1.80 \cdot 10^{15}$ kg [61]. Comme ils n'ont pas d'influence sensible sur les chiffres significatifs de la masse de Mars, ils sont négligés.

Jupiter Jupiter, la planète la plus massive du système solaire, possède 95 satellites naturels [62]. Pour calculer la masse considérée comme concentrée au barycentre de Jupiter, seules les quatre lunes galiléennes ont été prises en compte. Leurs masses sont présentées dans le tableau suivant :

Jupiter [61]	Ganymède [63]	Callisto [63]	Io [63]	Europe [63]
$1.89818722 \cdot 10^{27}$ kg	$1.482 \cdot 10^{23}$ kg	$1.076 \cdot 10^{23}$ kg	$8.93 \cdot 10^{22}$ kg	$4.80 \cdot 10^{22}$ kg

TABLE 6 – La masse de Jupiter avec ses lunes galiléennes

Ce qui donne en totale $1.89858032 \cdot 10^{27}$ kg.

Saturne Pour Saturne, la somme de toutes les lunes majeures de [64] a été additionnée à la masse de la planète qui vaut $5.6834 \cdot 10^{26}$ kg [61]. Cela donne une masse au barycentre de : $5.684805395 \cdot 10^{26}$ kg

Uranus pour Uranus aussi, j'ai seulement pris en considération les lunes majeures [65]. La masse d'Uranus même est tirée de [61] ce qui donne le résultat suivant : $8.6821876 \cdot 10^{25}$ kg.

Neptune La masse considérée comme concentrée au barycentre de Neptune est la masse de Neptune, soit $1.02409 \cdot 10^{26}$ kg [61]. En additionnant cette masse à celle de sa lune la plus massive, Triton, dont la masse est de $2.14 \cdot 10^{22}$ kg [66], nous obtenons un total de $1.02430 \cdot 10^{26}$ kg.