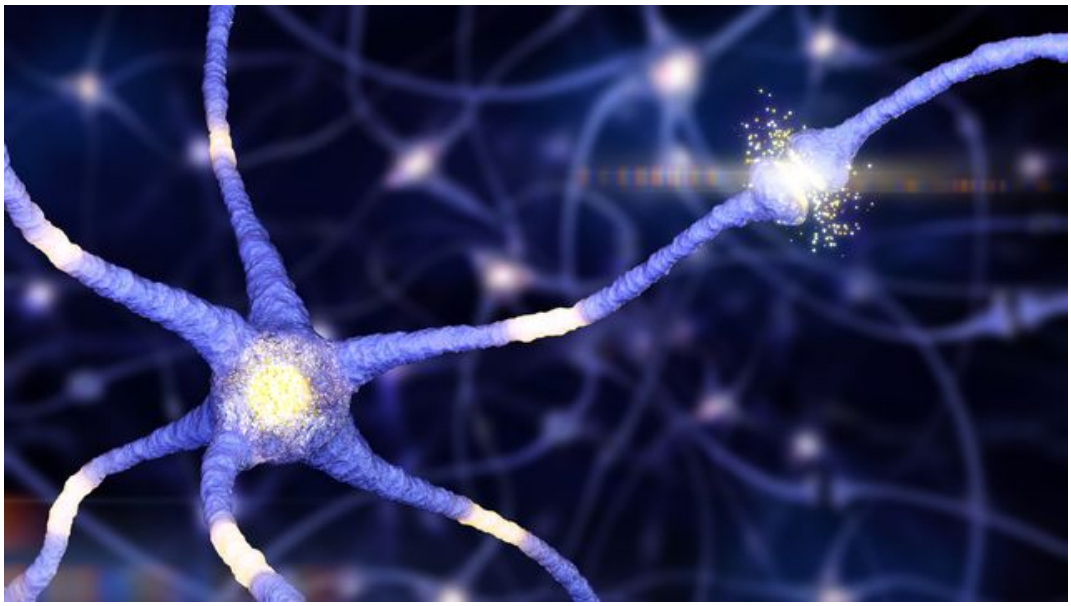
Premier bachelier en sciences de l'ingénieur
Année académique 2017-2018

Projet d'introduction à l'analyse numérique : Étude du comportement du neurone

Victor Delecloz, Nicolas Rotheudt & Tom Winandy

Avril 2018



Introduction

L'objectif principal de ce projet, outre, bien sûr, de nous familiariser avec ce type de travail qui fait partie intégrante de notre cursus en Faculté des Sciences Appliquées et de nous habituer à l'utilisation du logiciel `MatLab`, était d'étudier les réactions d'un neurone à diverses stimulations. Le modèle mathématique modélisant cette activité neuronale (modèle de Fitzhugh-Nagumo) est donné par :

$$\frac{dV}{dt} = V - \frac{V^3}{3} - n^2 + I_{app} \tag{1}$$

$$\frac{dn}{dt} = \epsilon(n_\infty(V) + n_0 - n) \tag{2}$$

avec

$$n_\infty(V) = \frac{2}{1 + e^{-5V}} \tag{3}$$

Ce rapport permet de refléter les conclusions que nous avons pu tirer quant aux possibles réactions d'un neurone. Nous sommes relativement satisfaits de celles-là, car elles nous semblent plutôt complètes et cohérentes.

Ce document se compose de quatre parties majeures, correspondant aux quatre questions, à savoir la création de méthodes de recherche de racines d'une fonction, l'implémentation du système d'équations différentielles ainsi que sa résolution (par la suite, nous utiliserons principalement `ode45` et ce pour des raisons explicitées dans le rapport), l'étude de l'activité électrique nécessaire pour produire une réaction du neurone (où nous avons pu par exemple constater que le neurone ne produit des *spikes* que pour un faible intervalle de valeurs de courant) et enfin l'étude de la réaction du neurone lorsqu'il est soumis à un courant de type "constant par morceaux" (où nous avons pu conclure qu'il existait encore une fois une "intensité seuil", à partir de laquelle il y a production de *spikes*).

Avant de commencer la lecture du rapport, il est important de noter que les codes relatifs au projet n'y sont pas détaillés, mais que beaucoup d'explications se trouvent dans les commentaires de ceux-ci. Afin d'alléger les écritures, dans toute cette présentation, l'abréviation *EDO* signifiera *équation(s) différentielle(s) ordinaire(s)*.

Enfin, pour faciliter l'analyse de notre projet, voici la liste des codes associés à chacune des questions :

- **Question 1.1 :** `bissection.m`, `bissection_bis`
- **Question 1.2 :** `secante.m`, `secante_bis`
- **Question 2.1 :** `nagumo.m`
- **Question 2.2 :** `Euler_explicite.m`, `Euler_test.m`, `intermediaire.m`, `Jacobian_test.m`, `Jacobian_calcul.m`
- **Question 2.3 :** `ode.m`, `test_tolerance_ode.m`
- **Question 2.4 :** `simulation_Q2_4.m`, `comparaison_ode_Euler.m`
- **Question 3.1 :** `simulation_neurone.m`, `cherche_max.m`, `derivee.m`
- **Question 3.2 :** `resultats_Q3_2.m`, `Q3_2_I_min.m`, `frequence_fct_de_I_app.m`
- **Question 4.1 :** `react_neurone.m`, `creation_vecteurs_I_app.m`
- **Question 4.2 :** `transcription_signal.m`, `transcription_signal_robuste.m`,

1 Question 1 : Les méthodes de la sécante et de la bissection

1.1 Question 1.1 : La bissection

Tout d'abord, il est utile de rappeler quelques précisions quant à la méthode de la bissection. Cette méthode nécessite principalement de commencer avec deux valeurs initiales qui ont des images de signes opposés. C'est la raison pour laquelle notre code teste cette condition, tout en demandant à l'utilisateur de saisir de nouvelles valeurs si celles préalablement encodées possèdent des images de même signe. Par ailleurs, la fonction traitée par la bissection doit évidemment être continue pour que la méthode puisse être utilisée. En effet, celle-ci se base sur le théorème des valeurs intermédiaires (se référer à un cours d'analyse pour plus de précisions).

Cette méthode étant basée sur une boucle, il faut en sortir à un certain moment : nous avons donc décidé d'effectuer l'algorithme tant que notre intervalle $[x0, x1]$ excédait une certaine précision (cette tolérance nous sert de gardien de boucle). Cependant, nous aurions très bien pu utiliser le nombre d'itérations comme gardien de la boucle de l'algorithme. En effet, dans la méthode de la bissection, il est assez aisé de déterminer la taille de l'intervalle après un certain nombre d'itérations grâce à la formule suivante :

$$taille_{intervalle} = \frac{|b_0 - a_0|}{2^k} \tag{4}$$

où k est le nombre d'itérations (initialement nul, sans quoi le dénominateur est 2^{k-1}, mais cela ne change pas grand chose à l'essence des formules (4) à (6)) et où a_0 et b_0 sont les valeurs initiales.

Ainsi, puisque nous désirions une précision ϵ de 10^{-7} sur la solution (la raison du choix de cette valeur sera explicitée lors du détail de la question 3.2.), il est naturel d'avoir fixé la tolérance de notre code à 2×10^{-7} (ce qui correspond à la taille finale de $[x0, x1]$), de sorte que le résultat déterminé par notre algorithme ne s'écartera alors pas de plus de un ϵ du résultat analytique. En effet :

$$|x_k - \bar{x}| \leq \frac{1}{2}\frac{|b_0 - a_0|}{2^k} \leq \epsilon \tag{5}$$

où x_k est une valeur obtenue numériquement après k itérations et $\bar{x}$ est la réponse analytique.

Par ailleurs, il nous aurait été difficile d'imposer une tolérance beaucoup plus basse sans que la soustraction $x_0 - x_1$ ne devienne dangereuse, ceci étant dû à l'existence de l'epsilon machine ϵ_M. 10^{-7} est donc une valeur raisonnable. En effet, la représentation des nombres en virgule flottante se fait sur une mantisse de taille finie. Une précision infinie est donc impossible car la plus petite différence entre deux nombres représentables (en format *IEEE754* double précision) est $2^{-52} \simeq 2,22{*}10^{-16}$ (c'est d'ailleurs la définition de l'ϵ_M).

En travaillant sur l'équation (4), nous aurions donc pu déterminer le nombre d'itérations à effectuer pour atteindre cette précision ϵ.

$$k \geq \frac{\ln\left(\frac{b-a}{2\epsilon}\right)}{\ln 2} \tag{6}$$

Enfin, la méthode de la bissection a un taux de convergence d'ordre 1 ou linéaire ce qui est plutôt lent. Cependant, la méthode est très robuste, ce qui en fait sa force. En effet, à partir du moment où les images des valeurs initiales sont bien de signes opposés et que la fonction est continue, il y aura systématiquement convergence de la méthode vers le résultat analytique recherché.

1.2 Question 1.2 : La sécante

Premièrement, la méthode de la sécante ne nécessite que très peu d'hypothèses de départ : il faut principalement que les images des deux valeurs initiales par la fonction soient distinctes. C'est la raison pour

laquelle, si celles-ci sont trop proches, nous demandons à l'utilisateur d'en saisir de nouvelles. En effet, on ne peut vérifier l'égalité stricte de deux nombres réels, en raison des erreurs introduites par l'ϵ_M. Il vaut donc mieux tester si elles diffèrent d'une certaine quantité (par exemple 10^{-7}).

Ensuite, la force de cette méthode est sa vitesse de convergence mais son plus gros problème est son manque de robustesse. L'ordre de convergence de la méthode de la sécante étant de $1+\frac{\sqrt{5}}{2} \simeq 1.618$ (ordre de convergence superlinéaire), la racine recherchée est rarement trouvée après plus d'une vingtaine d'itérations, voire 30 ou 40 si les valeurs initiales sont vraiment mal choisies. Nous avons donc décidé qu'après 50 itérations (ce nombre nous sert donc de garde-fou pour la boucle principale de la méthode), il valait mieux saisir de nouvelles conditions initiales, et ainsi éviter une boucle infinie. En effet, si un tel nombre est atteint, la recherche de la racine par la méthode de la sécante devient fort inefficace, et cela peut traduire une convergence vers un "pseudo zéro". (cfr. Figure 1 : alors que nous recherchions la racine en $x = 0.1$, la méthode de la sécante peut nous conduire à rechercher la "racine" vers laquelle la fonction tend "à l'infini"). Ainsi, pour une recherche de racine plus incertaine, on se gardera de vouloir profiter de l'avantage de cette méthode, à savoir sa rapidité.

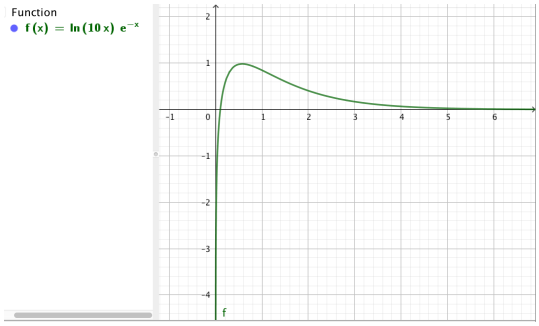


FIGURE 1 – Exemple de mauvaise convergence de la méthode de la sécante.

Enfin, de manière similaire à la méthode de la bissection, nous avons fixé une tolérance par rapport à la racine à rechercher (également à 10^{-7} donc) puisque, pour des raisons déjà détaillées, nous ne pouvions tester explicitement l'égalité de l'ordonnée et de zéro (présence de ϵ_M).

NB : Autant pour la sécante que la bissection, on aurait très bien pu passer la tolérance par rapport au résultat en paramètre de la fonction. Ceci aurait pu encore en augmenter la flexibilité. Nous avons également fait le choix de manipuler les images de la fonction mathématique dans des variables spécifiques et non d'appeler systématiquement cette fonction. En effet, cela permet un important gain de temps, surtout lorsque ces appels sont coûteux (on évite de devoir évaluer la fonction à chaque fois qu'on manipule l'image d'une valeur). Enfin, il existe une version plus "robuste" de ces méthodes (il s'agit du même code pour les deux), respectivement `bissection_bis` et `secante_bis`, qui, bien que loin d'être parfaite, a le mérite d'être un peu plus robuste. Partant de l'algorithme de la sécante, cette version teste à chaque itération si elle peut trouver la solution de manière certaine, c'est-à-dire en utilisant la bissection. Ainsi, la méthode essaye par elle-même d'assurer sa convergence (à défaut de sa rapidité). Le code étant largement commenté, il suffit de s'y référer pour obtenir plus de détails.

2 Question 2 : L'implémentation et la résolution du système d'EDO

2.1 Question 2.1 : L'implémentation : `nagumo`

Il n'y a pas grand chose à expliciter sur la fonction `nagumo`. Il s'agit simplement de l'implémentation du système d'EDO. Il est juste à noter que nous avons utilisé la fonction `interp1` dans le but d'avoir une plus

grande flexibilité pour la question 4 (cfr. section 4 pour plus d'informations au sujet de cette fonction).

2.2 Question 2.2 : Une méthode de résolution : `Euler explicite`

Afin de déterminer un pas adapté pour cette méthode, nous avons décidé de nous en référer à la théorie. En effet, dans le cas de la résolution d'une EDO, il est important de choisir un pas tel que l'erreur, qui se propage d'une itération à l'autre, diminue au lieu d'augmenter. Dans ce cas, la méthode est dite *stable*, ce qui est le cas si

$$-2 < hJ_i < 0 \quad \forall i \tag{7}$$

où h est le pas et J_i le jacobien (la matrice jacobienne se réduit à un simple scalaire).

On remarque que la méthode d'Euler explicite n'est jamais stable si l'équation différentielle n'est elle-même pas stable ($J_i > 0$). Dans le cas d'un système d'EDO, ce sont les parties réelles des valeurs propres de la matrice jacobienne qui doivent vérifier la condition donnée par (7).

De manière analytique, mais aussi à l'aide de `MatLab` (cfr. codes `Jacobian_calcul` et `Jacobian_test`), nous avons calculé la matrice jacobienne associée au système d'EDO. Nous avons pu ainsi estimer l'évolution de ses valeurs propres au cours des itérations et, par conséquent, déterminer un pas qui, théoriquement serait adéquat. L'examen systématique de ces valeurs propres nous a permis de constater que toutes s'inscrivaient dans un intervalle de l'ordre de $[-2.8143; -0.0048]$ (ceci dépendant fortement du courant appliqué, il s'agit d'une approche plus qualitative que réellement quantitative ; les valeurs données ici correspondant à un courant de 1 (l'intervalle est plus petit pour un courant de 0.1 par exemple)).

Puisqu'on doit vérifier $|hJ_i| < 2$, on en déduit que $h < 0.71 \sim 0.7$. Pour une valeur de h inférieure à $0.7\ ms$, la méthode est donc censée être "fonctionnelle".

Ce pas est donc la valeur théorique pour laquelle la méthode est stable et pourrait donner une réponse satisfaisante. Cependant rien ne garantit que l'erreur, bien que décroissante, soit négligeable ! En effet, en pratique, on constate aisément à l'aide de divers tests qu'un pas de $0.05\ ms$ serait plus adapté. Comme le montre la Figure 2 (cfr. code `Euler_test`), pour des pas plus importants, l'erreur devient vite très importante et la méthode en perd toute fiabilité. Par exemple, après 1 ms d'exécution, la valeur obtenue pour V à l'aide d'un pas de $0.5\ ms$ est de $-1,79408$ alors que la valeur obtenue avec un pas de $0.05\ ms$ est de -1.75326, soit une erreur relative de 2.33%. De même, après $0.95\ ms$ d'exécution, la valeur obtenue pour V à l'aide d'un pas de $0.95\ ms$ est de -1.99875 alors que la valeur obtenue avec un pas de $0.05\ ms$ est de -1.74972, soit une erreur relative de 14.23% (pourtant, la méthode est censée être stable même pour ce pas lorsque le courant est de ~ 0.1, mais l'erreur est loin d'être négligeable) !

En outre, bien que le pas choisi soit relativement court (entraînant une perte de rapidité du programme), cela ne porte pas à conséquence : nous ne réutiliserons en effet pas le script `Euler_explicite` lors de ce projet.

Après ces considérations théoriques, il nous a suffi d'implémenter l'algorithme d'Euler explicite en lui-même comme décrit dans le cours théorique.

Il est à noter que les premières lignes du script `Euler_explicite` sont issues de nos recherches quant à l'amélioration de l'aspect de nos graphiques (et seront réutilisées à chaque script nécessitant l'affichage de graphiques). Elles n'ont aucune vraie incidence et, tout comme le `subplot`, ne sont là que pour le côté "esthétique". Si `Euler_explicite` avait été réutilisé de nombreuses fois dans le reste du projet, nous aurions retiré ces artifices car ils ne sont pas indispensables et ralentissent l'exécution (c'est d'ailleurs le cas pour ode45).

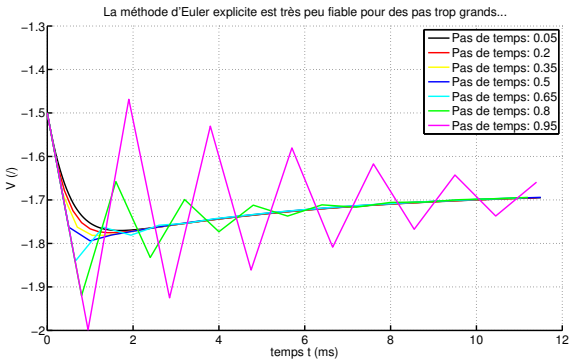


FIGURE 2 – La méthode d'Euler explicite pour différents pas de temps.

Ceci nous a donc permis d'obtenir des graphiques de V et n en fonction du temps t. C'est la cas, par exemple, pour $V_{initial} = -1.5$, $n_{initial} = 0.5$, $I_{app} = 0.1$, $temps_{simulation} = 200$ (cfr. Figure 3), où il est évident que le neurone ne produit aucun *spike* (valeur de I_{app} trop faible), ce qui concorde avec les résultats obtenus dans d'autres codes de ce projet (cfr. section 3 par exemple).

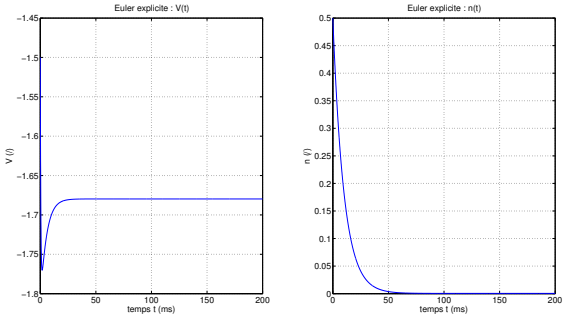


FIGURE 3 – Résultat fourni par la méthode d'Euler explicite.

2.3 Question 2.3 : Une seconde méthode : `ode45`

La fonction `ode45` de `Matlab` correspond à la méthode de Runge-Kutta-Fehlberg classique d'ordre 4 avec un pas adaptatif (ou plutôt de Dormand-Prince, mais les deux sont similaires). Elle permet donc une précision bien plus importante qu'Euler explicite de manière générale. L'inconvénient est qu'elle nécessite un grand nombre d'appels à la fonction (ici : `nagumo`). Cependant, et ceci constitue déjà une des raisons pour laquelle nous privilégierons cette méthode dans la suite du projet, les appels à la fonction `nagumo` ne sont pas trop coûteux et la perte de temps n'est donc pas trop importante.

Nous avons décidé de fixer les erreurs absolues respectivement sur V et sur n à 10^{-7} et 10^{-4} et l'erreur relative à 10^{-7}. Nous avons été plus sévères sur les valeurs de V pour la simple raison que ce sont celles qui nous intéresseront le plus pour la suite du projet (la tolérance sur n n'a aucune vraie incidence). Généralement, l'erreur relative a plus de sens d'un point de vue physique et informatique, mais étant donné l'ordre de grandeur les valeurs considérées, cela importe peu. Pour rappel, si $\tilde{a}$ désigne la valeur approchée d'une quantité dont la valeur exacte est a. On définit :

$$\text{L'erreur absolue : la } \textit{quantité} \quad \tilde{a} - a \tag{8}$$

$$\text{L'erreur relative : la } \textit{quantité} \quad \frac{\tilde{a} - a}{a} \tag{9}$$

Pour choisir ces tolérances, nous avons effectué plusieurs tests et avons comparé nos résultats avec ceux d'`Euler_explicite`. Nous nous sommes dès lors aisément rendus compte qu'à partir d'un moment, pour une même tolérance exigée (qualitativement), le pas que nous devrions fixer pour `Euler_explicite` serait excessivement petit, et donc pas forcément plus rapide qu'`ode45`, et ce malgré que cette dernière méthode nécessite 6 évaluations de fonction à chaque itération ! Nous avons dès lors étudié (cfr. code `test_tolerance_ode`) comment évoluait le temps d'exécution de la fonction d'`ode45` en fonction de la tolérance exigée et avons remarqué que celui-ci explosait après 10^{-7} (cfr. Figure 4 ; il est à noter que la première valeur, i.e. 10^{-1} n'est pas représentative (initialisations etc.) et peut donc être négligée). Ainsi, en ayant choisi cette valeur, nous réussissons à obtenir des valeurs plus précises sans trop allonger le temps d'exécution (en comparaison avec `Euler_explicite`) !

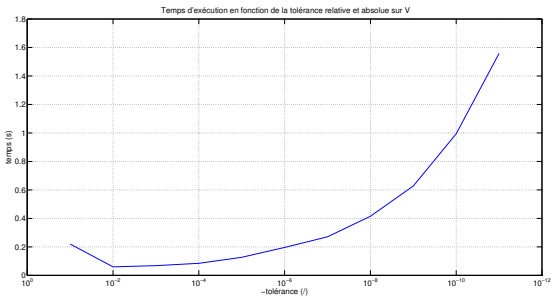


FIGURE 4 – Temps d'exécution d'`ode45` (en s) en fonction de la tolérance relative et absolue sur V

`ode45` nous a permis d'obtenir des graphiques de V et n en fonction du temps t, par exemple pour $V_{initial} = -1.5$, $n_{initial} = 0.5$, $I_{app} = 1$, $temps_{simulation} = 200$, cette fonctionnalité d'affichage de graphiques ayant ensuite été supprimée pour des raisons d'efficacité (cfr. Figure 5). Il est évident que le neurone produit des *spikes* réguliers pour cette valeur de I_{app}, ce qui concorde avec les résultats obtenus dans d'autres codes de ce projet (cfr. section 3 par exemple).

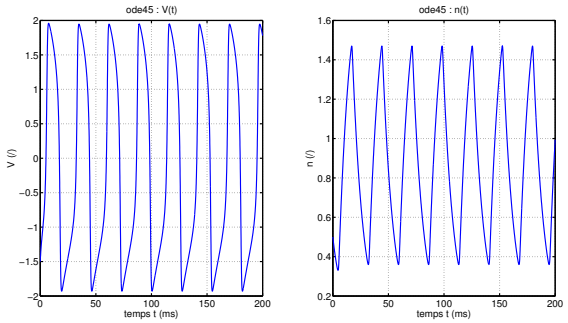


FIGURE 5 – Résultat fourni par la méthode de Runge-Kutta-Fehlberg classique d'ordre 4 (`ode45`).

2.4 Question 2.4 : L'application

La théorie et nos tests nous montrent que la méthode d'`ode45` est bien plus précise que la méthode d'`Euler_explicite`. En effet, dans le premier cas, le terme d'erreur est $O(h^5)$, alors qu'il est $O(h^2)$ dans le second, où h représente le pas (les deux méthodes sont basées sur un développement de Taylor). Ainsi, pour un temps d'exécution globalement similaire, il est plus intéressant d'utiliser `ode45` qu'`Euler_explicite` (comme détaillé en sous-section 2.3). C'est encore une raison pour laquelle le reste de ce projet, tout comme cette question (cfr. code `simulation_Q2_4`), utilise `ode45` comme outil de résolution des EDO. La Figure 6 illustre le fait qu'au fil des itérations, la réponse fournie par `Euler_explicite` s'éloigne de celle fournie par `ode45` (cfr. code `comparaison_ode_Euler`).

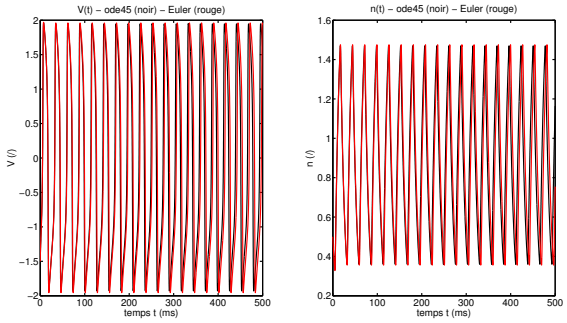


FIGURE 6 – Écart croissant entre Euler explicite et `ode45` au fil des itérations.

3 Question 3

3.1 Question 3.1 : Détermination de la fréquence de production de *spikes*

Notre recherche de la fréquence de production de *spikes* par le neurone due à un courant d'entrée s'est déroulée en plusieurs étapes (cfr. code `simulation_neurone`).

Dans un premier temps, il était, forcément, nécessaire d'obtenir les valeurs de $V(t)$, ce que nous avons fait grâce à `ode45`, pour des raisons déjà explicitées (cfr. section 2).

Ayant décidé de baser notre calcul de la fréquence sur l'écart entre deux maxima successifs, i.e. la "période" de production de *spikes*, il nous a ensuite fallu implémenter une fonction remplissant cette tâche (cfr. code `cherche_max`). Celle-ci étant largement expliquée dans les commentaires du code, nous n'en donnons ici que l'idée générale. `cherche_max` prend en entrée des données de V et de temps, et renvoie les maxima ainsi que les instants auxquels ceux-ci se produisent. La recherche de maxima est basée sur le calcul de la dérivée (cfr. code `derivee`) du vecteur de données par application des règles de l'algèbre discrète. La méthode la plus précise reste la "différence finie centrée", mais pour des raisons de fonctionnalité des codes, nous avons travaillé avec la dérivée "classique". Pour rappel,

$$\text{La différence finie centrée : } f'(x) = \frac{f(x+h) - f(x-h)}{2h} \tag{10}$$

$$\text{La dérivée "classique" : } f'(x) = \frac{f(x+h) - f(x)}{h} \tag{11}$$

En calculant ensuite la dérivée du signe de cette première dérivée (ce qui nous intéressait était en effet de localiser les points où celle-ci changeait de signe, et ce sans être gêné par les éventuelles imprécisions), nous

avons pu déterminer les endroits où les données considérées admettaient des maxima et minima. Cependant, nous avons défini une variable de protection, i.e. `seuil_min`, imposant que la différence entre un maximum et les minima qui l'entourent (s'ils existent), doit être supérieure à `seuil_min`. En effet, il arrive très souvent que, par accumulation d'approximations, une fonction qui, aux premiers abords, semble parfaitement constante, soit en fait caractérisée par des valeurs oscillant continuellement entre deux valeurs distinctes d'à peine $\sim 10^{-7}$ (micro-oscillations). Les exemples de micro-oscillations sont nombreux et ce `seuil_min` est donc essentiel tout au long de ces dernières questions du projet.

Par ailleurs, à partir d'une certaine intensité, le premier des *spikes* produits est plus important que les autres et est fortement irrégulier (le phénomène s'accentue lorsque I_{app} augmente). Dès lors, dès que l'intensité excède 1, nous ignorons le premier maximum (nous voyons en effet déjà sur la Figure 5 que le premier *spike* est légèrement plus haut que les autres).

Enfin, si les maxima (i.e. les *spikes*) sont suffisamment réguliers, ceci étant vérifié par l'écart-type sur les différentes fréquences (nous tolérons un écart-type de 5% par rapport à la valeur moyenne, ce qui nous parait raisonnable), nous pouvons calculer la fréquence comme la moyenne des fréquences calculables entre deux pics successifs. Dans la négative, on considère que la fréquence est nulle.

La Figure 7 illustre un résultat ayant été obtenu par l'exécution de `simulation_neurone` pour toutes les intensités de courant appliqué au neurone entre 0.001 et 4, par pas de 0.001 (cfr. code `frequence_fct_de_i_app`). Ce résultat semble tout à fait logique par rapport aux divers résultats ayant été observés par l'application de `ode45` et `Euler_explicite`.

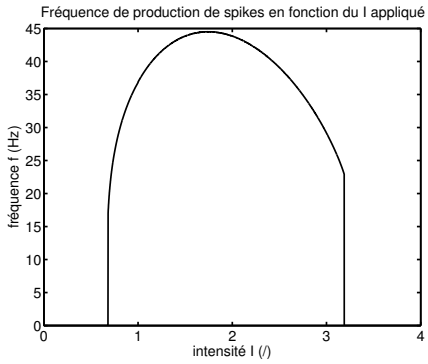


FIGURE 7 – Fréquence de production de *spikes* du neurone en fonction du courant appliqué.

3.2 Question 3.2 : Recherche du courant correspondant à une certaine fréquence

Nous avons utilisé la méthode de la bissection pour effectuer cette recherche. En effet, comme explicité en sous-section 1.1, cette méthode est particulièrement robuste et, pour autant que les valeurs initiales soient judicieusement choisies, elle nous fournira la solution (il est à noter que, au vu de la Figure 7, la fonction est bien continue). Dans ce cas, la fonction traitée par la bissection est `nagumo - freq`, où `freq` est la fréquence vers laquelle on souhaite converger. En effet, si

$$f_k - freq \to 0 \qquad \Leftrightarrow \qquad f_k \to freq \qquad \text{, comme souhaité.} \tag{12}$$

Pour le cas particulier de `freq = 0`, la recherche convergera vers la valeur minimale de courant à appliquer pour percevoir une réaction du neurone (ceci est dû à la manière dont nous avons implémenté la bissection, qui peut traiter le cas où la fréquence serait nulle (et donc où la fonction ne serait jamais négative)).

Il a ensuite fallu déterminer les valeurs initiales adéquates. Lors de l'examen des valeurs renvoyées par `frequence_fct_de_i_app` (grâce à la commande Matlab `find(max(A) == A)`, où `A` désigne le vecteur de fréquences), nous avons pu nous apercevoir que la fréquence maximale pouvant être atteinte était de 43.5103000 à une erreur $O(10^{-7})$ près, et ce pour un courant appliqué de 1.735. Par ailleurs, nous avons pu remarquer que la fréquence pour un courant de 0.1 est nulle (cfr. Figure 3 et Figure 7). Il a dès lors suffi de placer comme valeurs initiales de la bissection celle menant à la fréquence produite maximale et une menant à une fréquence nulle, de sorte que peu importe la fréquence demandée, pour autant qu'elle soit inférieure à celle qui est maximale, i.e. $43.5103000 \simeq 43.5103$, la méthode permettra de déterminer l'intensité du courant correspondant.

Les appels à la fonction `simulation_neurone` sont relativement coûteux, de sorte que notre tolérance ne devait pas être trop importante pour que le temps d'exécution du code reste raisonnable. En outre, cela n'avait pas de sens d'imposer une tolérance plus sévère que 10^{-7} puisque les valeurs calculées au préalable par `ode45` ne sont pas plus précises ! De toute manière, une précision de 10^{-7} nous suffit largement dans ce contexte (recherche d'une intensité à appliquer, qui ne pourrait en pratique que rarement atteindre cette précision). Il est cependant à noter que, malgré ce choix, le temps d'exécution de la fonction `Q3_2_I_min` reste assez important, mais une étude du `profile` de celle-ci nous apprend que cela est majoritairement dû à `interp1`, qui pourrait donc être retiré de `nagumo` pour améliorer l'efficacité du code, et ce sans conséquence puisque jusque maintenant, le courant appliqué est constant !

Intensité en fonction de la fréquence	
Fréquence	Intensité
0	0,6812851
4	0,6812851
8	0,6812851
12	0,6812851
16	0,6812851
20	0,6951348
22	0,7080875
24	0,7251333
26	0,7471994
28	0,7750218
30	0,8094991
32	0,8519294
34	0,9043595
36	0,9688336
38	1,0456099
40	1,1461687
42	1,2843410
44	1.5309966

TABLE 1 – Résultats de `simulation_neurone`

Ainsi, cette fonction nous a permis de déterminer les valeurs de courant menant à différentes fréquences, comme illustré dans le Tableau 1. On notera deux choses pour terminer. Premièrement, ces valeurs concordent parfaitement avec le graphique obtenu en Figure 7, ce qui est tout à fait normal. Deuxièmement, on remarque que chaque fréquence peut être obtenue à l'aide de deux intensités distinctes (exception faite de la fréquence maximale), de sorte que pour atteindre l'autre valeur (supérieure à $I = 1.735$), il suffit de remplacer 0.1 par une autre valeur menant à une fréquence nulle, 3.5 par exemple.

4 Question 4

4.1 Question 4.1 : Réaction du neurone à un courant constant par morceaux

Cette question est sensiblement similaire à la question 2. En effet, la seule différence réside dans le fait que le courant appliqué n'est plus constant (il est "constant par morceaux"). Pour étudier ce phénomène, nous avons utilisé, comme mentionné en sous-section 2.1, `interp1` qui, par défaut, est une interpolation linéaire, et peut donc facilement traduire ce type de fonction "constante par morceaux". En outre, en raison de la nature de celle-ci, nous ne rencontrons pas de problème sur la qualité de l'interpolation, même si elle est simplement linéaire !

Comme à chaque fois dans ce projet, nous avons ensuite naturellement décidé d'utiliser `ode45` pour traduire la réaction du neurone (cfr. code `react_neurone`). Il est à noter que nous avons utilisé la fonction `creation_vecteurs_I_app` pour rendre le vecteur intensité, initialement donné par une suite de courants appliqués, "utilisable" pour l'interpolation (s'en référer au code pour plus d'informations).

La Figure 8 illustre le résultat obtenu pour une intensité appliquée de la manière décrite par le graphique de droite. Celui-ci est une nouvelle fois parfaitement cohérent avec les résultats préalablement établis, i.e. pas de réaction notable du neurone pour un I_{app} faible (0.6), mais bien pour un I_{app} supérieur à un certain seuil (0.75, 1.2), à savoir le même seuil que celui à partir duquel sont produits des *spikes* (cfr. résultats de la section 3).

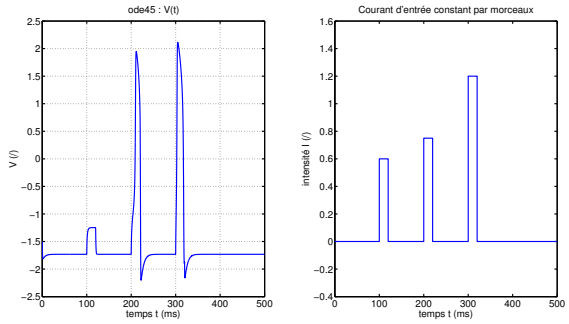


FIGURE 8 – Réaction du neurone à une application de courant non constant.

4.2 Question 4.2 : Transcription du signal généré par le neurone

Nous avons adopté deux approches différentes pour répondre à cette dernière question.

La première, la plus "attendue", consiste à prendre en entrée une suite de courants appliqués, une période (écart temporel séparant deux applications de courants), et une durée d'impulsion (durée pendant laquelle l'application est réalisée), ainsi que, bien entendu, les initialisations de V et n (cfr. code `transcription_signal`). Puisque nous partions de la supposition que les courants sont appliqués de manière périodique, il nous a donc suffi de vérifier si la fonction produisait des maxima locaux (grâce à `cherche_max`) aux instants où un courant était censé être appliqué, et ci ceux-là étaient notables, i.e. supérieurs à 0 (d'après les termes de l'énoncé). La fonction renvoie donc un vecteur dont la taille est égale au nombre de courants appliqués, et associant un 0 ou un 1 à chacun d'entre eux, respectivement si un *spike* notable est produit ou non. Il est à noter que ce sont bien les valeurs de V que nous analysons dans cette fonction et non directement les valeurs d'intensité données en entrée (ce qui n'aurait pas grande utilité). La seule raison pour laquelle nous n'avons pas décidé de passer les valeurs de V en paramètre de la fonction est qu'il aurait alors fallu

préalablement effectuer une autre opération pour les obtenir, ce qui n'est pas très élégant.

La seconde a pour but d'être un peu plus robuste (cfr. code `transcription_signal_robuste`). En effet, elle est capable de fournir une réponse dans la majorité des cas, même si les durées d'application de courant et l'intervalle de temps séparant deux applications successives sont variables. La majorité des cas seulement, car il est nécessaire de respecter certaines conditions pour que la méthode fonctionne, dont la plus restrictive est sans doute que la durée du plus grand intervalle de temps durant lequel un courant est appliqué doit être strictement inférieure à la plus courte des périodes séparant deux applications. En outre, l'appel à cette fonction est un peu plus lourd que `transcription_signal` puisque, les intervalles étant variables, les vecteurs `temps_I_app` et `I_app` doivent être entièrement explicités (et sous une syntaxe légèrement différente de la première version). Enfin, cette fonction ne renvoie une valeur (0 ou 1) que pour chaque courant non nul ayant été appliqué. Cette deuxième version, bien que loin d'être parfaite, avait donc essentiellement pour objectif de réussir à traiter certains cas que la première ne pouvait traiter.

Le Tableau 2 donne quelques exemples de lignes de commande appelant les deux fonctions décrites ci-dessus et les résultats fournis. Il est simplement à noter que ceux-ci sont cohérents avec les graphiques que nous fournissait `react_neurone`.

Quelques exemples d'appels de ces fonctions et leurs résultats	
Appels à la fonction	**Résultats**
`transcription_signal(-1.5, 0.5, 50, 200, [0 0 1])`	001
`transcription_signal(-1.5, 0.5, 20, 150, [3 0 1])`	101
`transcription_signal(-1.5, 0.5, 30, 120, [1.2 0.6 0.9 1 0.1])`	10110
`transcription_signal(-1.5, 0.5, 10, 40, 0)`	0
`transcription_signal_robuste(-1.5, 0.5, [0;0; 0.6;0.6; 0;0; 1.2;1.2; 0;0], [0;99.9999;100;129.9999; 130;189.9999;190;239.9999; 240;350])`	01
`transcription_signal_robuste(-1.5, 0.5, [0,0], [0 10])`	Empty matrix : 0-by-1

TABLE 2 – Résultats de `transcription_signal` et `transcription_signal_robuste`

Conclusion

En guise de courte conclusion, nous rappellerons simplement que nous sommes relativement satisfaits du travail accompli, tant au niveau codes qu'au niveau clarté de ce rapport. Ce projet constitue le premier pas vers quelques caractéristiques de notre futur métier (du moins nous l'espérons), à savoir le travail en équipe, le respect de *deadlines* ou encore le *self-learning* (en ce qui concerne l'utilisation de `MatLab`). Il nous a donc permis de nous rendre compte de l'importance de celles-ci, et nous espérons que l'ensemble des projets qui jalonneront nos années d'étude en Faculté des Sciences Appliquées contribueront de la même manière à l'acquisition de ces compétences essentielles.

Table des matières

Table des figures

Liste des tableaux

Références

- Hardy Joris, élève moniteur.
- Louveaux Quentin, *Introduction aux Méthodes Numériques*, 2018.
- MatLab, *help MatLab*.