LIÈGE université

PROJ0001-1
Introduction aux méthodes numériques et projet

MODELISATION DE L'ACTIVITE NEURONALE

Lione Maxime
Valentiny Damien
Siboyabasore Cédric

Premier Bachelier en sciences de l'ingénieur

Année académique 2017-2018

Table des matières

1. Implémentation des fonctions bissection et sécante

Tout d'abord, avec pour objectif de trouver une racine d'une fonction, nous avons implémenté deux méthodes numériques différentes.

1.1 Méthode de la bissection

Premièrement, nous avons implémenté la méthode de la bissection (ou dichotomie). Cette fonction *bissection*.m prend en entrée une fonction continue f, dont on cherche une racine, et deux points a_0 et b_0 de f. Ces points doivent avoir leurs ordonnées de signes opposés de sorte qu'il existe forcément une racine dans l'intervalle $[a_0, b_0]$.

En suivant l'algorithme de cette méthode, nous avons d'abord défini une erreur ϵ qui correspond à la différence entre l'axe des abscisses et la courbe de f. Nous avons fixé cette erreur à 10^{-7} . Elle nous assure une réponse avec 7 décimales correctes, ce qui nous semblait, après plusieurs essais, être le bon équilibre entre précision et rapidité. Ensuite, nous avons créé une boucle qui calcule la suite des points $x_i = \frac{a_{i-1}+ b_{i-1}}{2}$ et qui réduit l'intervalle de la manière suivante : avec $f(a_0) < 0$ et $f(b_0) > 0$, si $f(x_i) < 0$, alors $a_i = x_i$ et $b_i = b_{i-1}$. Si, par contre, $f(x_i) > 0$, alors $a_i = a_{i-1}$ et $b_i = x_i$. La suite des x_i converge ainsi vers la racine cherchée. Le gardien de boucle k ayant été initialisé à $\frac{\ln(\frac{b_0-a_0}{2\epsilon})}{\ln 2}$ au début de la fonction, il assure que la valeur absolue de la différence entre le x_k trouvé à la fin de la boucle et la racine cherchée est inférieure à l'erreur ϵ[1]. La fonction retourne donc finalement ce x_k, qui est compris entre $[a_0, b_0]$.

De plus, pour éviter d'entrer dans la boucle inutilement, la fonction que nous avons implémentée cherche d'abord si un des points entrés en paramètre n'est pas une racine (= si la valeur absolue de son ordonnée est supérieure à ϵ). Si c'est le cas, elle le renvoie directement et son exécution s'arrête.

Enfin, nous nous protégeons aussi du cas où l'hypothèse que les points en entrée aient des ordonnées de signes opposés ne soit pas vérifiée (en vérifiant si le produit des ordonnées est supérieur à 0). Si c'est le cas, au lieu d'arrêter tout de suite l'exécution de la fonction et de renvoyer un message d'erreur, elle fait d'abord appel à la fonction sécante, expliquée au point suivant, qui prend en entrée les mêmes paramètres. Sa robustesse est ainsi renforcée.

[1] Vu l'analyse de la méthode de la bissection faite au cours théorique : Soit $\mathbb{x}$ la racine recherchée et $[a, b]$ l'intervalle de départ, on a, à l'itération k, un intervalle de taille $\frac{b-a}{2^k}$. Puisqu'on veut $|x_k - \mathbb{x}| \leq \epsilon$, on a $|x_k - \mathbb{x}| \leq \frac{1}{2}\frac{b-a}{2^k} \leq \epsilon$, donc $k \geq \frac{\ln(\frac{b-a}{2\epsilon})}{\ln 2}$.

1.2 Méthode de la sécante

La seconde fonction que nous avons implémentée pour la recherche d'une racine est appelée *secante*.m et prend donc les mêmes paramètres d'entrée que la bissection. Cependant, ici, nous n'avons plus besoin de faire l'hypothèse que les points en entrée a_0 et b_0 de f ont des ordonnées de même signe. En effet, l'algorithme de cette méthode effectue une recherche d'une racine de f en l'approximant par la droite qui relie les deux points x_{i-1} et x_i (avec initialement $a_0 = x_0$ et $b_0 = x_1$).

Dans notre fonction, ces points successifs sont calculés dans une boucle par la formule suivante : $x_{i+1} = x_i - \frac{f(x_i)(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})}$ *(1)* où x_{i+1} est la racine de la droite joignant x_{i-1} et x_i. Une variable appelée *erreur* (initialisée arbitrairement à 1 au départ) prend, à chaque itération, la valeur de la valeur absolue de l'abscisse du nouveau point x_{i+1}. A l'itération suivante, le x_{i+1} est calculé seulement si la valeur de *erreur* est plus grande que 10^{-7} (pour les mêmes raisons que la bissection). Si la condition n'est pas remplie, alors la fonction retourne x_i, qui correspond à la racine de la fonction à une tolérance de 10^{-7} près.

De plus, de la même manière que pour la méthode de la bissection, on vérifie si un des points entrés en paramètre n'est pas une racine. Mais ici, on vérifie aussi pour chaque itération que le dénominateur de *(1)* n'est pas égal à 0, ce qui correspondrait à 2 points de même ordonné, reliés donc par une droite parallèle à l'axe des abscisses qui ne pourrait ainsi jamais l'intersecter. Si cette condition est respectée, la fonction renvoie un message d'erreur. Pareillement, pour se protéger du cas où f en entrée ne possède pas de racine, la fonction envoie un message d'erreur si, le nombre d'itérations dépasse 300. Nous avons estimé que ce nombre était suffisant car nous avons observé que pour un grand nombre de fonctions dans des intervalles plus ou moins grands, le nombre d'itérations pour trouver une racine dépassait rarement 20.

Finalement, nous avons aussi pu constater que la méthode de la sécante converge plus rapidement vers la racine que la méthode de la bissection. En appelant la fonction *secante.m* au cas où la fonction *bissection.m* ne trouve pas de racine rend donc cette dernière très robuste. (Si un appel à la sécante a été fait, la fonction *bissection.m* renvoie un message pour le dire).

2. Implémentation et résolution du système d'équations différentielles

A présent, notre objectif est de modéliser le comportement de l'activité neuronale sur base du système d'équations différentielles suivant :

$$\frac{dV}{dt} = V - \frac{V^3}{3} - n^2 + Iapp \quad (1)$$

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

où $n_\infty(V) = \frac{2}{1+e^{-5V}}$ et où V représente la différence de potentiel de la membrane du neurone, n est la perméabilité de la membrane au potassium et $Iapp$ est le courant externe appliqué sur la membrane. Il est important de préciser que le modèle *(1)* - *(2)* est adimensionnel, à l'exception du temps, exprimé en *ms.*

(Précisons que, pour éviter de hard-coder les constantes, nous avons établi un fichier qui répertorie, dans une structure, l'ensemble des « variables globales » qui interviennent tout au long du travail. Ainsi, nous ferons appel à cette fonction *var_glob.m* au début des fonctions pour lesquelles c'est nécessaire.)

2.1 Résolution du système d'équations différentielles

2.2.1 La fonction *nagumo.m*

La première étape pour pouvoir exploiter le système *(1)* - *(2)* était d'implémenter une fonction *nagumo.m.* Cette fonction prend en entrée une valeur temps[1], des valeurs V et n et une valeur $Iapp$. Elle contient simplement les deux équations *(1)* et *(2)* du système, les résout, et renvoie son membre de droite dans une matrice 2×1.

2.2.2 Résolution avec la méthode d'Euler explicite

Une fois la fonction *nagumo.m* correctement implémentée, nous pouvons nous en servir pour résoudre le système d'équations différentielles sur un intervalle de temps quelconque. C'est ce que fait notre fonction *eulexp.m,* basée sur la méthode d'Euler explicite. Ses paramètres formels sont : une vecteur temps T, un vecteur contenant les conditions initiales C et une valeur de $Iapp$. Elle fonctionne à l'aide d'une boucle dans laquelle la fonction *nagumo.m* est appelée à chaque itération, en prenant initialement des valeurs de V et n données en entrée dans C. Ensuite, elle prend des valeurs V et n calculées par $x_{i+1} = x_i + hf(x_i, t_i)$, où h est le pas d'intégration (cfr. 2.2.3) et $f(x_i, t_i)$ est la dérivée de x_i par rapport au temps calculée à l'itération précédente.

[1] Ce paramètre d'entrée, bien qu'il soit demandé dans les consignes, n'est jamais utilisé.

De cette manière, elle nous renvoie l'évolution de V et n au cours du temps. Le critère d'arrêt de cette boucle est simplement calculé par $\frac{T(2)-T(1)}{h} - 1$ [1], de sorte que la boucle n'effectuent les calculs que pour l'intervalle de temps considéré.

La fonction *eulexp.m* retourne finalement une matrice dont la première ligne contient les valeurs du temps, la seconde et la troisième contiennent respectivement les valeurs successives de V et de n. Sur la *Figure 1*, on peut voir le résultat renvoyé par *eulexp.m* sur l'intervalle de temps $[0\ 50]$ avec un $Iapp = 0.1$, une condition initiale pour V de -1.5 et pour n de 0.5.

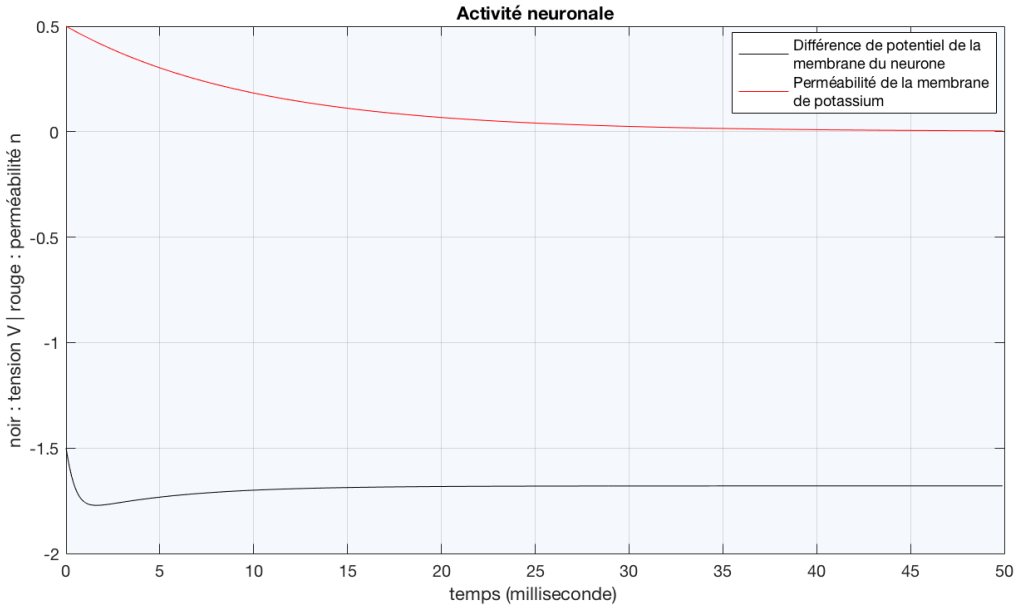

Figure 1 : Résultat par la méthode d'Euler explicite

2.2.3 Discussion du pas d'intégration

Le pas d'intégration h devait être bien choisi pour que la précision soit satisfaisante et que le temps d'exécution ne soit pas trop long. Nous avons donc testé plusieurs pas et nous avons conclu qu'un pas de 0.1 donnait le résultat le plus satisfaisant. Nous avons ensuite testé si la méthode d'Euler explicite était stable avec ce pas, ce qui est le cas s'il satisfait l'inégalité suivante : $|1 + hJ_i| < 1$ *(a)* pour tout i, où $J_i = \frac{\partial f}{\partial x}(\zeta_i, t_i)$ $(\zeta_i \in [\bar{x}_{i-1}\ ;\ x^*(t_{i-1})])$. Pour tester cette inégalité, nous avons implémenté un script *Jacob.m* qui, premièrement, utilise la fonction matlab *jacobian* pour trouver les dérivées partielles des fonctions du système d'équations différentielles. Nous avons ensuite testé si l'inégalité *(a)* est vérifiée pour les dérivées partielles de la fonction *(1)* du système d'équations (c'est cette fonction qui nous intéressera particulièrement dans la suite du travail) et pour toutes les valeurs de V et de n renvoyées par *eulexp.m* avec $Iapp = 0.1$. Et nous avons constaté que c'était bien le cas pour un pas h de 0.1. On peut donc en conclure que la méthode d'Euler explicite est stable et, par conséquent, la fonction *(1)* du système aussi.

[1] On retire 1 pour éviter une itération supplémentaire due au dernier x_{i+1} de la boucle, de sorte que la matrice renvoyée ait une taille donnée par un chiffre rond, ce qui nous sera utile pour la suite.

2.2.2 Résolution avec la fonction ode45

Le solveur *ode45*, déjà implémenté dans matlab, permet de résoudre des systèmes d'équations ordinaires en utilisant deux paires de méthodes de Runge-Kutte, soit les deux approximations d'ordre 4 et 5 de $x(t+h)$. Ode45 intègre le système d'équations différentielles dans l'intervalle de temps qu'on lui transmet en entrée et renvoie chaque valeur dans un vecteur colonne. Les tolérances par défaut sur les erreurs des valeurs renvoyées sont 10^{-3} pour l'erreur relative et 10^{-6} pour l'erreur absolue. Dans le script que nous avons implémenté, nous avons décidé de ne pas les modifier car la précision reste très satisfaisante.

Cependant, la fonction contenant le système d'équations différentielles en entrée de la fonction *ode45* ne peut contenir en paramètres formels qu'un vecteur temps et un vecteur de conditions initiales. C'est pourquoi nous ne pouvons pas utiliser *nagumo.m*, mais que nous avons plutôt implémenté une « copie » (appelée *fctode.m*) dans laquelle le $Iapp$ y est fixé au lieu d'être en paramètre d'entrée.

Le script, *ode.m*, résout donc le système d'équations présent dans *fctode*.m sur un intervalle de temps qui vaut $[0\ 500]$ et des conditions initiales qui, vu l'énoncé, valent -1.5 pour V et 0.5 pour n (ces données peuvent évidemment être modifiées si nécessaire). Nous faisons aussi directement appel à la fonction *plot* dans ce script pour pouvoir avoir un aperçu des résultats renvoyés par *ode45*. En affichant les résultats obtenus par *eulexp.m* et ceux obtenus par *ode.m* dans les mêmes conditions ($t = [0\ 200]$; $Ci = [-1.5\ 0.5]$; $Iapp = 1$), on obtient la *Figure 2*, dans laquelle on peut voir que la méthode d'Euler renvoie des valeurs de moins en moins précises, vu que chaque x_i est calculé avec l'erreur du précédent. La précision d'*eulexp.m*, avec un pas de 0.1, reste tout de même suffisamment fiable.

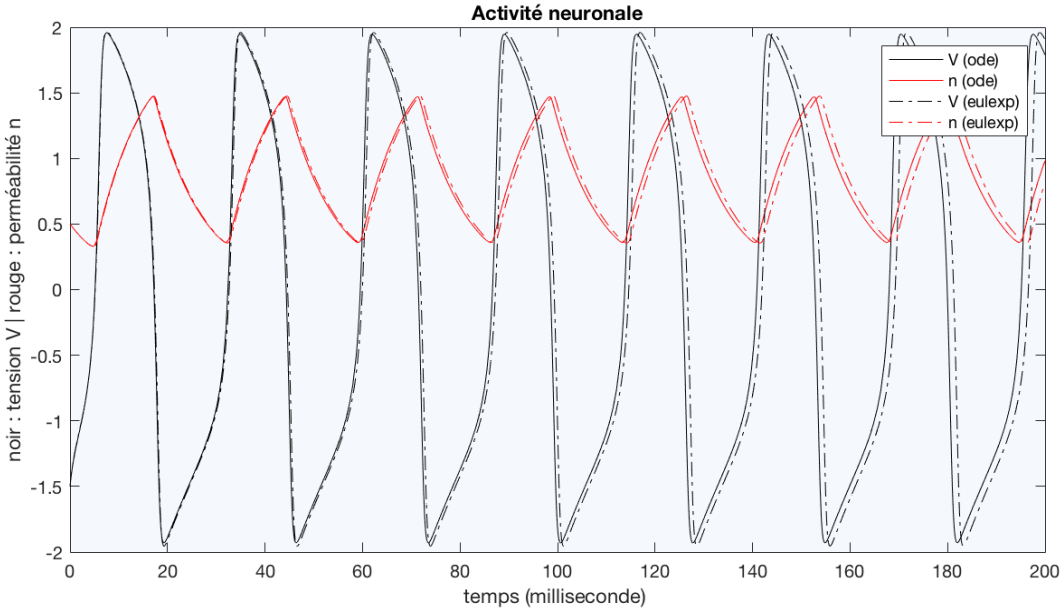

Figure 2 : Comparaison ode45 et Euler explicite

3. Détermination de l'activation nécessaire pour obtenir un spike

En utilisant la fonction *eulexp.m*, on a pu voir que, en fonction de l'intensité du courant $Iapp$ en entrée, le neurone produisait des spikes réguliers ou non. Dans cette partie, nous allons essayer de trouver les limites de $Iapp$ pour lesquelles les spikes sont réguliers et déterminer leur fréquence.

3.1 Spikes réguliers

Tout d'abord, nous avons implémenté une fonction *spike.m* qui détermine, pour une valeur de $Iapp$ en entrée, si elle provoque des spikes réguliers ou non. En premier lieu, *eulexp.m* est appelé dans l'intervalle T $[0\ 800]$ et avec le $Iapp$ entré en paramètre. Les valeurs renvoyées sont sauvegardées dans une matrice Y. Seules les valeurs de la tensions V, soit la deuxième ligne de Y, nous intéressent ici.

Comme l'objectif est de déterminer une fréquence, grâce à la relation $f = \frac{1}{T}$, où f est la fréquence et T la période, nous avons d'abord cherché à déterminer la période de la fonction Y (les valeurs successives de V). Pour se faire, une boucle (dont le gardien est $\frac{T(2)-T(1)}{h} - 1$) cherche d'abord tous les maxima locaux de la fonction Y et conserve leurs ordonnées et abscisses dans des vecteurs *ordmax* et *absmax*.

Ensuite, nous ne nous intéressons plus qu'aux fonctions pour lesquelles plus d'un maximum ont été trouvés, les autres étant à coup sûr non périodique (la fonction renvoie 0). Pour ces fonctions, nous avons remarqué qu'il fallait aussi prendre en compte certains comportements particuliers. Le premier se présente à partir d'une certaine intensité de $Iapp$. On voit que le premier spike s'élève à une tension plus élevée que les suivants, qui se comportent ensuite tous de la même manière, comme illustré sur la *Figure 3*. Le second comportement particulier que nous avons remarqué se présente pour un $Iapp$ de 3.27. Comme on peut le voir sur la *Figure 4*, il y a plusieurs maxima locaux, pourtant la tension tend assez vite vers 1 et la fonction n'est donc pas périodique.

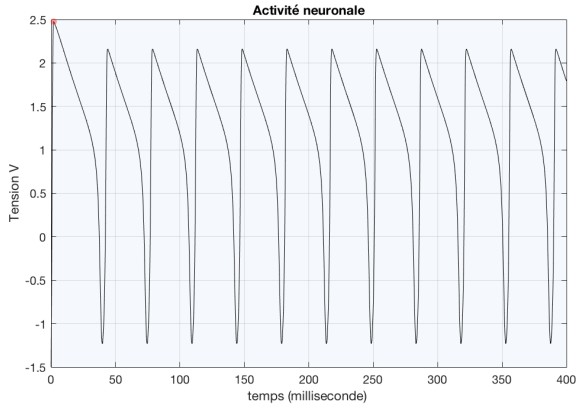

Figure 3 : Tension pour un Iapp de 3

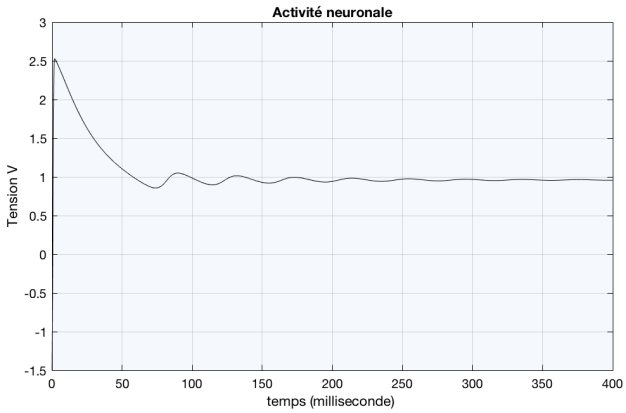

Figure 4 : Tension pour un Iapp de 3.27

Pour les cas similaires à la *Figure 3*, nous considérons que la fonction est tout de même périodique et que *spike.m* doit renvoyer la fréquence. Pour ceux similaires à la *Figure 4*, *spike.m* doit naturellement renvoyer 0. En somme, la condition pour que *spike.m* renvoie la fréquence d'une fonction est que cette dernière possède plus d'un maximum et que les ordonnées de tous ceux-ci, **à l'exception d'un**, soient égales (à une tolérance de 0.03 près).

Sous cette condition, la moyenne des périodes[1] entre deux maxima T_m est d'abord calculée à l'aide de la fonction *mean*, déjà implémentée dans matlab, et la fréquence est ensuite calculée selon $\frac{1}{T_m \times 10^3}$ (où le facteur 10^3 sert à convertir la période qui est donnée en millisecondes en secondes) et renvoyée. Dans tous les autres cas, *spike.m* renvoie 0.

3.2 Recherche d'intensités de courant provoquant des spikes réguliers

A présent, on considère le problème inverse. On aimerait connaitre quelle intensité de courant $Iapp$ faut-il appliquer à la membrane du neurone pour qu'il produise des spikes à une fréquence désirée. La fonction qui se charge de cette recherche est appelée *recherche_iapp.m* et prend donc en entrée la valeur d'une fréquence.

Tout d'abord, dans une boucle, la fonction fait appel à *spike.m* avec des valeurs de $Iapp$ commençant à 0 puis augmentées d'un pas de 0.01 à chaque itération pour déterminer chaque valeur de fréquence qui y correspond. Pour visualiser les opérations qui vont être faites, un script se charge de tracer ce graphique, qu'on peut voir sur la *Figure 5*. Cependant, après le maximum de ce graphique, les valeurs des fréquences trouvées en fonction du $Iapp$ n'ont plus de signification physique. La boucle s'arrête donc une fois ce maximum atteint et seul le graphique de la *Figure 6* est considéré dans la suite de la fonction *recherche_iapp.m.*

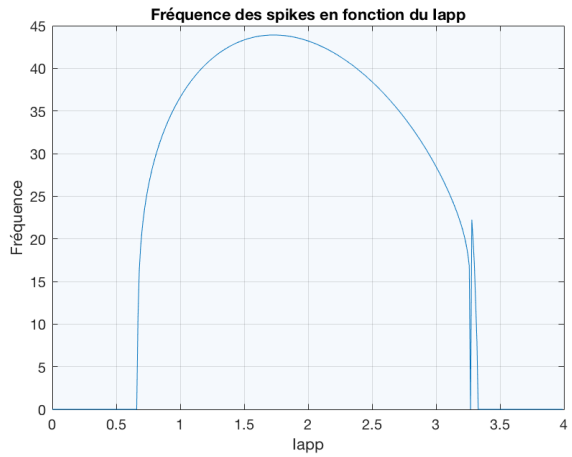

Figure 5 : Fréquences en fonction des Iapp

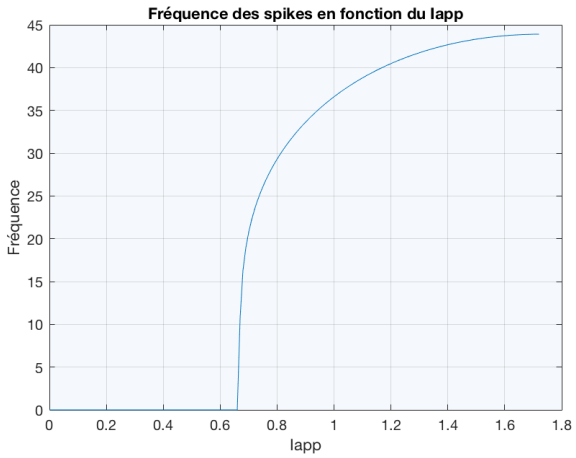

Figure 6 : Graphique considéré

Ensuite, pour déterminer le $Iapp$ correspondant à la fréquence en entrée, il ne reste qu'à soustraire cette fréquence à la fonction représentée sur la *Figure 6* et à rechercher la racine de cette nouvelle fonction. Cette recherche se fait en appelant la fonction *bissection.m*. Elle prend en entrée la nouvelle fonction « rabaissée » (directement implémentée dans *recherche_iapp.m*) dont elle recherche la racine entre 0 et le $Iapp$ maximum. Nous avons décidé de faire appel à la fonction *bissection.m* plutôt qu'à la fonction *secante.m* car bien que plus rigide, elle est plus robuste. De plus, vu son implémentation, elle renvoie toujours une racine appartenant à l'intervalle passé en entrée, ce qui correspond bien à notre demande ici. Finalement, *recherche_iapp.m* renvoie la racine trouvée qui correspond bien au $Iapp$ recherché.

[1] Nous avons décidé de calculer la moyenne des périodes plutôt qu'une seule dans le but d'être plus précis.

De plus, *recherche_iapp.m* retourne un message d'erreur si la fréquence en entrée dépasse la fréquence maximale qui peut être provoquée.

Aussi, si c'est *'limite'* qui est entrée en paramètre, la fonction crée une matrice A qui contient les intensités de courant successives sur la première ligne et les fréquences correspondantes sur la deuxième. Ensuite, toutes les colonnes dans lesquelles la fréquence vaut 0 sont supprimées. La première valeur sur la ligne des $Iapp$ correspond donc maintenant à la première valeur pour laquelle la fréquence n'est pas nulle, soit pour laquelle le neurone commence à produire des spikes réguliers. Ainsi, l'utilisateur peut appeler la fonction en entrant *'limite'* s'il désire que cette dernière lui renvoie l'intensité du courant à partir de laquelle les spikes produits sont réguliers.

En somme, les fonctions créées dans cette partie ont permis de mieux connaître, caractériser et visualiser le comportement du neurone. On peut maintenant les utiliser pour aller plus loin dans cette analyse, ce que nous allons faire à la section suivante.

4. Transmission de messages par le neurone

A présent, notre nouvel objectif est double. Nous allons d'abord tenter d'illustrer le comportement du neurone lorsqu'il est soumis à un courant d'entrée constant par morceaux. A l'inverse, nous essayerons ensuite d'exprimer l'information reçue par ce type de message analogique à l'aide d'un message digital.

4.1 Message analogique

La fonction qui reproduit le message analogique du neurone est appelée *analogique.m.* Le résultat de cette fonction est assez similaire à celui de la fonction *eulexp.m*, sauf qu'ici, le courant d'entrée est constant par morceaux et seules les valeurs de la tension V nous intéressent à la sortie. La fonction *analogique.m* prend donc en entrée un vecteur t de taille m qui correspond à des intervalles successifs de temps et un vecteur $Iapp$ de taille $m-1$, dont les valeurs doivent être des intensités de courants correspondant aux intervalles. Pour satisfaire à l'objectif, nous avons créé une boucle qui appelle la fonction *eulexp.m* pour chaque intervalle de temps et chaque $Iapp$ correspondant. Dans cette boucle, on réinitialise aussi, à chaque itération, les valeurs des conditions initiales d'entrée pour *eulexp.m* pour qu'elles correspondent bien à la prochaine borne inférieure du vecteur temps. Le gardien de boucle est simplement la longueur du vecteur $Iapp$. Les résultats renvoyés par *eulexp.m* sont stockés dans une matrice qui contient, à la sortie, tous les points successifs V qui correspondent à la réaction du neurone au courant constant par morceaux $Iapp$.

Sur la *Figure 7*, on peut observer un exemple d'une réaction d'un neurone soumis à des courants d'entrée de $0;\ 5;\ 0;\ 0.3;\ 0;\ 0.9;\ 0;\ 0.7;\ 0$ sur les intervalles de temps respectifs $[0\ 100]$; $[100\ 120]$; $[120\ 200]$; $[200\ 220]$, … $[420\ 500]$.

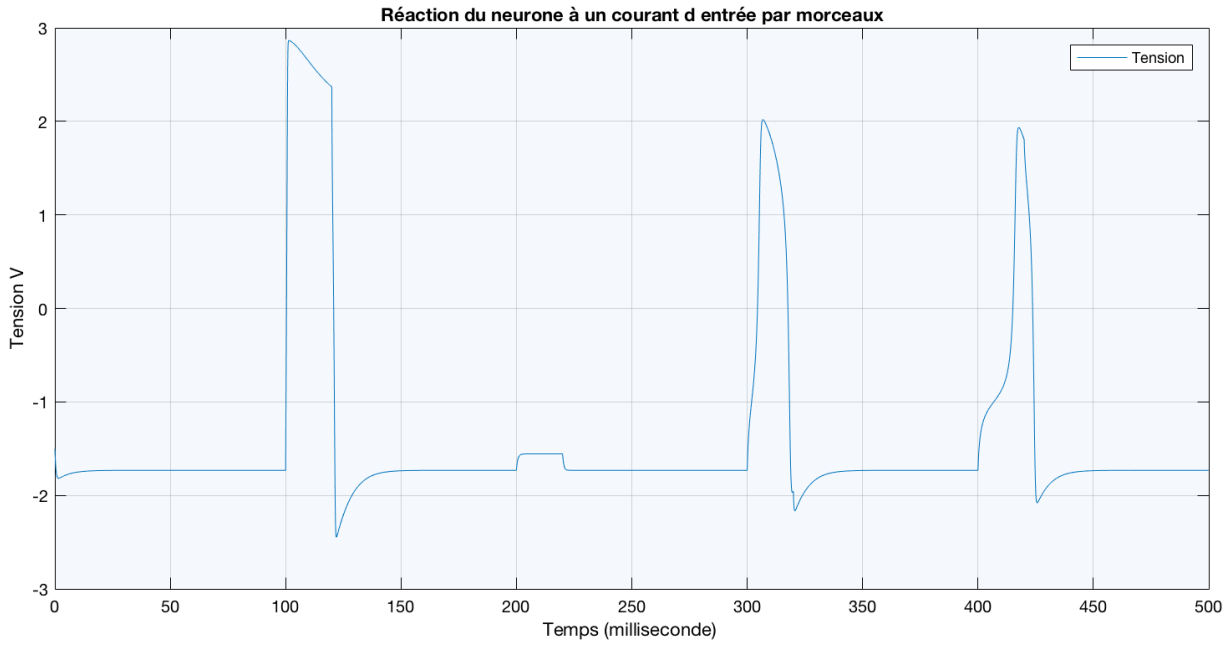

Figure 7 : Comportement analogique du neurone

4.2 Message digital

La dernière fonction que nous avons implémentée, *digital.m*, fait l'inverse de la fonction *analogique.m*. En effet, elle reçoit, en entrée, des informations produites par le neurone et a pour but de déterminer les impulsions qui lui ont étés envoyées. Elle retourne le message digital correspondant aux informations reçues en entrée, qui est composé de 0 et de 1, 1 si l'impulsion envoyée donne lieu à un spike et 0 dans le cas inverse. En tenant compte de la convention qu'une impulsion de courant de 20 millisecondes est envoyée toutes les 100 millisecondes, nous avons imaginé l'algorithme suivant :

Au temps $t_0 = 0\ ms$, on connait : les conditions initiales, le $Iapp$ $(= 0)$ et le premier intervalle de temps ($[0\ 100]$) ⇨ On peut donc utiliser la fonction *eulexp.m* pour déterminer les valeurs de V et de n qui serviront de conditions initiales pour l'intervalle de temps suivant.

Au temps $t_1 = 100\ ms$, on connait : les conditions initiales (trouvées juste avant) et l'intervalle de temps considéré ($[100\ 120]$) ⇨ Il reste donc à rechercher le $Iapp$ qui a été envoyé au neurone. Pour cela, une boucle appelle *eulexp.m* avec une valeur de $Iapp$ d'abord nulle et augmentant d'un pas de 0.01 à chaque itération. Et on recherche quel $Iapp$ provoque le même comportement que celui donné en entrée dans le même intervalle de temps, en comparant si les tensions maximales dans ces intervalles sont égales (à une tolérance de 0.01 près). Si c'est le cas, la valeur du $Iapp$ trouvée est celle qui avait été envoyée au neurone, elle est donc est conservée.

Au temps $t_2 = 120\ ms$, les conditions sont déterminées facilement car elles correspondent aux dernières valeurs de V et de n renvoyées par *eulexp.m* pour le $Iapp$ déterminé juste avant. On utilise ensuite le même procédé que pour le temps t_0 car vu les conventions, on connait l'intervalle de temps et le $Iapp$ correspondant ($[120\ 200]$ et 0). On peut donc trouver les nouvelles conditions avant l'impulsion suivante ($t_3 = 200\ ms$) et déterminer le $Iapp$ de cette impulsion de la même manière que pour le temps t_1. En fonctionnant ainsi de proche en proche, on trouve toutes les valeurs des $Iapp$ qui avaient provoqué le comportement du neurone donné en entrée. Le gardien de boucle est donné par $\frac{length(tensions)}{\frac{100}{h}} - 1$, où $length(tensions)$ est la longueur du vecteur contenant les tensions mesurées en entrée et h est le pas de temps avec lequel ces mesures ont étés faites. On obtient ainsi le bon nombre d'impulsions à rechercher.

La dernière étape du code consiste à transformer le vecteur contenant les valeurs retrouvées de $Iapp$ (à 0.01 près) en un message digital. Pour cela, on fait une boucle qui fait appel à la fonction *spike.m* pour chaque composante du vecteur et qui renvoie 1 si la fonction *spike.m* renvoie une fréquence (ce qui veut dire que cette valeur de $Iapp$ est dans l'intervalle qui provoque des spikes réguliers) et un 0 sinon. Le message digital final est ainsi formé et renvoyé.

5. Conclusion :

En conclusion, nous avons beaucoup appris sur les méthodes numériques et sur leurs utilités. Nous avons aussi remarqué que chacune des fonctions de notre travail avait son importance et que beaucoup d'entre elles faisaient appel à d'autres, d'où l'importance de se montrer précis et rigoureux à leur implémentation. Nous sommes satisfaits du travail que nous avons réalisé et du résultat global qui nous parait déjà très concluant et intéressant.

Bibliographie

1. Enoncé 2018. Introduction aux méthodes numériques et projet : Modélisation de l'activité neuronale. Brüls O. Louveaux Q. Nguyen F.

2. LOUVEAUX Q. Introduction aux méthodes numériques. Edité par la Centrale des Cours de la FSA asbl.