Universidad Nacional Autónoma de México

Facultad de Ciencias

Implementación de un modelo termodinámico de excitabilidad membranal con Python y pyQt

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

Licenciado en Ciencias de la Computación

PRESENTA:

Martín García Mata

TUTOR

Mariana Duhne Ramírez

Ciudad Universitaria, Cd. Mx., 2019

Universidad Nacional Autónoma de México

Biblioteca Central

Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Dedicado a mis padres Jey y Raúl,
los mejores amigos y guías.

Agradecimientos

Gracias a mi tutora, la Lic. IBB Mariana Duhne Ramírez por la dedicación, compromiso y esfuerzo para que de la mejor manera, este escrito sea una realidad.

Al Dr. José Bargas Díaz por sus consejos y el lugar en su laboratorio en el Instituto de Fisiología Celular para realizar mi trabajo, institución al que también agradezco las facilidades otorgadas.

Al Dr. Marco Arieli Herrera Valdez por toda la orientación acerca de su modelo y las observaciones a este trabajo.

A la M. en I. Karla Ramírez Pulido por las correcciones y conocimientos que permitieron pulir el mensaje que se buscaba dar con este escrito.

A la Dra. Ursula Xiomara Iturrarán Viveros por la revisión a esta tesis.

Y no menos importantes agradezco a mis padres Jey, Raúl y mi familia además de un recuerdo a Elena y todo aquel que ya no se encuentre entre nosotros. A los nuevos amigos, Mariana, Esther, José, todos en el laboratorio de la Dra. Yazmín Ramiro, todos en el laboratorio del Dr. Fatuel Tecuapetla, Antonio Laville. A los viejos amigos, Francisco, Daniel, Iván, Karina, Arturo... y si usted cree que falta dejaré un espacio aquí adelante para con gusto, ser agregado ________________.

Muchas gracias por permitirme aprender todo lo que sé de la vida por ustedes, pero principalmente, *nunca dejarme sólo*.

Índice general

Introducción **1**

Objetivos **2**

Capítulo 1: Conocimientos previos **3**

1.1. Células excitables 3
1.2. Potencial en reposo 3
1.3. Potencial de acción 4
1.4. Movimiento de iones y leyes físicas 5
1.5. Modelo de Goldman Hodgkin y Katz 7
1.6. Modelo termodinámico 8
1.7. Modelos matemáticos 10
1.8. Modelo Hodgkin y Huxley: una explicación 10
1.9. Reducción a 2 dimensiones (modelo de Rinzel) 12
1.10. Modelo de deriva-difusión 13
1.11. Modelo a implementar 15
1.12. Ecuaciones diferenciales en computación 17
1.13. Método Runge-Kutta 18
1.14. Programación Orientada a Objetos 19
1.15. Modelo-Vista-Controlador 20
1.16. Python: ventajas para el proyecto 21
1.17. Interfaces gráficas con pyQt: adaptarse al usuario final 22
1.18. Método Runge-Kutta contra biblioteca OdeInt 23

Capítulo 2: Desarrollo del sistema computacional. **25**

2.1. Antecedentes del proyecto 25
2.2. Lista de requerimientos 26
2.3. Diagrama de casos de uso 27
2.4. Descripción teórica de la implementación 29
2.5. Estructura lógica de los archivos del proyecto 29
2.6. Estructura del código del proyecto 30
2.7. Estructura lógica de los archivos de la interfaz gráfica 40

2.8. Estructura del código de la interfaz gráfica 41

Capitulo 3: Resultados del proyecto **49**
3.1. Resultados de la ejecución del proyecto 49
3.2. Simulación de células excitables 53
3.2.1. Célula de nodo sinoauricular 53
3.2.2. Interneurona estriatal de disparo rápido 54
3.2.3. Interneurona colinérgica 55

Conclusiones **56**

Apéndice **57**
4.1. Biblioteca de valores ejemplo 57

Bibliografía **63**

Introducción

Los modelos matemáticos de potencial transmembranal celular son una herramienta útil para el estudio del comportamiento de células excitables. Provee explicaciones de la dinámica del potencial celular que permite llevar a cabo experimentos *in sílico*, que aún no son realizables con técnicas electrofisiológicas *in vitro*, debido a limitaciones físicas o tecnológicas. La mayoría de los modelos actuales usan una ecuación para describir el flujo por cada tipo de canal, lo que hace difícil modelar diferentes tipos de células. Sin embargo, el modelo de termodinámico (Herrera-Valdez (2015)) propone una ecuación general para describir el flujo iónico transmembranal mediado por canales y transportadores, que facilita el modelado de diferentes células excitables.

Uno de los problemas que surgen es que para hacer uso de este modelo termodinámico (así como otros) hay que llegar a la solución de la ecuación general de flujo iónico transmembranal, la cual no existe por lo que se realiza una aproximación con métodos numéricos. Debido a lo anterior, la solución de la ecuación general del modelo se programa y ejecuta en un equipo de cómputo.

Aunque ya existen diferentes implementaciones del modelo para aprovechar las bondades que éste ofrece, es necesario diseñar una implementación que ocupe las estructuras teóricas de cómputo apropiadas como las presentes en el paradigma Orientado a Objetos, las cuales permiten aprovechar las bondades intrínsecas del modelo.

El problema expuesto se resolvió usando el lenguaje de programación Python, utilizando las bibliotecas disponibles para crear un entorno que permita la manipulación y experimentación para el usuario final a través del uso de una aplicación con interfaz gráfica.

Objetivos

General:

- Implementar el modelo termodinámico (Herrera-Valdez (2018a), Herrera-Valdez (2015)) para modelar el potencial transmembranal de varios tipos de células excitables con Python.

Secundario:

- Generar un entorno para aprendizaje y experimentación con el modelo termodinámico de potencial transmembranal de deriva-difusión

Particulares:

- Integrar la implementación a una Interfaz Gráfica de Usuario programada en pyQT, para permitir la manipulación ergonómica del modelo para personas no familiarizadas con programación.
- Implementar utilizando el modelo - vista - controlador de manera que el entorno de manipulación y el modelo matemático permanezcan como estructuras independientes, pero con la cercanía necesaria para realizar una aplicación compleja con un correcto intercambio de mensajes.
- Evaluar la funcionalidad modelando distintos tipos celulares.

Capítulo 1: Conocimientos previos

En este capítulo repasamos algunos conceptos de fisiología, física, matemáticas y la interacción que estas definiciones llegan a tener, que se consideran útiles para acercarse a los alcances y limitantes que el proyecto tiene en su elaboración.

1.1. Células excitables

Las neuronas en el cerebro, las células cardíacas en el corazón y las células beta en el páncreas, son ejemplos de células con actividad eléctrica o excitables. Dicha actividad es el resultado de la distribución de las cargas eléctricas de los iones tanto en la parte interna y externa de la membrana celular y de la capacidad de mover estos iones a través de la membrana celular. Estas unidades estructurales tienen diferentes funciones en el cuerpo: en el caso de las neuronas permiten el intercambio de señales entre el cerebro y otras partes del cuerpo; las células cardíacas, aseguran la correcta contracción del corazón para bombear sangre al resto del cuerpo y las células pancreáticas permiten la secreción de insulina para el control de los niveles de glucosa en la sangre (Miura (2002)).

1.2. Potencial en reposo

Las células se encuentran delimitadas por una bicapa lipídica que separa el interior y exterior de la misma. Esto aísla el ambiente intracelular, limitando el tráfico de sustancias hacia dentro o fuera de la célula. La membrana tiene proteínas embebidas que comunican el interior con el exterior celular y por lo tanto permiten el flujo controlado de sustancias. El flujo de átomos que acarrean carga eléctrica, es decir, iones es lo que define la excitabilidad celular. Los iones que participan en la excitabilidad celular son: Sodio (Na+), Potasio (K+), Calcio (Ca2+) y Cloro (Cl-), identificando que las primeras tres tienen una carga positiva (cationes) y el último negativa (anión). El flujo iónico a través de la membrana celular es posible por la presencia de transportadores y de poros que se denominan canales y generalmente permiten el paso solo de un ion específico (Kandel et al. (2000)).

La membrana regula el intercambio iónico generando una concentración diferencial de cargas fuera (extracelular) y dentro (intracelular) de la membrana, por lo que

existe una diferencia de potencial eléctrico, definida como “potencial de membrana en reposo”. La diferencia de potencial es negativa adentro en relación con el exterior, por lo tanto, a la transición a valores menos negativos se denomina “despolarización” y si pasa a valores en extremo negativos se le llama “hiperpolarización” (Kandel et al. (2000)).

1.3. Potencial de acción

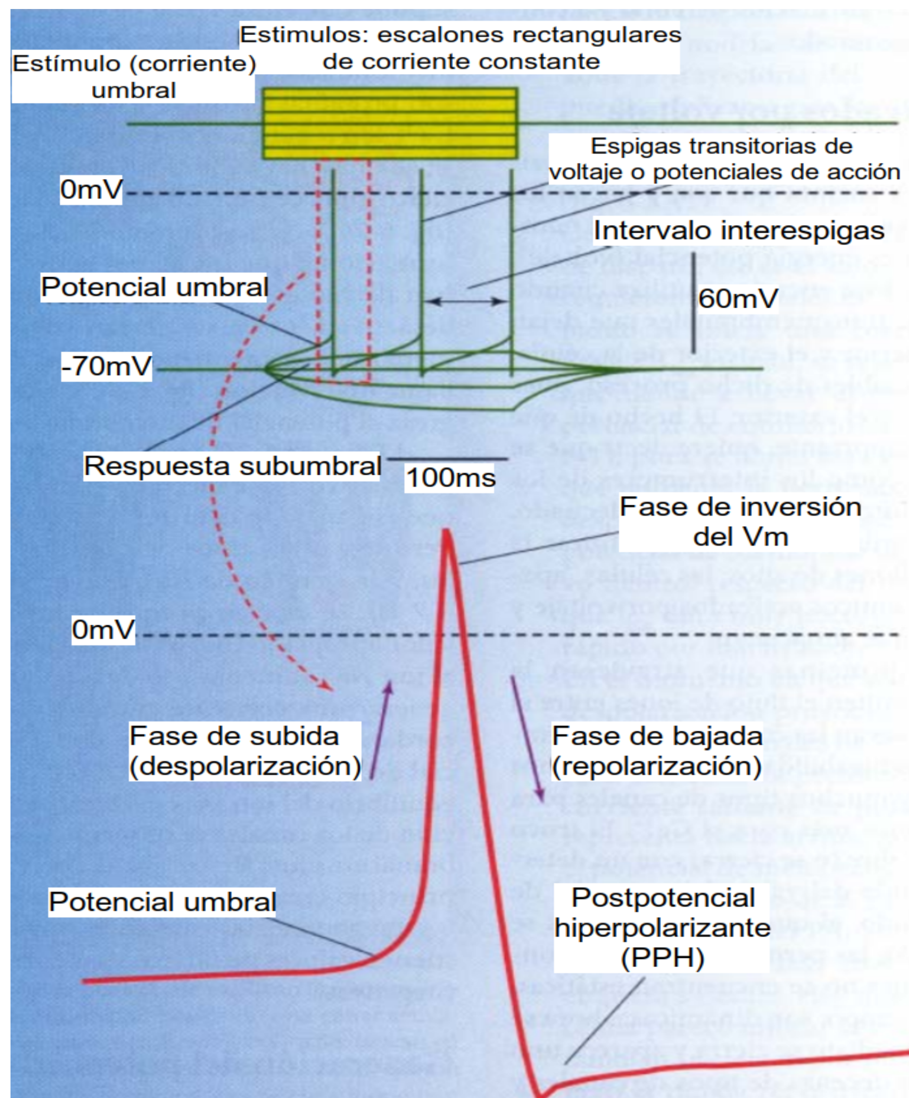


Figura 1.1: Diagrama de un Potencial de acción (Galarraga and Bargas-Díaz (2010)).

Cambios instantáneos en el potencial de membrana en reposo, en ventanas de milisegundos se define como "potencial de acción" o por su forma "espiga". Las funciones de las células excitables se asocian a la presencia de potenciales de acción. Estos cambios súbitos en el potencial de membrana dan lugar a la comunicación neuronal, a la contracción del corazón o a la secreción de insulina según sea el tipo celular que se esté estudiando. El potencial de acción se origina por el flujo de iones a través de la membrana, porque al acarrear cargas hacia dentro o fuera de la membrana el valor de la diferencia del potencial cambia (Galarraga and Bargas-Díaz (2010)).

El flujo de iones a través de la membrana se describe en la Figura 1.1, esta muestra la respuesta en voltaje registrada en una sola célula (trazo verde), con la técnica de *patch clamp* en célula entera en la configuración de fijación de corriente. La corriente inyectada a la célula para generar dicha respuesta se muestra en amarillo. En respuesta a un pulso depolarizante que llega al umbral, se producen potenciales de acción o espigas. La magnificación de un potencial de acción se muestra abajo (trazo rojo). Al llegar el potencial (voltaje) a umbral hay una fase de subida (despolarización) dada por la apertura de canales de sodio dependientes de voltaje. Cuando estos se inactivan y se activan los canales de potasio viene la fase de inversión y más tarde la fase de bajada (repolarización). Al regresar el voltaje baja más de su nivel original es decir hay un pospotencial hiperpolarizante (PHP). La generación de potenciales de acción son la firma de las células excitables (Galarraga and Bargas-Díaz (2010)).

1.4. Movimiento de iones y leyes físicas

Para describir y modelar el movimiento de iones a través de la membrana se utilizan las leyes físicas. El movimiento de iones a través de la membrana es descrito por el flujo de masa que se llama difusión, expresada por la Ley de Fick (Johnston and Wu (1995))

$$J_{diff} = -D\frac{\delta[C]}{\delta x} \tag{1.1}$$

Donde: J_{diff} = flujo de difusión ($moléculas/seg - cm^2$), $-D$=coeficiente de difusión (cm^2/seg), $[C]$=concentración de iones ($moléculas/cm^3$), el signo negativo en D denota que J tiene un flujo que va de altas a bajas concentraciones.

Describe la tendencia del movimiento de partículas cuando no hay una distribución homogénea de las moléculas en el espacio, es decir, donde existe un gradiente de concentración entre dos puntos, como lo hay para las diferentes especies iónicas en ambos lados de la membrana (Johnston and Wu (1995)).

Al tratarse de átomos con una carga, la difusión además está relacionada con un campo eléctrico y un cambio en el potencial eléctrico.

Por ello, a la descripción de la ley de Fick se le agregan términos relacionados con la Ley de Ohm para determinar velocidad a la deriva (Johnston and Wu (1995)):

$$J_{drift} = \delta_{el}E = -\mu z[C]\frac{\delta V}{\delta x} \tag{1.2}$$

Donde: J_{drift} = flujo de corriente ($moléculas/seg - cm^2$), δ_{el}=conductividad eléctrica ($moléculas/V - seg - cm$), E=campo eléctrico (V/m) = $\frac{\delta V}{\delta x}$, μ = movilidad ($cm^2/V - seg$), z = valencia del ion, $[C]$=concentración.

La velocidad a la deriva es el promedio de velocidad de los electrones fluyendo dentro de un conductor (o conducto) bajo la influencia de un campo eléctrico, esta velocidad es directamente proporcional entonces a la fuerza del campo eléctrico. Los iones moviéndose a través de una membrana tendrían esta misma influencia. El coeficiente de difusión es (Johnston and Wu (1995))

$$D = \frac{kT}{q}\mu \tag{1.3}$$

Donde: k = constante de Boltzmann ($1.38 \times 10^{-23} Joule/^\circ K$), T=temperatura absoluta ($^\circ K$), q=carga de la mólecula (C).

La función establece que la difusión y deriva en el mismo medio son procesos que se suman, debido a que la resistencia presente en el medio para estos procesos es la misma. Esta función simplifica el comportamiento del movimiento de iones en los sistemas biológicos, dado que los gradientes de concentración y potencial eléctrico tienen influencia sobre éstos (Johnston and Wu (1995)).

La corriente iónica por unidad de área (I) es igual al flujo (J) multiplicado por la valencia y la constante de Faraday. Esto viene de sumar el flujo por velocidad a la deriva y la difusión, como se muestra a continuación en la Ecuación de Nernst-Planck para el flujo total de especies cargadas (Johnston and Wu (1995)):

$$I = J \cdot zF = -(\mu z^2 F[C]\frac{\delta V}{\delta x} + \mu zRT\frac{\delta [C]}{\delta x}) \tag{1.4}$$

Donde: I = corriente iónica en la densidad actual (A/cm^2), J=Ecuación de Nernst-Plank en representación molar J/N_A ,N_A es el número de Avogrado ($6.02 \times 10^{23}/mol$),

J original es $J_{drift} + J_{diff}$, R=constante de gas ($1.98 cal/°K - mol$), F = constante de Faraday ($96,480 C/mol$), u = μ / N_A movilidad molar ($cm^2/V - seg - mol$).

A partir de la ecuación Nernst-Planck, podemos además calcular el potencial de equilibrio de los diferentes iones con la Ecuación de Nernst (Johnston and Wu (1995)):

$$E_i = V_m(I_i = 0) \stackrel{def}{=} V_{in} - V_{out} = \frac{RT}{zF} \ln \frac{[C]_{out}}{[C]_{in}} \tag{1.5}$$

Donde: E_i = equilibrio de potencial del ion, V_m=potencial de membrana.

El potencial de equilibrio del i-ésimo ion no es más que el potencial de membrana de la célula en el cual el flujo neto es igual a cero. Cuando la corriente que lleva el ion i es igual a cero, este se encuentra en el equilibrio de potencial del ion en términos de su concentración tanto afuera como adentro de la membrana (Johnston and Wu (1995)). El flujo de iones no es libre a través de la membrana, esta es impermeable a moléculas con carga, por lo que el flujo de iones es exclusivo de proteínas canal y transportadoras, de forma que los iones fluyen a través de la membrana siempre y cuando el potencial del mismo sea diferente a su potencial de equilibrio y exista algún canal o transportador que permita su paso a través de la membrana celular (Galarraga and Bargas-Díaz (2010)).

1.5. Modelo de Goldman Hodgkin y Katz

Como se mencionó anteriormente en las células excitables hay flujo de diferentes especies iónicas y cada una de ellas tiene su propio potencial de equilibrio. Además los canales y transportadores son específicas, de forma que no todas las especies iónicas pueden pasar por los mismos. El modelo de Goldman, Hodgkin y Katz, es una aproximación a esto y considera a la célula como un circuito eléctrico. Representa a la membrana como un capacitor eléctrico capaz de almacenar carga generando diferencia de potencial y a los canales iónicos como resistencias variables conectadas en paralelo (Clay (2005)).

$$\begin{aligned} I &= I_C + I_K + I_{Na} + I_{Cl} \\ &= C\left(\frac{\delta V}{\delta t}\right) + g_K(V - E_k) + g_{Na}(V - E_{Na}) + g_{Cl}(V - E_{Cl}) \end{aligned} \tag{1.6}$$

Donde: I= corriente total, I_n= corriente de cada ion a través de la membrana, g_n = conductancia de cada ion en un determinado momento, E_i = potencial de equili-

brio de cada ion, V=diferencia de potencial entre el interior y el exterior de la célula, C=capacitancia de la membrana.

En estado de reposo (basal):

$$I = 0 = \frac{\delta V}{\delta t}$$

entonces:

$$V = \frac{g_K E_K + g_{Na} E_{Na} + g_{Cl} E_{Cl}}{g_K + g_{Na} + g_{Cl}} \tag{1.7}$$

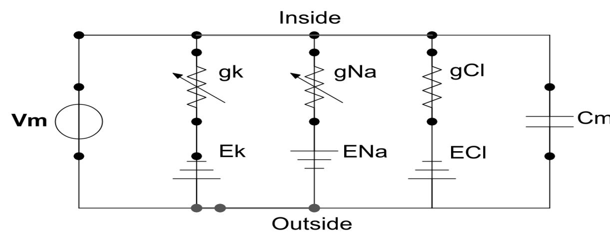


Figura 1.2: Modelo de conductancia paralela para corrientes iónicas (Johnston and Wu (1995))

De esta forma se puede calcular la corriente total que fluye a través de la membrana como la suma de la corriente que pasa a través del conjunto de canales y transportadores que permiten el paso de una especie iónica. La Figura 1.2 es el circuito eléctrico considerado en este modelo. La resistencia variable está relacionada con la diferencia entre el potencial de equilibrio del ion y la permeabilidad de la membrana a ese ion o conductancia que varía a lo largo del tiempo. Sin embargo esta variabilidad no es lineal debido a las propiedades de los canales y la membrana pero existen aproximaciones para modelar el comportamiento de la conductancia de los canales y transportadores (Clay (2005)). Uno de esos modelo es el termodinámico, modelo que se usó en este trabajo.

1.6. Modelo termodinámico

Utilizar un enfoque termodinámico para la generación de modelos, es realizar una descripción de la célula en términos de barreras energéticas que pueden ser traspasadas (Johnston and Wu (1995)). El cambio súbito de la diferencia de potencial en las células excitables es una consecuencia del movimiento de los iones a través de los espacios intracelular y extracelular de la membrana.

Estos iones se mueven a través de los canales iónicos debido a gradientes electroquímicos que actúan en ellos (Miura (2002)), por ello se puede definir el flujo de iones como el rompimiento de una barrera energética, este flujo es diferente dependiendo de la dirección del mismo, (Johnston and Wu (1995))

$$\begin{aligned} J_{inward} &= k_1 \beta [C]_{out} \\ J_{outward} &= k_2 \beta [C]_{in} \end{aligned} \tag{1.8}$$

Donde: J= El sentido de flujo unidireccional, k_i= coeficiente de cruce de membrana, $[C]$ = concentración interna o externa, β = coeficiente de reparto de membrana tipo agua de/para el ión.

Los coeficientes de cruce de membrana en equilibrio termodinámico se relacionan a la energía libre de activación (ΔG_0) y la constante de Boltzmann (esta constante define la relación entre la temperatura absoluta y la energía cinética contenida en cada molécula) y se calculan (Johnston and Wu (1995)):

$$\begin{aligned} k_1 &= Ae^{-(\Delta G_0 + (1-\delta)zFV)/RT} = k_0 e^{-(1-\delta)zFV/RT} \\ k_2 &= Ae^{-(\Delta G_0 + \delta zFV)/RT} = k_0 e^{\delta zFV/RT} \end{aligned} \tag{1.9}$$

Donde: k_0= $Ae^{\Delta G_0/RT}$, δ= Es un valor de asimetría de la barrera, valor 1 si la barrera esta afuera del margen de la membrana, valor 0 si la barrera está dentro de ese margen.

Con base en la ecuación de Nernst-Planck y combinando los valores definidos tenemos una nueva función para el modelo (Johnston and Wu (1995)):

$$\begin{aligned} I &= zF(J_{outward} - J_{inward}) \\ &= zF\beta k_0 [[C]_{in} e^{\delta zFV/RT} - [C]_{out} e^{(1-\delta)zFV/RT}] \end{aligned} \tag{1.10}$$

Donde: J= El sentido de flujo unidireccional, k_i= coeficiente de cruce de membrana, $[C]$ = concentración interna o externa, β = coeficiente de reparto de membrana tipo agua de/para el ión.

En sistemas biológicos no es real que todos los iones tengan que pasar una sola barrera de energía en su recorrido a través de la membrana, de modo que el uso de este modelo se restringe a la barrera que sobresale de manera exagerada a todas las demás que puedan existir (Johnston and Wu (1995)).

El modelo matemático que se implementará para este trabajo está basado en el movimiento de iones a través de la membrana de acuerdo a estas leyes físicas, de tal forma que se reproduzcan las características electrofisiológicas de células obtenidas de manera experimental.

1.7. Modelos matemáticos como herramienta para el estudio de células excitables

La computación neuronal, es la implementación en software de tecnología basada en redes de neuronas o unidades similares a una neurona, de una abstracción hecha por la teoría del cerebro o teoría celular. (Mallot (2013))

El cerebro humano cuenta con aproximadamente 1012 neuronas (Johnston and Wu (1995)), y existe una gran variedad de tipos de neuronas con diferencias radicales (Johnston and Wu (1995)). Por lo tanto, no es preciso decir que existe la neurona "típica", como comúnmente se cree. En la teoría del cerebro, se realiza una abstracción de las neuronas reales para entender el desarrollo, aprendizaje o sus funciones.

Esta abstracción deriva en el concepto conocido como "modelo" y las matemáticas aportan evidencia de su funcionalidad al predecir el resultado cuantitativo de las mediciones, a partir de fórmulas o mecanismos similares a los físicos, resultado del análisis de un conjunto de mediciones anteriores con características parecidas. Se pueden considerar pocas o muchas variables, así como se puede elegir si modelar la relación entre ellas o no. Debido a esto, el análisis de los modelos se puede convertir en una tarea no trivial, por la cantidad de pasos y factores tanto internos como externos que pueden participar (Rinzel (1985)). Es entonces que la computación toma importancia al acrecentar la fiabilidad de los resultados.

Debido a la unión de las disciplinas anteriores tenemos un conjunto de ejemplos y modelos que fueron y son un punto de inicio para encontrar y comprender elementos clave acerca del comportamiento particular de una neurona o red de neuronas (Fraser and Huang (2007)).

1.8. Modelo Hodgkin y Huxley: una explicación

El modelo de Hodgkin-Huxley, que pertenece a la rama de los modelos matemáticos biofísicos (Gerstner et al. (2014)), es el resultado de la publicación en 1952 de una serie de artículos que describen la teoría y los experimentos hechos para reproducir el potencial de acción y su propagación en el axón de calamar gigante (Hodgkin and Huxley (1952)).

En una analogía con un circuito eléctrico, la membrana es descrita como un capacitor con filtraciones, que son los canales. Y estos tienen diferente conductividad para cada tipo de ion. Este modelo se concentra en dos tipos de canal, Sodio (Na+) y Potasio (K+), cuya conductividad se puede calcular a través de las variables $\bar{m}$ y $\bar{h}$.

En el caso del Sodio, $\bar{m}$ es la variable de activación, es decir, determina la apertura de canales y $\bar{h}$ la de inactivación, un estado de los canales en los que no se encuentran cerrados pero no hay paso de iones a través de los mismos. Y en el caso del Potasio, la permeabilidad únicamente se describe una variable de activación $\bar{n}$.

Además, agregan lo que denominan corriente de fuga ($leak$) que representa la permeabilidad de la célula a iones y cuya conductancia es constante, dada por canales no específicos (Hodgkin and Huxley (1952)).

Conservando esta analogía del circuito eléctrico, se mantiene tanto la ley de la conservación de la carga y de la conservación de la energía para circuitos cerrados establecidas por Gustav Kirchhoff en 1845. Otro aspecto por considerar es que aunque distintos tipos de células excitables tienen comportamientos similares, el modelo considera el área de las células como constante, lo que no permite agregar los efectos que provoca un tamaño diferente de la célula (Hodgkin and Huxley (1952)).

El modelo define el cambio en el potencial de membrana V con respecto al tiempo como se muestra en las ecuaciones a continuación. Consiste en un término que representa la corriente capacitiva y una colección de términos que representan las corrientes iónicas de un tipo específico de célula. Esta función en su forma general para n corrientes iónicas es:

$$
\begin{aligned}
C\frac{\delta V}{\delta t} &= \sum_{i=1}^{n} \bar{g} x_i^p y_i^q (V - V_i) + I_{appl}(t)\frac{\delta x_i}{\delta t} \\
&= \frac{x_{i\infty}(V) - x_i}{\tau_{x_i}(V)} \frac{\delta y_i}{\delta t} = \frac{y_{i\infty}(V) - y_i}{\tau_{y_i}(V)}
\end{aligned}
\tag{1.11}
$$

Cuadro 1.1: Coeficiente flujo de membrana (Miura (2002)).

Donde: C= Capacitancia, V= Despolarización, g= Conductancia maximal del canal para el ion i, x_i= Variable de activación, y_i= Variable de inactivación, V_i=Potencial de Nernst para el i-ésimo ion, I_{appl}= Corriente aplicada, $x_{i\infty}$= Estado estable de activación en el potencial V, $y_{i\infty}$= Estado estable de inactivación en el potencial V, τ_{n_i}= Constante de relajación de tiempo en el potencial V para x_i ó y_i, i= $1,2,\ldots,n$.

El término V es usado el lugar de usar el potencial de membrana, pero su significado es similar, ya que es definido como la diferencia entre el potencial de membrana y el potencial de referencia (Miura (2002); Mallot (2013); Gerstner et al. (2014)). Hodgkin y Huxley usaron esta ecuación para reproducir el potencial de acción del axón gigante de calamar como se muestra a continuación.

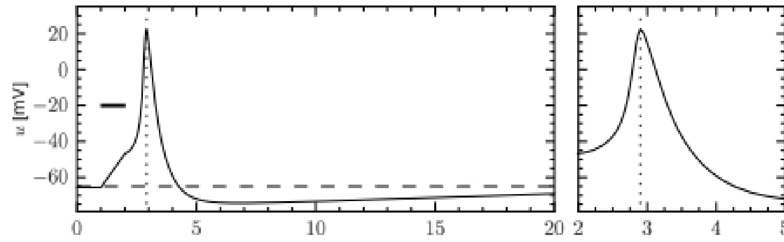


Figura 1.3: Potencial de acción (Johnston and Wu (1995)).

En la Figura 1.3 observamos que con un pulso de corriente corto (1 ms de duración). La espiga, como también se le conoce al potencial de acción alcanza una amplitud de 100 mV. y un ancho de espiga a su máxima altura de 2.5 ms, después de eso el potencial decae por debajo del potencial de reposo y regresa de una forma lenta al mismo que es -65 mV. El modelo de deriva-difusión alcanza los mismos objetivos al considerar los mismos mecanismos de suma de corrientes pertenecientes a la célula analizada. Al trabajo realizado por Hodgkin y Huxley en 1952, le siguen análisis y correcciones al sistema de Cole, Antosiewicz y Rabinowitz (Cole et al. (1955)) finalizando con un sistema de cuatro ecuaciones diferenciales, el cual se muestra a continuación[1]:

$$I = C\hat{V} + \bar{g}_{Na} m^2 h(V+115) + \bar{g}_K n^4 (V-12) + \bar{g}_L (V+10.5989) \qquad (1.12)$$

$$\begin{aligned} \bar{m} &= \phi[(1-m)\alpha_m(V) - m\beta_m(V)] \\ \bar{h} &= \phi[(1-h)\alpha_h(V) - h\beta_h(V)] \\ \bar{n} &= \phi[(1-n)\alpha_n(V) - n\beta_n(V)] \end{aligned} \qquad (1.13)$$

Donde: C= Capacitancia, I= Densidad de la corriente de la membrana, positiva de entrada, V= Diferencia de potencial de la membrana, externa con relación a la interior, m= Activación de sodio, h= Inactivación de sodio, p=activación de potasio (m,h,p sin dimensión y varían entre 0 y 1), ϕ= Coeficiente de conductancia para la ecuación a temperatura estándar de 6.3 grados centígrados.

1.9. Reducción a 2 dimensiones (modelo de Rinzel)

Los métodos computacionales utilizados para la solución del sistema de ecuaciones de Hodgkin y Huxley (entre los que se encuentra el método Runge-Kutta el cual

[1] Es importante recordar que la temperatura se asume como un valor constante.

explicaremos más adelante), mostraron que el modelo resulta computacionalmente demandante por el tiempo requerido para encontrar las soluciones. Es así como Fitzhugh (1960) propone una reducción del sistema a dos ecuaciones diferenciales ordinarias. Y después J.Rinzel (1984-85) reemplaza las variables n y $-h$ del modelo del H&H, por un nuevo valor W proponiendo un nuevo sistema de ecuaciones para modelar la excitabilidad celular con sus variables combinadas las cuales son explicadas a continuación:

$$C_m \frac{\delta V}{\delta t} = I - \bar{g}_{Na} m_\infty^3(V)(1-W)(V-V_{Na}) - \bar{g}_K (W/s)^4 (V-V_K) - \bar{g}_L (V-V_L) \tag{1.14}$$

además:

$$\frac{\delta W}{\delta t} = \frac{\phi[W_\infty(V) - W]}{\tau(V)} \tag{1.15}$$

Donde: n= W/s, h= $1-W$, W= $s[n+s(1-h)/1+s^2]$, s= $(1-h_0)/n_0$, h_0, n_0= valores en reposo de h y n respectivamente, m_∞= Rinzel considera el estado de activación del sodio estable con respecto al cambio de valor de V, entonces el valor m original lo aproxima con este valor.

En palabras de Rinzel "Más allá de usar prejuicios en modelos reducidos, uno puede de manera más fácil revelar la esencia biofísica-matemática de la excitación y oscilación de la membrana. Para formular modelos reducidos uno debe reconocer y explotar las diferencias entre escalas de tiempo entre variables, así como las similitudes entre los roles funcionales" (Av-Ron et al. (1991)).

1.10. Modelo de deriva-difusión

Con los modelos anteriormente descritos, el total de la corriente a través de la membrana es la suma de las corrientes generadas en los canales. Estas barreras o corrientes son modeladas como producto de una conductancia, y una función lineal que describe el potencial de membrana. A partir de ahora se les llama modelos "basados en conductancia" (CB). No obstante, el mecanismo de transporte de los iones a través de los canales se describe por la deriva eléctrica y el fenómeno de difusión. Sin embargo en los modelos CB el mecanismo de transporte de los iones sólo está en función de la deriva eléctrica (Herrera-Valdez (2012)).

Es por eso que utilizando una derivación de la ecuación de Nernst-Planck se genera un nuevo tipo de descripción de estas corrientes, donde se toma en cuenta la deriva eléctrica y la difusión. Estos modelos son llamados: "deriva-difusión" ó *drift-diffusion* (DD). Uno de los fenómenos que puede reproducir DD en contraste con CB

es el fenómeno de la rectificación de un canal, lo que permite considerarlo más apegado al comportamiento de las células excitables (Herrera-Valdez (2018b)).

El fenómeno de rectificación ocurre en diferentes tipos de canales, es cuando la conductancia de un canal en función del voltaje no es lineal. El modelo DD es computacionalmente más eficiente, al requerir un número menor de cálculos para lograr una simulación aceptable y significativa (Herrera-Valdez (2012)). En particular el modelo utiliza el modelo descrito en el trabajo de Goldman (Figura 1.2), que define a los canales iónicos como conjunto de poros, la existencia de un diferencial entre cargas dentro y fuera de la célula. Por tanto, la corriente de cada canal para un conjunto de iones de tipo S es igual a:

$$I = \bar{a}_s T \sqrt{S_e S_i} sinh[\frac{z_s(v - v_s)}{2v_B}] \quad (1.16)$$

Donde: I=Corriente, v= potencial de membrana, v_s=potencial de Nernst, z_s=valencia, S_e, S_i=concentración del ion S extracelular e intracelular, a= constante de aproximación a una función que depende de las propiedades del poro, campo electromagnético, movilidad de S, etc, v_B= Kt/q_e, k constante de Boltzmann, q_e carga elemental, T= temperatura absoluta, $sinh$= seno hiperbólico (Johnston and Wu (1995), Weiss (1996), Herrera-Valdez (2012)).

Ahora se agrega un factor de apertura para el canal representado en 0 ó 1 y una representación del número de canales.

$$I = N_s p_s \bar{a}_s T \sqrt{S_e S_i} sinh[\frac{z_s(v - v_s)}{2v_B}] \quad (1.17)$$

Donde: N_s= Número de canales en la membrana, p_s= Factor de apertura del canal.

Para la constante de aproximación, se considera constante la temperatura absoluta y la concentración transmembranal por lo tanto la función utilizada es (Herrera-Valdez (2018b)):

$$\bar{a}_S = \bar{a}_s N_s T \sqrt{S_e S_i} \quad (1.18)$$

Donde: N_s= Número de canales en la membrana, p_s= Factor de apertura del canal.

Entonces para representar el potencial de membrana, se toma parte de lo utilizado en modelos CB, y tenemos una función del tipo (Herrera-Valdez (2018b)):

$$C\frac{\delta v}{\delta t} = I_S - I_N - I_K - I_L \tag{1.19}$$

Donde: v= Potencial de membrana, C= Capacitancia de la membrana, I_x= Corrientes que provienen de cada canal.

Los modelos DD presentan muchas ventajas con respecto a los modelos CB que son más utilizados en la literatura. Ya que su precisión y veracidad permite realizar mediciones directamente desde las corrientes sin realizar cálculos extra, debido a que los coeficientes ya se encuentran en unidades de corriente y esto beneficia el traslado a su programación para usar en un equipo de cómputo (Herrera-Valdez (2014)).

1.11. Modelo a implementar

El modelo que se implementó utiliza una ecuación general de flujo que considera (Herrera-Valdez (2018a))[2]:

- Transporte activo a través de bombas y pasivo a través de canales.
- El transporte de iones y moléculas neutras.
- Flujo bidireccional y asimétrico.

El flujo de moléculas cargadas a través de la membrana (Φ) es,

$$\Phi = r[exp[b(\frac{\eta v - v_o}{v_T})] - exp[(b-1)(\frac{\eta v - v_o}{v_T})]] \tag{1.20}$$

Donde: r=velocidad a la que se lleva a cabo el transporte ($\frac{moleculas}{ms}\mu m^{-2}$), b=factor de que determina la dirección del flujo, $1/2$ significa que es simétrico, η=número neto de cargas que se mueven a través de la membrana, si $\eta < 0$, una carga positiva se mueve hacia adentro o negativa hacia afuera y si $\eta > 0$, una carga positiva se mueve hacia afuera o una negativa hacia adentro, v=voltaje, v_o=potencial de Nernst por el número de cargas que se mueven a través de la membrana, v_T=constante universal de los gases R multiplicado por la temperatura del sistema en Kelvin dividido por la constante de Faraday.

[2] Basado en modelos termodinámicos que consideran barreras energéticas en la membrana.

Para una célula el cambio en el voltaje se da por la suma de sus corrientes, como ya se describió en los modelos H&H y Rinzel, descrita (Herrera-Valdez (2018b)):

$$\delta_T v = -\sum_{l=1}^{M} a_l p_l \varphi_l(v) \tag{1.21}$$

Donde: φ=flujo a es la densidad, la cual se relaciona con el número de canales en la membrana, p=es la porción de canales abiertos que depende de la activación e inactivación de los mismos, a= se refiere a la cantidad de canales en la membrana.

De la misma manera que en el modelo de Rinzel la activación de canales de potasio y la inactivación de los canales de Sodio se puede considerar un solo parámetro (Herrera-Valdez (2018b)):

$$\delta_T w = w[F_w(v) - w]S_w(v) \tag{1.22}$$

Donde: v=voltaje actual, w=valor actual de la variable de recuperación.

F y S son funciones que describen la activación de canales de acuerdo a la fuerza electromotriz y la rectificación.

$$F_u(v) = \frac{exp[g_u(\frac{v-v_u}{v_T})]}{1+exp[g_u(\frac{v-v_u}{v_T})]}, u\varepsilon[m,w] \tag{1.23}$$

Donde: v=valor de voltaje, v_u=voltaje al cual la mitad de los canales se encuentran abiertos, v_T=constante universal de los gases S multiplicado por la temperatura del sistema en Kelvin dividido por la constante de Faraday, g_u=ganancia.

$$S_w(v) = r_w[exp[b_w g_w(\frac{v-v_w}{v_T})] - exp[(b_w-1)g_w(\frac{v-v_w}{v_T})]] \tag{1.24}$$

Donde: v=valor de voltaje, v_w=voltaje al cual la mitad de los canales se encuentran abiertos, v_t=constante universal de los gases S multiplicado por la temperatura del sistema en Kelvin dividido por la constante de Faraday.

Además se puede considerar la dinámica intracelular de Calcio c que es gual a (Herrera-Valdez (2018b)):

$$\delta_T c = r_c(c_\infty - c) - k_c[\text{ corriente total de calcio}] \tag{1.25}$$

Donde: r_c=velocidad con la que la concentración de calcio regresa a su estado estable, c_∞=estado estable del calcio, c=concentración de calcio intracelular, k_c=impacto del flujo del calcio en concentración intracelular.

Este modelo al igual que los otros que se han revisado requiere de la integración de las diferentes variables en función del tiempo por métodos numéricos. Éstos se revisan a continuación (Herrera-Valdez (2018a)).

1.12. Métodos numéricos de solución de ecuaciones diferenciales: ventajas computacionales de implementación

Las ecuaciones diferenciales ordinarias surgen como una técnica de modelado matemático para describir un problema adaptable a distintas disciplinas como la ciencia, economía, ingeniería, etcétera. Como en muchos casos, la complejidad no permite un cálculo a mano del modelo, por lo tanto, para obtener un resultado fiable y eficiente, es necesario el uso de una simulación por computadora (Ascher and Petzold (1998)).

La derivada, utilizada para el cálculo de una tangente en un punto, es útil para resolver el siguiente problema. Si un fenómeno puede expresarse utilizando el cambio de estado entre un conjunto de variables, entonces podemos utilizar una o varias funciones para describir los cambios de estado de una condición a otra, al conjunto de ecuaciones resultante se le denomina ecuación diferencial. Cualquier función definida en un intervalo que permite resolver la ecuación diferencial se puede considerar una respuesta para ésta (Becerril Espinosa and Elizarraraz Martínez (2004)).

Existen dos tipos de ecuaciones diferenciales, si la ecuación tiene una o un conjunto de derivadas formuladas con respecto a una sola variable independiente entonces se le cataloga como ordinaria, siendo estas las más utilizadas para la implementación en cómputo. Por otra parte, si se encuentra que la ecuación tiene más de una variable independiente se le cataloga como parcial (Kaw (2011a)).

Al generar un modelo con una ecuación esta debe permitir dar una interpretación física, por lo tanto hay que tener en cuenta las siguientes fases de uso de una ecuación (Kaw (2011a)):

- Formular una ecuación diferencial a partir de un fenómeno físico.

- Resolver la ecuación diferencial dado un conjunto de condiciones y observar el cambio de las variables y la repercusión de las constantes.
- Interpretar los resultados para implementar de manera física.

Además de dividir las ecuaciones diferenciales a partir del número de incógnitas, también podemos catalogarlas por la cantidad de datos o características antes de su análisis, una de estas categorías, es la referente al problema del valor inicial, los problemas de este tipo son aquellos que se tiene una ecuación de la forma:

$$\frac{\delta y}{\delta x} = f(x,y)$$

donde:
$y(0)= y_0$.

En el problema del valor inicial se han utilizado diversas técnicas de resolución del problema buscando siempre el método que brinda mejores aproximaciones (Kaw (2011a)). En los modelos de células excitables se toman ecuaciones diferenciales ordinarias en donde la variable independiente es el tiempo como se muestra en la sección anterior para los diferentes modelos celulares. De tal suerte que se puede resolver el sistema de ecuaciones de manera relativamente sencilla.

1.13. Método Runge-Kutta

El Método Runge-Kutta es un método multipaso basado en método de Euler (Kaw (2011b), Iserles (2012)), pero que mejora la precisión del resultado usando cálculos previos para determinar el valor de la integral. El método más utilizado dentro de los métodos Runge Kutta es el de cuarto orden o Runge-Kutta 4 que se define así:

Para una ecuación diferencial de primer orden en donde conocemos el valor inicial, podemos calcular el valor de la integral en el paso inmediato siguiente con un paso de tamaño h mayor que cero pero pequeño, de la siguiente manera (Iserles (2012)):

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)h \tag{1.26}$$

Donde: h=Tamaño de paso, y_i=Valor de paso anterior.

Así como:
k_1=$f(x_i, y_i)$.
k_2=$f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1 h)$.
k_3=$f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2 h)$.
k_4=$f(x_i + h, y_i + k_3 h)$.

La idea es que para calcular el siguiente valor y_{n+1} se obtenga el resultado de los valores intermedios k_{1-4} que después se suman como se muestra en las ecuaciones de arriba (Kaw (2011b), Iserles (2012)). Debido a que hace estos cálculos intermedios, Runge-Kutta es un método que brinda una precisión aceptable para dar la respuesta más cercana de los estados del fenómeno descrito por las ecuaciones diferenciales de primer orden formuladas.

1.14. Programación Orientada a Objetos

La programación Orientada a Objetos es un paradigma de programación en lenguajes de alto nivel que se basa en la abstracción (Husband, Mark et al. (2008)).

En lugar de programar de manera lineal como en el Paradigma Estructurado, el código se organiza por entidades. Una entidad básica es un objeto, a este se le asignan características cuyo valor se conserva de manera local (Husband, Mark et al. (2008)). Las propiedades de los objetos se conservan de manera uniforme, lo que permite crear diferentes objetos con las mismas características pero con diferente valor para cada una de ellas (Stefik and Bobrow (1985)).

Los valores (datos) con características en común se agrupan en conjuntos denominados "tipos". Por ejemplo, el tipo *int* (entero), *double* (números reales), *char* (una letra), *string* (una palabra o frase). Las variables son colecciones de tipos con un identificador (nombre) necesario para su manipulación. Con éstos se pueden describir atributos que describen al objeto (Kelly (2016)).

El concepto de "clase" es usado en el Paradigma Orientado a Objetos como el lugar donde un objeto es definido, su manipulación está delimitada a sus propiedades y atributos, a partir de subrutinas que se definen como "métodos" (Summerfield (2008)).

El objeto puede o no necesitar variables para su creación, lo importante es que en este espacio se puede describir el alcance de las variables que la conforman, así como la respuesta interna y externa que debe seguir dentro de la ejecución (Summerfield (2008)). Üna clase entonces contiene las descripciones de todos los comportamientos de los objetos que representa. En términos de las ciencias de las computación, llamamos a estos comportamientos individuales de la clase como sus métodos"(Husband, Mark et al. (2008)).

Hay propiedades adicionales que pertenecen al paradigma, pero no se encuentran implementadas en todos los lenguajes que lo manejan como: el polimorfismo, capacidad de diferentes clases u objetos de responder al mismo conjunto de reglas o comportamiento (protocolo) y la herencia, capacidad que tienen objetos parecidos de compartir características (Summerfield (2008)). Estas propiedades permiten mantener programas cortos y organizados.

Un ejemplo, es la herencia jerarquizada donde una clase u objeto es definida en términos de una sola superclase, a partir de esta podemos agregar o quitar características, agregar variables y propiedades independientes de lo que contenía en la superclase (Summerfield (2008)).

Tener como base teórica de programación el Paradigma Orientado a Objetos para resolver este problema en particular, permitió potenciar y delimitar tanto las propiedades como las acciones de cada uno de los agentes que participan en el fenómeno que se describe.

Para cumplir con el objetivo de esta tesis, se generó la interacción de los objetos con una interfaz gráfica de usuario (Graphic User Interface). Ésta se controla tomando como base el Modelo Vista Controlador.

1.15. Modelo-Vista-Controlador

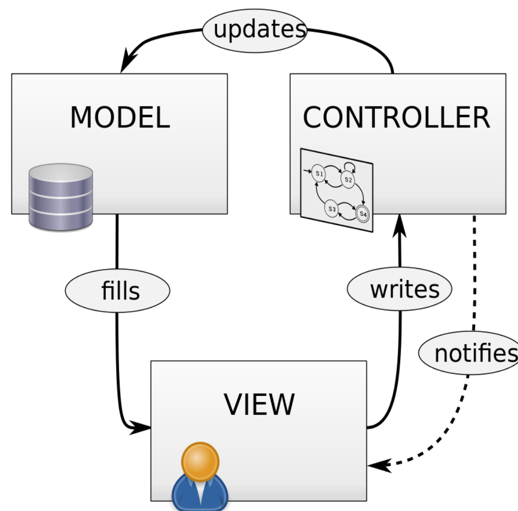


Figura 1.4: Representación Modelo Vista Controlador (Wikimedia (2019)).

El Modelo Vista Controlador permite establecer la comunicación de manera ordenada y clara entre un modelo y un usuario. Esto simplifica el desarrollo, prueba y mantenimiento futuro del sistema. Su aplicación puede ir desde sistemas aislados hasta aplicaciones complejas (Bucanek (2009)). La filosofía deja al final el espacio suficiente para generar una rúbrica de estilo en la aplicación (modelo) y al hacer actualizaciones de puedan tomar las decisiones necesarias que se adecuan al proyecto en beneficio del impacto en el desarrollo del proyecto (Freeman (2015)).

A continuación se describen cada uno de los elementos del Modelo Vista Controlador (Kelly (2016)):

Un modelo representa los datos o atributos asociados con los objetos, comúnmente se compone de pocos métodos que se relacionan con guardar o serializar los datos.

La vista es la representación directamente visual o descripción representativa de los modelos. Esto se adapta al tipo de modelo utilizado.

El controlador es el enlace entre la vista y el modelo. A un requerimiento de la vista el controlador actualiza el modelo para presentar un nuevo estado, es decir un nuevo conjunto de valores.

Las siguientes conceptos, ayudan a comprender mejor (*):

- Los datos contenidos en los objetos del Modelo encapsulan información.
- Los objetos en la Vista muestran información al usuario.
- Los objetos en el Controlador implementan acciones
- Los objetos en la Vista actualizan su estado cada que los datos de los objetos del Modelo cambian durante la ejecución.
- Los objetos en la Vista permiten el paso de datos de entrada y los datos son trasferidos a los objetos del Controlador que realizan las acciones.

Un buen análisis acerca del rol de los objetos es importante para la implementación (Bucanek (2009)).

La forma en que se comunican todas las partes para el intercambio de información o señales de inicio y término es un aspecto más a considerar durante el diseño (Bucanek (2009)).

Se debe considerar que “un método de diseño que sea muy simple puede fallar en resaltar las propiedades claves de los objetos en el sistema. Aunque un problema persistente en los métodos actuales es la complejidad que puede resultar de su implementación. Un método de diseño necesita ser simple e intuitivo para seguir la razón del cambio de un valor con un simple seguimiento. De ser complicado, diseñadores y programadores no lograrán trabajar en conjunto para conseguir un buen proceso repetitivo.” (Ortega Arjona (2005)

1.16. Python: ventajas para el proyecto

Python es un lenguaje de programación de alto nivel creado en 1989 por Guido Van Rossum en Stichting Mathematisch Centrum en Holanda surge como una herramienta popular para las ciencias, es moderna, fácil de aprender y originalmente utilizaba el Paradigma Orientado a Objetos (Severance (2015)).

Es un lenguaje interpretado, por lo que se puede revisar a simple vista y en sesiones interactivas. Aunque la falta de elementos sintácticos que se utilizan en otros lenguajes como corchetes ($[], \{\}$) o punto y coma (;) puede ser difícil de entender la primera vez que se estudia, sin embargo una vez superada esta etapa de aprendiz se encontrará con un lenguaje limpio en su forma de escribir y leer (Severance (2015)).

Python es un lenguaje conciso, es decir, que con un pequeño número de líneas escritas se obtiene un programa rápidamente, lo que en otros lenguajes lograría con un número considerable de líneas (Summerfield (2008)). Otra ventaja es la modularidad del lenguaje, que permite agregar bibliotecas del mismo lenguaje e integrar diferentes lenguajes de programación y sus componentes, para lograr sistemas de mayor tamaño y complejidad que permite diferente tipo de análisis (Muller et al. (2015)).
Además, el intérprete de Python cuenta con una estructura de capas interactiva que facilita la prueba de código por segmentos antes de agregarlos en la aplicación final, y se puede agregar variables a *clases* al momento de la ejecución.
El soporte nativo de diferentes paradigmas como lo son el imperativo, Orientado a Objetos y funcional permiten aplicar el mejor estilo de solución problema durante el desarrollo del proyecto (Langtangen (2004)). Existe una cantidad considerable de bibliotecas que aprovechan la modularidad, cuenta con una base de programadores que cooperan en el desarrollo del lenguaje y corrigen problemas de programación en un tiempo corto. Bibliotecas como NumPy han alcanzado la popularidad y fiabilidad necesaria dentro de la comunidad científica.

Por último, su vasta documentación permite entender el funcionamiento de los módulos para poder ser agregados a los proyectos, desde cómo trabaja la biblioteca básica de Python, hasta enlaces a funcionalidades para una tarea específica, aunque es cierto que hay material bibliográfico físico para su consulta, es recomendable buscar en Internet las adecuaciones y cambios más recientes (Summerfield (2008)).

Python convive con otro lenguaje de programación, MATLAB, en el cual científicos tienen ya importantes cantidades de código escrito. Python aunque intenta ganar adeptos además de las características antes mencionadas es un lenguaje gratuito. Permite el desarrollo de nuevos módulos enfocados en emular servicios ofrece MATLAB. Incluso cuenta con un intérprete que permite ejecutar código escrito en MATLAB (Muller et al. (2015)).

1.17. Interfaces gráficas con pyQt: adaptarse al usuario final

Las Interfaces Gráficas para Usuarios (GUI) es un conjunto de objetos llamados *widgets*, cada uno de éstos puede ser un botón, un texto, deslizadores, imágenes, por mencionar algunos. Están ordenados de forma jerárquica y en conjunto representan una ventana completa de nuestra aplicación.

Cada GUI es un ciclo que se encuentra a la espera de la aparición de eventos para ejecutar procesos. Los eventos se capturan con dispositivos de entrada, teclado, ratón, cámara, entre otros. Con esto se establecen límites, entre lo que un usuario puede controlar y lo que no debe tener injerencia para no afectar el flujo de la ejecución (Langtangen (2004)).

Qt es una aplicación para el desarrollo de interfaces gráficas escrita originalmente en C++. Tiene la ventaja de generar proyectos con soporte para correr en una variedad de plataformas entre las más populares encontramos Windows, Linux, Mac OS X, además de plataformas basadas en Unix (Summerfield (2008)). La aceptación de este tipo de bibliotecas es tal, que existen programas que sin tener que escribir una línea de código, solo de manera gráfica, permiten diseñar el aspecto, tamaño y posición de todos los *widgets*. Esto permite invertir tiempo de programación hacia otras necesidades.

Al usar Python dado este conjunto de utilidades, se obtiene pyQT, nombre de la biblioteca encargada de generar ventanas, botones, textos, entre otros, para la creación de interfaces gráficas. De modo que tenemos como resultado un conjunto de herramientas provistas por el lenguaje a utilizar para el desarrollo dinámico de diferentes tipos de aplicaciones en diferentes tipos de disciplinas (Summerfield (2008)).

El uso de esta plataforma, el lenguaje Python, el Modelo Vista Controlador y el Paradigma Orientado a Objetos permite implementar el modelo matemático-biológico de manera completa, flexible y facilita el uso del programa por parte de usuarios no familiarizados con la computación. También se puede hacer una extensión del mismo para realizar implementaciones más complejas como redes neuronales dejando de lado la interfaz gráfica.

1.18. Método Runge-Kutta contra biblioteca OdeInt

Como característica común, tanto el método como la biblioteca, son hechos para resolver el problema del valor inicial (mencionado en sección 1.12). La biblioteca OdeInt resuelve el sistema de ecuaciones utilizando el algoritmo LSODA (Scipy (2019)), este algoritmo permite la resolución de sistemas *dy/dt=f* reajustando el método de resolución dependiendo del punto de la curva en la que se encuentre, en consecuencia se reajusta el tamaño del paso a dar, lo que representa una mejor precisión en los resultados del cálculo del área bajo la curva resultante, lo que en este tipo de curvas solución puede convertirse en una desventaja (potenciales de acción), ya que es proclive a regresar un mensaje de error si no puede estimar el tamaño correcto del paso y método para calcular ese intervalo específico (Mulansky and Ahnert (2014) y Stepleman (1983)).

El desarrollo, implementación y ejecución del método Runge-Kutta nos permite contar con un algoritmo que nos resuelve un intervalo de la gráfica de manera cons-

tante, con una buena certeza en el resultado, además sin tener la necesidad del uso de criterios ambiguos o que deriven en un mismo tipo de error que tienen la biblioteca OdeInt (Mulansky and Ahnert (2014)).

Durante este capítulo se describieron fenómenos físicos con el uso de herramientas matemáticas, el uso de técnicas para adaptar estas soluciones en un equipo de cómputo, así como los recursos computacionales que toman ventaja de la implementación para realizar cálculos que permitan mostrar resultados que, aunque los números resultantes contienen un error, la combinación de estas definiciones dejan estos en rangos mínimos y manejables para su uso.

Capítulo 2: Desarrollo del sistema computacional.

Este capítulo tratará sobre el análisis realizado para estructurar y escribir los componentes a utilizar en el proyecto. Mientras se avance también se mostrará un descripción de la organización de los archivos y de cada archivo individual para lograr la modularidad y replicación de los códigos generados.

2.1. Antecedentes del proyecto

La inclusión del análisis computacional de los modelos matemáticos se remonta desde el modelado de Hodgkin y Huxley en 1952 (Hodgkin and Huxley (1952)), para células individuales, ésto llevó a los investigadores a implementar a partir de este modelos de redes neuronales como el modelo propuesto por Donald Hebb en los años cuarenta en su libro “The Organization Behaviour”, logrando una implementación exitosa en el Instituto Tecnológico de Massachusetts (MIT) en 1954.

A la fecha de escritura de este trabajo, hay al menos cuatro entornos de modelado de redes notables, éstos son Eilif Muller (Muller et al. (2015)) y Ruben A.Tikidji-Hamburyan (Tikidji-Hamburyan et al. (2018)):

- NEURON (hecho por la universidad de Cambridge)
- GENESIS
- BRIAN
- NEST(realizado por el llamado Human Brain Project)

Todos de alguna forma u otra aprovechan nuevas herramientas computacionales como objetos, concurrencia , ejecución multiprocesador, ejecución en cluster e interfaces gráficas (Tikidji-Hamburyan et al. (2018)).

Una forma de ganar adeptos entre los programadores de algún lenguaje de programación es el presentar cierta compatibilidad con otros lenguajes ya usados o populares entre la comunidad de programadores. Un ejemplo de esto es la hecha por

parte del software NEURON con el lenguaje de programación Python para crear simulaciones de comportamiento de neuronas sin la necesidad de utilizar la interfaz gráfica que tiene el software (Hines et al. (2009)) o como un receptor de parámetros como es con el software pyNEST que pretende ser una interfaz para Neural Simulation Technology (NEST) (Muller et al. (2015)).

Esta área actualmente se encuentra en continuo desarrollo lo que no permite listar todos los trabajos sobre el tema pero sirvan estos para tener en cuenta los más destacados que utilizan los trabajos hechos a lo largo del tiempo por los investigadores (Muller et al. (2015)).

2.2. Lista de requerimientos

Para poder realizar un buen desarrollo del proyecto, se utilizarán herramientas adecuadas para, describir los alcances deseables para la implementación. La primera, aprovechando la organización descrita para el Modelo Vista Controlador para su organización, es la lista de requerimientos. Establecer en requerimientos permite describir la abstracción hecha por el programador de que debe hacer el sistema con una descripción de la funcionalidad y características del producto final. Con el objetivo que el usuario, tenga un lugar donde validar y comprobar si las respuestas a su problemática son resueltas antes de avanzar en la etapa de implementación (Gómez (2011)). Este paso tiene un mayor impacto en el avance del proyecto, si, aprendiendo de implementaciones anteriores y como regla de modo empírico, el costo de reparar un error se incrementa en un factor de diez de una fase de desarrollo a la siguiente, por lo tanto la preparación de una especificación adecuada de requerimientos reduce el riesgo general asociado con el desarrollo (Norris and Rigby (1994)). Entonces en el Cuadro 2.2 se muestra la tabla final con los requerimientos para este sistema en específico.

Número	Requerimiento
	El modelo:
1	Ocupará las ecuaciones descritas en Herrera-Valdez (2018b).
2	El sistema de ecuaciones debe incluir el cálculo de los valores de voltaje(V), la variable de recuperación(W) y si aplica la concentración intracelular de Calcio(c) en función del tiempo (simulación).
3	Debe existir un lugar dentro del código donde los valores a utilizar se puedan editar.
4	Los resultados de las simulaciones deben poder graficarse.

	La vista:
5	Debe presentar todos las variables disponibles para editar.
6	Mostrará cuales son los rangos máximos y mínimos para las variables a editar.
7	La descripción y unidades de cada variable debe describirse.
8	Mostrará claramente que valores pertenecen a cada objeto que interactúa en el modelo.
9	Permitirá cargar y guardar datos producidos en la vista.
10	Tendrá una división clara de que tipo de célula se está manipulando.
	El Controlador:
11	Conectará los botones creados en la vista con métodos que propios del modelo.
12	Tendrá los métodos necesarios para validar las variables en la vista antes de enviarlas al modelo.
13	Habilitará o deshabilitará secciones de la vista para evitar errores en los usuarios.
14	Podrá reiniciar los valores de la vista.
15	Establecerá los valores predeterminados de la vista en la primera ejecución del programa.

Cuadro 2.2: Lista de requerimientos.

2.3. Diagrama de casos de uso

El siguiente objetivo, es tener una descripción de la interacción que deben tener los usuarios con el software para lograr la realización de una tarea en específico. El Diagrama 2.5 señala hacia que actor de la interacción corresponde el paso para cumplir la meta establecida con lo que se logran principalmente (Torlak (2016)):

- Tener un consenso entre los usuarios y los desarrollos del sistema acerca de los requerimientos (establecer que es un caso de éxito).

- Alertar a los desarrolladores acerca de las situaciones problemáticas, así como casos que llevan a un error o que sirven para probar el sistema.

- Saber si se cumplen los niveles de funcionalidad que se habían establecido como meta a cumplir.

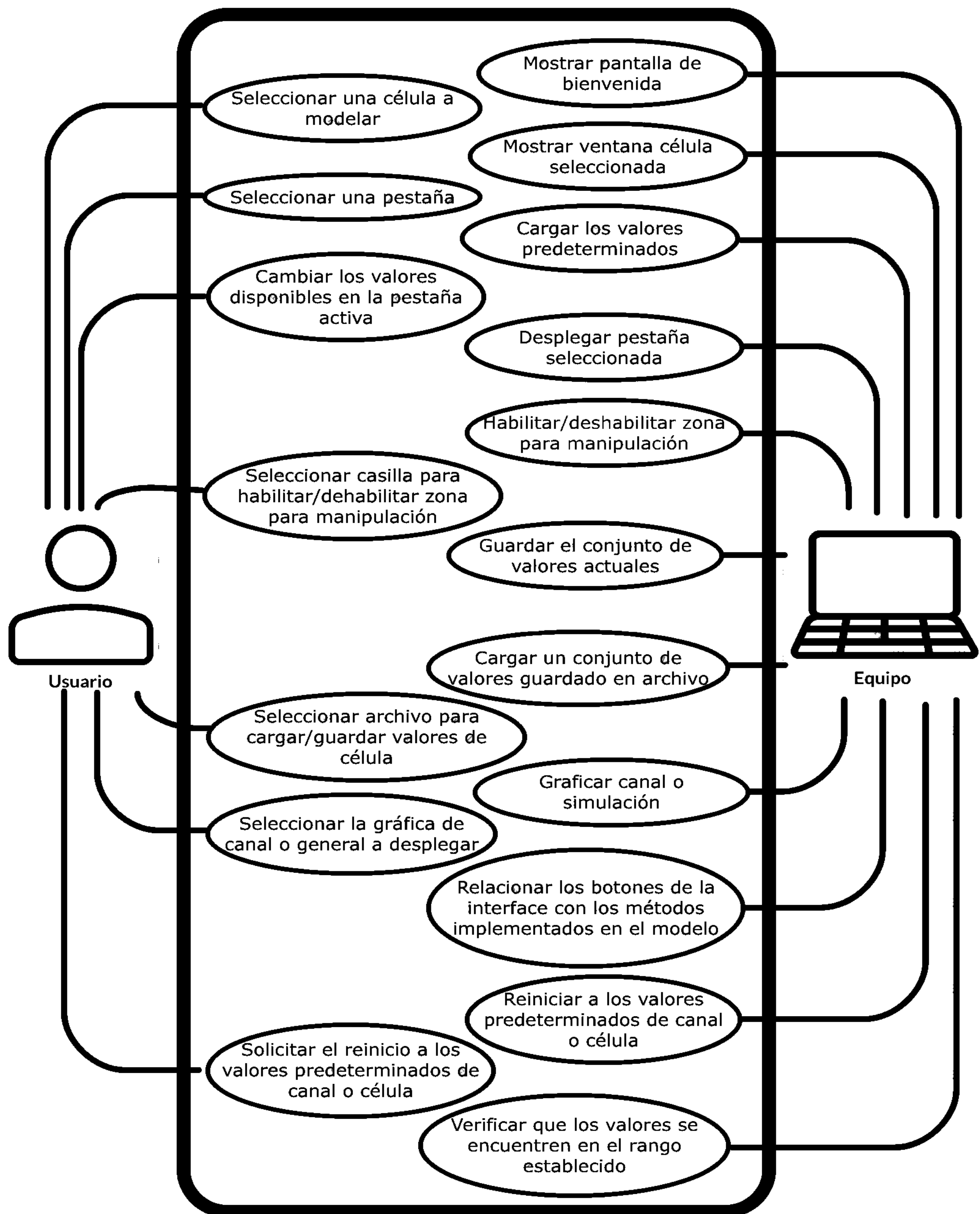


Figura 2.5: Diagrama casos de uso de la célula HH.

Una vez que la abstracción del problema ha llegado a un punto en que el proyecto cumple las necesidades mínimas de ejecución, es momento de describir específicamente como se va a implementar todo este diseño en archivos que van a formar parte del producto final.

2.4. Descripción teórica de la implementación

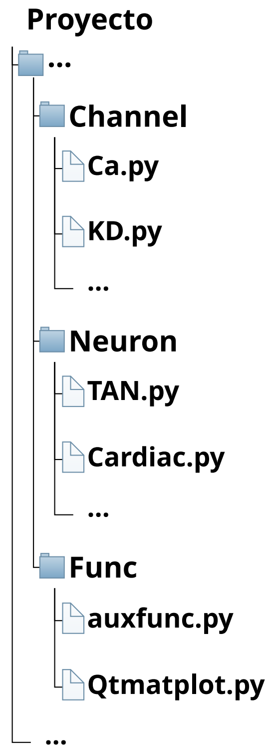
El objeto *canal* es el análogo *in sílico* de los canales y bombas que se encuentran en una célula. Estos poseen características como amplitud, rectificación (s), métodos y funciones. El objetivo es crear una biblioteca que aloje los diferentes canales descritos en la literatura y que a su vez se puedan crear nuevos objetos *canal*, en caso de que se quiera modelar un *canal de novo* es decir, un canal producto de nueva investigación. Cada *canal* tiene los parámetros relevantes para el mismo y éste se encarga de llamar a las funciones generales almacenadas en otra instancia del código.

La *célula* constituye un segundo tipo de objeto, contiene la descripción de las células/neuronas. Cada *célula* tiene un conjunto de objetos del tipo *canal* específico, que llama para su ejecución, además de otras características propias como la concentración de iones y la temperatura.

También existen algunos métodos y algoritmos que no pertenecen ni a las células, ni a los canales, pero si nos permiten obtener valores de cualquiera de estos objetos. Son parte de la literatura física, biológica y matemática. Por ejemplo el cálculo de potencial de Nernst, la fórmula de flujo o el método Runge-Kutta para integrar el sistema de ecuaciones diferenciales. Estas funciones se encuentran en una instancia separada a los objetos descritos.

2.5. Estructura lógica de los archivos del proyecto

El Cuadro 2.3 muestra la organización de archivos para el proyecto. La primera carpeta es la denominada "*Channels*", que contiene a los objetos canal. Dentro del objeto se encuentra la función matemática para calcular la corriente que pasa por el mismo, así como los parámetros biofísicos que lo caracterizan y diferencian de otros canales. Cada archivo de extensión "*.py*" pertenece a un canal diferente. Las células se encuentran en la carpeta "*Neuron*" contiene descripción de las células/neuronas. De la misma manera, cada archivo muestra un tipo de célula diferente. Y en la parte más baja de esta sección se coloca la carpeta "*func*" con "*auxfun.py*", donde están los métodos con las funciones y algoritmos que no pertenecen ni a las células, ni a los canales además de "*Qtmatplot.py*" que permite colocar implementaciones de la biblioteca *matplot* en ventanas implementadas con la biblioteca *pyQT*.



Cuadro 2.3: Estructura de los archivos para las células

2.6. Estructura del código del proyecto

Una vez revisada la estructura de los archivos del modelo como conjunto, se presenta la organización de los archivos para su escritura (ver Cuadro 2.4).

Código *Channel*

- **Zona de bibliotecas**
- **Zona de preparación del canal**
- **Zona de presentación**

Cuadro 2.4: Estructura del código de los canales

Primero se encuentra la zona para importar bibliotecas (parte superior Cuadro 2.4 y Código 2.1 lineas 5 a 13), destacamos las desarrolladas para cómputo científico *scipy* (scientific Python línea 7), *numpy* (línea 8) y la utilizada para gráficas *matplotlib* (línea 10). Así como la biblioteca de métodos *auxfunc.py* (línea 13).

Código 2.1: Ejemplo canal proyecto.

```
1 """
2 Na.py
3 Martin Garcia Mata
4 """
5 #Zona de precargas de bibliotecas
6 #Biblioteca de computo cientifico de python
7 import scipy as sc
8 import numpy as np
9 #Biblioteca para graficas (plot)
10 from Func.Qtmatplot import Qtmatplot, QtmatplotT, QtmatplotTab
11 #Biblioteca de metodos con funciones genericas contenida en
12 #/Func/auxfunc.py
13 from Func.auxfunc import *
```

En la segunda zona (Cuadro 2.4 y Código 2.2 lineas 15 a 39) se encuentran los métodos (*init*) que realizan operaciones previas a la ejecución de la función del canal, la carga de valores necesarios proveninentes de la célula (ejemplo de estos valores, revisar capitulo *Apéndice* Código 4.10), variables derivadas que requieren calcularse a partir de valores provenientes de la célula o asignación para que estas sean compartidas con otros objetos como la célula u otros canales (el potencial de Nernst y constante de Boltzman, línea 21 método *prepair*) y también se encuentra la función que obtiene la corriente del canal (línea 31 método *I*).

Código 2.2: Ejemplo canal proyecto.

```
14 class Na():
15     #Cuando se crea un objeto en este metodo se pueden crear y
16     #asignar nuevos valores
17     def __init__(self,biophysna,v=0.1,w=0.1,c=1):
18         #threading.Thread.__init__(self)
19         self.prepair(biophysna,c=c)
20     #Metodo creacion atributos objetos a partir del biophysna
21     def prepair(self,biophysna,c=0.1):
22         for key in biophysna:
23             setattr(self, key, biophysna[key])
24     #Metodo funcion de la corriente recibe
25     #vn=voltaje de la neurona, wn=valor de recuperacion de
26     #la neurona
27     #c= concentracion de calcio de la neurona,
28     #vt=constante de los gases*t(en Kelvin)/constante de Faraday
29     #de la neurona
30     #vna=potencial de inversion del sodio calculado en la neurona
31     def I(self,vn,wn,cn,vt,vna):
```

```
32                  return (1−wn)∗F(v=vn,v0=self.halfact,
33                  n=self.gain,vt=vt)
34                  ∗S(v=vn,vr=vna∗self.nu,s=self.s,vt=vt,
35                  a=self.a,n=self.nu)
36      #valor de el metodo/funcion I dividido entre la concentracion
37      #de calcio
38      #def J(self,vn,wn,cn,vt,vna):
39          #return self.I(vn=vn,wn=wn,cn=cn,vt=vt,vna=vna)/cn
```

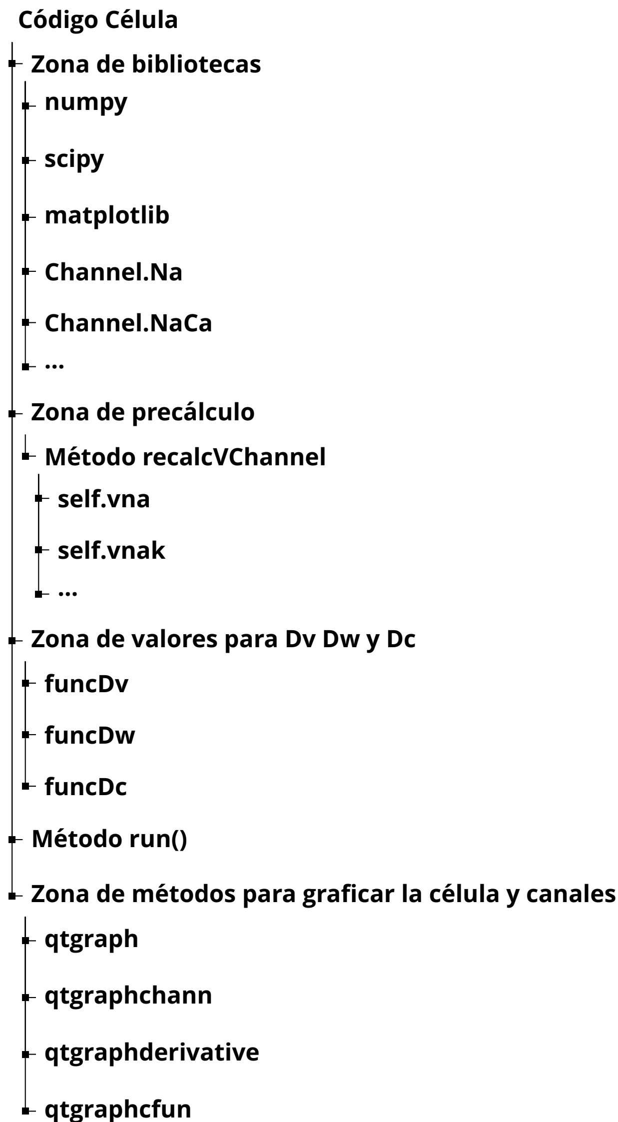
Código 2.2 (Cont.): Ejemplo canal proyecto.

La última zona es de presentación (ver Cuadro 2.4 y Código 2.3 de las lineas 40 a la 63), en esta zona se encuentran los métodos con las funciones matemáticas auxiliares y métodos para graficar de manera individual el comportamiento del canal, antes de la ejecución conjunta con el resto de los canales para evitar cálculos inútiles hasta tener los valores adecuados.

Código 2.3: Ejemplo canal proyecto.

```
40 #Metodo para graficar el comportamiento de la corriente
41      def qtgraph(self,v0,vf,step,vt,vna,opt):
42          xvect=np.linspace(start=v0,num=vf/step,stop=vf)
43          yvect=[]
44          self.windowch=Qtmatplot()
45          ax=self.windowch.figure.add_subplot(111)
46          ax.clear()
47          if opt == 0:#funcion F
48              for i in xvect:
49                  yvect.append(F(v=i,v0=self.halfact,
50                  n=self.gain,vt=vt))
51                  #plt.figure('Na funcion F')
52                  ax.title.set_text("Na␣funcion␣F")
53                  ax.set_ylabel('F') #nombre del eje y
54          if opt == 1:#funcion S
55              for i in xvect:
56                  yvect.append(S(v=i,vr=vna,s=self.s,vt=vt,
57                  a=self.a,n=self.nu))
58                  ax.title.set_text("Na␣funcion␣S")
59                  ax.set_ylabel('S') #nombre del eje y
60          ax.set_xlabel('Voltaje␣(mV)') #nombre del eje x
61          ax.plot(xvect,yvect)
62          self.windowch.canvas.draw()
63          self.windowch.show()
```

La estructura del código de las células (Cuadro 2.5) es similar a la estructura de los canales (ver *channels* Cuadro 2.4).



Cuadro 2.5: Estructura del código de la célula

En la estructura de la células (ver Cuadro 2.5 y Código 2.4 de las lineas 5 a la 22). Primero se encuentra la zona para importar bibliotecas. Al igual que en el caso de los *canales*, se importan las bibliotecas *scipy, numpy, matplotlib y auxfun.py*. Adicional a estas, se importan los canales utilizados en la célula.

Código 2.4: Ejemplo célula proyecto.

```
1 """
2 hh.py
3 Martin Garcia Mata
4 """
5 #Zona de precargas de bibliotecas
6 #Biblioteca de computo cientifico de python
7 import scipy as sc
8 import scipy.integrate
9 import numpy as np
10 #Biblioteca para abrir varias instancias en paralelo y
     concurrencia
11 import threading
12 #Biblioteca para hojas de calculo
13 import csv
14 import pandas as pd
15 #Biblioteca para graficas (plot)
16 from Func.Qtmatplot import Qtmatplot, QtmatplotT, QtmatplotTab
17 #Biblioteca de funciones genericas contenida en $/Func/auxfunc.py
18 from Func.auxfunc import *
19 #Corrientes que componen la celula contenida en $/Channel/*.py
20 from Channel.Na import *
21 from Channel.NaK import *
22 from Channel.KD import *
```

La segunda parte (ver Cuadro 2.5 y Código 2.5 de las lineas 26 a la 49) contiene los métodos de preparación de la célula, un ejemplo de el archivo con los valores predeterminados esta en el Capítulo *Apéndice* Código 4.10.

Código 2.5: Ejemplo célula proyecto.

```
23 class HH2(threading.Thread):
24     #Cuando se crea un objeto en este metodo se pueden crear
25     #y asignar nuevos valores
26     def __init__(self,biophyshh,biophysna,biophysnak,biophyskd
27     ,v=0.5,w=0.5,c=3):
28         threading.Thread.__init__(self)
29         #Este for es para crear atributos del objeto
30         #a partir de el biophys
31         for key in biophyshh:
32             setattr(self, key, biophyshh[key])
33         #creamos los atributos que llamamos "corrientes"
34         self.c=0.0#(no existente para este modelado de celula)
```

```
35            self.Na=Na(biophysna,v=self.v,w=self.w,c=self.c)
36            self.NaK=NaK(biophysnak,v=self.v,w=self.w,c=self.c)
37            self.KD=KD(biophyskd,v=self.v,w=self.w,c=self.c)
38            #Zona de precalculos
39            self.recalcVChannel()
40        #Metodo de precalculo de Vnak
41        def getVnak(self):
42            return (3*self.vna)-(2*self.vk)+(self.vATP)
43        #Metodo para valcular los voltajes de canales
44        def recalcVChannel(self):
45            self.vna=vNernst(vT=self.vt, cIn=self.inNa,
46            cOut=self.outNa, val=1.0)
47            self.vk=vNernst(vT=self.vt, cIn=self.inK,
48            cOut=self.outK, val=1.0)
49            self.vnak=self.getVnak()
```

Código 2.5 (Cont.): Ejemplo célula proyecto.

En la tercera parte (ver Cuadro 2.5 y Código 2.6 de las lineas 50 a la 100) encontramos la zona de métodos de célula, utilizados durante las iteraciones partícipes del sistema de ecuaciones diferenciales. Se recomienda que sean puestos de manera individual para su fácil lectura e interpretación. Estos métodos contienen las funciones diferenciales para calcular el voltaje (v), la variable de recuperación (w) y la concentración intracelular de Calcio (c), este último no se modela en todas las células para simplificar la simulación. Después se encuentra el método run que permite la ejecución conjunta de todos los métodos, el motivo de su nombre es dejarlo listo para la futura implementación con hilos y lograr la ejecución simultánea de objetos y así modelar redes celulares.

Código 2.6: Ejemplo célula proyecto.

```
50 #Zona de ecuaciones diferenciales de la celula
51        #Cambio en el voltaje=suma de las corrientes /capacitancia
52        #de la membrana
53        def funcDv(self,t,v,w,c):
54            IK=self.KD.I(vn=v,wn=w,cn=c,vt=self.vt,vk=self.vk)
55            INa=self.Na.I(vn=v,wn=w,cn=c,vt=self.vt,vna=self.vna)
56            INaK=self.NaK.I(vn=v,wn=w,cn=c,vt=self.vt,vnak=self.vnak)
57            return (-INa-IK-INaK+self.leak)/self.cm
58        #Variable de recuperacion
59        def funcDw(self,t,v,w,c):
60            aux=self.KD.getValues()
61            wdif=F(v=v,v0=aux['vw'],n=aux['nw'],vt=self.vt)-w
62            return w*wdif*C(v=v,vr=aux['vw'],g=aux['nw'],
63            b=aux['bw'],vt=self.vt,r=aux['rw'])
```

```
64     #Cambio de la concentracion del Calcio(En este caso no hay
           funcion)
65     def funcDc(self,t,v,w,c):
66         return 0.0
67     #Funcion para realizar integracion usando odeint de scipy
68     def rhs(self,u,t):
69         dv= self.funcDv(t=t,v=u[0],w=u[1],c=u[2])
70         dw= self.funcDw(t=t,v=u[0],w=u[1],c=u[2])
71         dc= self.funcDc(t=t,v=u[0],w=u[1],c=u[2])
72         self.v=dv
73         self.w=dw
74         self.c=dc
75         return sc.array([dv,dw,dc,t])
76     #Metodo rundemo("run" en el futuro), crea arreglos de tiempo
           y
77     #valores de v w y c respectivamente, con valores de entrada
78     #t0= tiempo inicial, tf= tiempo final, h=valor del paso
79     #ejecuta Runge-Kutta para las 3 funciones dv,dw,dc
80     #y los 4 parametros
81     #t=tiempo actual, v,w,c, al finalizar la iteracion
82     #conjunta los vectores y lo devuelve en forma de matriz de 3*
           n
83     #con n =numero de valores obtenidos
84     def rundemo(self,t0,tf,h,choption):
85         self.recalcVChannel()
86         t=np.linspace(start=t0,num=tf/h,stop=tf)
87         u=[self.v,self.w,self.c,t[0]]
88         aux=[u]
89         for i in t:
90             u=rk4(e=self.funcDv,f=self.funcDw,
91             g=self.funcDc,t=i,x=u[0],y=u[1],z=u[2],h=h)
92             aux.append(u)
93         aux=sc.transpose(aux)
94         dvvec=aux[0,:]
95         dwvec=aux[1,:]
96         dcvec=aux[2,:]
97         tvec=aux[3,:]
98         dic_csv={'V':dvvec,'W':dwvec,'T':tvec}
99         self.csvsave2(dic_csv)
100        self.qtgraph(t=tvec,v=dvvec,w=dwvec,c=dcvec,opt=choption)
```

Código 2.6 (Cont.): Ejemplo célula proyecto.

Está la sección para graficar los resultados (ver Cuadro 2.5 y Código 2.7 de las lineas 101 a la 209), un método de entrada (*qtgraph*) con el que se puede concentrar todas las opciones disponibles , de modo que a partir de los parámetros seleccionados se representa de la mejor forma las ventanas solicitadas.

Código 2.7: Ejemplo célula proyecto.

```
101 #Metodo para graficar el comportamiento de la celula
102     def qtgraph(self, t=[1], v=[1], w=[1], c=[1], opt=0):
103         #version para integracion con interfaz
104         self.window=QtmatplotTab()
105         ax=self.window.figt1.add_subplot(111)
106         ax.clear()
107         ax.plot(t,v,color="blue",label="V")
108         ax.plot(t,w,color="red",label="W")
109         ax.set_xlabel('tiempo␣(ms)') #nombre del eje x
110         ax.set_ylabel('voltaje␣(mV)') #nombre del eje y
111         ax.legend(loc='upper␣right')
112         self.window.canvast1.draw()
113         self.window.addTab(self.window.tab1,"Total")
114         if opt==1:
115             self.qtgraphchann(t=t,v=v,w=w,c=c)
116             self.qtgraphderivative(t=t,v=v,w=w,c=c)
117         self.window.show()
118     #Metodo para graficar el comportamiento de
119     #los canales
120     def qtgraphchann(self, t=[1], v=[1], w=[1], c=[1]):
121         ax=self.window.figt2.add_subplot(231)
122         ax.clear()
123         ax.plot(t,v)
124         ax.title.set_text("V␣(mV)")
125         ax.set_xlabel('tiempo␣(ms)') #nombre del eje x
126         ax.set_ylabel('voltaje␣(mV)') #nombre del eje y
127         ax2=self.window.figt2.add_subplot(232)
128         ax2.clear()
129         ax2.plot(t,w)
130         ax2.title.set_text("W")
131         ax2.set_xlabel('tiempo␣(ms)') #nombre del eje x
132         #ax2.set_ylabel('voltaje (mV)') #nombre del eje y
133         ikdvec=[]
134         inavec=[]
135         inakvec=[]
136         x=0
137         while x<len(t):
138             ina=self.Na.I(vn=v[x],wn=w[x],cn=c[x],vt=self.vt,
139             vna=self.vna)
140             ikd=self.KD.I(vn=v[x],wn=w[x],cn=c[x],vt=self.vt,
141             vk=self.vk)
142             inak=self.NaK.I(vn=v[x],wn=w[x],cn=c[x],
143             vt=self.vt,vnak=self.vnak)
144             ikdvec.append(ikd)
145             inavec.append(ina)
```

```
146                 inakvec.append(inak)
147                 x=x+1
148             ax=self.window.figt2.add_subplot(234)
149             ax.clear()
150             ax.plot(t,inavec)
151             ax.title.set_text("ina")
152             ax.set_xlabel('tiempo (ms)') #nombre del eje x
153             ax.set_ylabel('Corriente (pA)') #nombre del eje y
154             ax2=self.window.figt2.add_subplot(235)
155             ax2.clear()
156             ax2.plot(t,ikdvec)
157             ax2.title.set_text("ikd")
158             ax2.set_xlabel('tiempo (ms)') #nombre del eje x
159             ax2.set_ylabel('Corriente (pA)') #nombre del eje y
160             ax3=self.window.figt2.add_subplot(236)
161             ax3.clear()
162             ax3.plot(t,inakvec)
163             ax3.title.set_text("inak")
164             ax3.set_xlabel('tiempo (ms)') #nombre del eje x
165             ax3.set_ylabel('Corriente (pA)') #nombre del eje y
166             self.window.figt2.tight_layout()
167             self.window.canvast2.draw()
168             self.window.addTab(self.window.tab2,"Canales")
169         #Metodo para graficar el comportamiento de la derivada de
170         #los canales
171         def qtgraphderivative(self,t=[1],v=[1],w=[1],c=[1]):
172             vdervec=[]
173             wdervec=[]
174             cdervec=[]
175             x=0
176             while x<len(t):
177                 xvder=self.funcDv(t=t[x],v=v[x],w=w[x],c=c[x])
178                 xwder=self.funcDw(t=t[x],v=v[x],w=w[x],c=c[x])
179                 xcder=self.funcDc(t=t[x],v=v[x],w=w[x],c=c[x])
180                 vdervec.append(xvder)
181                 wdervec.append(xwder)
182                 cdervec.append(xcder)
183                 x=x+1
184             ax=self.window.figt3.add_subplot(121)
185             ax.clear()
186             ax.plot(v,vdervec)
187             ax.title.set_text("v")
188             ax.set_xlabel('Voltaje (mV)') #nombre del eje x
189             ax.set_ylabel('dV/dt (mV/ms)') #nombre del eje y
190             ax2=self.window.figt3.add_subplot(122)
191             ax2.clear()
192             ax2.plot(w,wdervec)
```

Código 2.7 (Cont.): Ejemplo célula proyecto.

```
193             ax2.title.set_text("w")
194             ax2.set_xlabel('W') #nombre del eje x
195             ax2.set_ylabel('dW/dt (1/ms)') #nombre del eje y
196             self.window.figt3.tight_layout()
197             self.window.canvast3.draw()
198             self.window.addTab(self.window.tab3,"Derivada")
199 
200         #Metodo para graficar winf de la funcion C
201         #no utilizado en este caso
202         def qtgraphcfunc(self,v0,vf,h,vr,g,b,vt,r):
203             xvect=np.linspace(start=v0,num=vf/h,stop=vf)
204             yvect=[]
205             for i in xvect:
206                 yvect.append(C(v=i,vr=vr,g=g,b=b,vt=vt,r=r))
207             self.windowc=Qtmatplot()
208             ax=self.windowc.figure.add_subplot(111)
209             ax.clear()
210             ax.plot(xvect,yvect)
211             ax.title.set_text("Funcion C")
212             ax.set_ylabel('C') #nombre del eje x
213             ax.set_xlabel('voltaje (mV)') #nombre del eje y
214             self.windowc.canvas.draw()
215             self.windowc.show()
```

Código 2.7 (Cont.): Ejemplo célula proyecto.

Por último esta la sección para guardar los resultados (ver Cuadro 2.5 y Código 2.8 de las lineas 216 a la 227), con el uso del método (*csvsave2*) estos pasan a un archivo de hoja de calculo gracias al uso de la biblioteca *pandas* para comodidad de los usuarios.

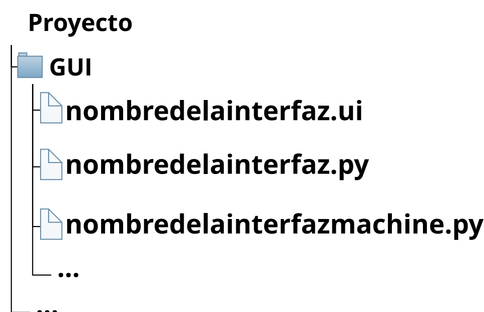
Código 2.8: Ejemplo célula proyecto.

```
216     #Metodo para salvar en Hojas de calculo
217         def csvsave2(self, my_dict):
218             array=[]
219             for i in my_dict.keys():
220                 if len(array)==0:
221                     array=my_dict[i]
222                 else:
223                     array=np.vstack([array,my_dict[i]])
224             array=sc.transpose(array)
225             df=pd.DataFrame(array)
226             df.columns=my_dict.keys()
227             df.to_csv('save/hh.csv',index=False)
```

Con el código descrito se pueden modelar diferentes tipos celulares. Sin embargo, este proceso necesita que el usuario entienda código escrito en Python y familiarizarse con la estructura del proyecto, porque al agregar un canal se requiere agregar los archivos necesarios en todas las zonas del proyecto.

Generalmente puede llegar a ser complicado para los alumnos que no tienen bases de programación (Davison et al. (2009)), por ello se tomó la decisión de diseñar una interfaz gráfica de usuario que permita interactuar con el modelo sin necesidad de estar en contacto directo con el código .

2.7. Estructura lógica de los archivos de la interfaz gráfica



Cuadro 2.6: Estructura propuesta de los archivos de interfaz gráfica

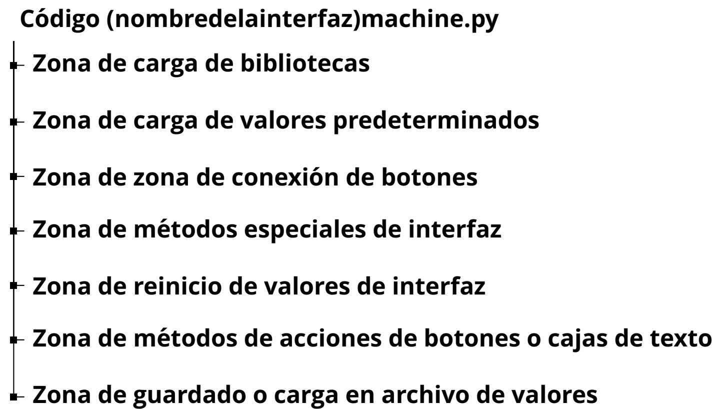
La interfaz gráfica (Cuadro 2.6) tiene como base la generación de pantallas “*vistas*” con las que interactúa el usuario. Dichas pantallas necesitan de tres tipos de archivo para ejecutarse. Primero ”*nombredelainterfaz.ui*” que es el archivo generado al utilizar el programa ”*Qtdesigner*”, Qt es un conjunto de rutinas agrupadas en bibliotecas (framework) que permite el desarrollo de interfaces de usuario. QtDesigner permite colocar elementos como botones, cajas de texto, etiquetas y pestañas de manera gráfica en una ventana, sin la necesidad de tener que escribir en código la colocación de todos estos elementos, ahorrando tiempo de programación (Qt (2019)).

Estas descripciones se guardan es en un archivo con extensión *.ui. Para que la interfaz se pueda ejecutar en Python, necesitamos convertir el archivo con extensión *.ui generado, a extensión *.py, utilizando la biblioteca ”pyuic” que genera el archivo ”*nombredelainterfaz.py*” esto lo hace de manera automática cada vez que se modifica la interfaz.

Cada modificación genera un nuevo archivo que sustituye al anterior, por lo que hacer modificaciones a éste no es recomendable. Por ello, los métodos para el uso de las pantallas se colocan en el tercer tipo de archivo de nombre ”*nombredelainterfazmachine.py*”. Este último contiene los protocolos de interacción entre la interfaz y el modelo.

2.8. Estructura del código de la interfaz gráfica

La escritura de los archivos "*nombredelainterfazmachine.py*" es la siguiente (Cuadro 2.7):



Cuadro 2.7: Estructura del código donde se alojan las acciones de GUI diseñada

En la zona de bibliotecas (Cuadro 2.7 y Código 2.9 linea 15), se importa el archivo creado con la interfaz desarrollada en Qt para después importar el archivo de la célula.

Las zonas de carga de valores predeterminados de las diferentes variables de la célula (Cuadro 2.7 y Código 2.9 lineas 20 a 22),es en donde se relacionan las cajas de texto creadas en la interfaz grafica y valores que se extraen de la biblioteca de la célula (ver apendice falta) y estos se asignan en la sección determinada. Un ejemplo de la línea para establecer el texto en las cajas de texto con los valores es:

```
1 self.nombre–textBox.setText(str(biblioteca.biophys['valor']['
      max/min']))
```

Cuadro 2.8: Ejemplo insertar texto en *textbox*

En el Cuadro 2.8 en la linea 1 y Código 2.9 linea 37:

- la palabra *self* corresponde al objeto en el cual se está trabajando
- *nombre-textBox* es para reconocer la etiqueta dada al *textBox* en el que se establecen los datos y este objeto tiene un método llamado *setText* que solicita un objeto del tipo *string*
- *str* es un método que convierte valores numéricos en *strings* (cadenas)

- finalmente la secuencia de caracteres a convertir, en este caso, se encuentra en *biblioteca.biophys* donde por convención le colocaremos dos etiquetas *valor* y *max* o *min* denotando si es el mínimo o máximo del valor que la variable puede alcanzar en la célula

La siguiente zona conecta los botones en la interfaz con los métodos que ejecutan operaciones dentro del objeto célula creados en la carpeta *neuron* del proyecto (Cuadro 2.7). Para poder realizar este enlace, se utiliza la implementación de la biblioteca Qt con nombre QtCore, la siguiente línea de código es:

```
1 QtCore.QObject.connect(self.boton_conectado,QtCore.SIGNAL(
      _fromUtf8("clicked()")), lambda:self.metodo_conectado())
```

Cuadro 2.9: Ejemplo conectar botones de interfaz

En el Cuadro 2.9 en la linea 1 y y Código 2.9 lineas 45 a 67 hay que notar que dos métodos son utilizados de la biblioteca QtCore:

- en primera *connect*, que encapsula los objetos y métodos a relacionar; además de *SIGNAL* como el segundo método que se encuentra a la espera de una acción.
- en este caso *clicked*, una vez detectada la señal ejecuta el método marcado con la etiqueta lambda asignando las entradas en caso de ser necesario.

La zona de métodos especiales de la interfaz (Cuadro 2.7 y Código 2.9 lineas 69 a 102)) cuenta con los métodos utilizados para la operación interna del GUI. Por ejemplo:

- revisa si los valores ingresados en las cajas de texto son numéricos.
- si exceden un valor máximo o mínimo.
- cambia el color de la caja dependiendo del caso.

La zona de reinicio de valores de interfaz (Cuadro 2.7 y Código 2.9 lineas 103 a 115):

- se encarga de los valores predeterminados
- maneja la forma de colocarlos dentro de la interfaz

así el usuario no necesita cerrar y abrir el programa para regresar a un punto donde la interfaz trabaje de forma exitosa y segura.

En la zona de métodos de acciones de botones o cajas de texto (Cuadro 2.7 y Código 2.9 lineas 117 a 181) es en donde se encuentran los métodos que conectamos en la zona de conexión de botones y estos son llamados cada vez que se aprieta un botón en la interfaz como ya se había mencionado anteriormente.

La última zona, que es de guardado o carga en archivo de valores (Cuadro 2.7 y Código 2.9 lineas 182 a 207) utiliza la biblioteca *Qt* para crear un dialogo para guardar o cargar los valores en toda la interface para su uso futuro, la extensión del archivo en el que se guarda o carga es con extensión *npy*.

El Código 2.9 es un ejemplo de esta organización de código, es prudente advertir que a este se le han eliminado lineas repetitivas dejando sólo algunas representativas para fines de la explicación en este capítulo.

Código 2.9: Ejemplo archivo conexión interfaz.

```
1  """
2  hhguimachine2.py
3  Martin Garcia Mata
4  """
5  #! /usr/bin/python
6  #La siguientes lineas son usadas para compatibilidad en
7  #el sistema de archivos
8  import sys
9  import os.path
10 sys.path.append(os.path.join(os.path.dirname(__file__), '..'))
11 #Biblioteca interfaz grafica
12 from PyQt4 import QtGui,QtCore
13 import numpy as np
14 '''Interfaces para gui'''
15 from hhgui2 import Ui_HHForm
16 from errorwindow import errorwindow
17 #from qtmatplot import qtmatplot
18 from Neuron.HH2 import HH2
19 '''Valores predeterminados para las celulas'''
20 import Test.HHTest as thht
21 import Test.MxMn as mxmn
22 import Test.valdef as vdf
23 #Biblioteca de funciones genericas contenida en /Func/auxfunc.py
24 from Func.auxfunc import *
25 ...
26 '''Cada clase es una ventana de GUI en cada clase existen
27 las conexiones y metodos necesarios para su correcto uso'''
28 class HHForm(QtGui.QDialog, Ui_HHForm):
29     def __init__(self, parent=None):
30         QtGui.QDialog.__init__(self, parent)
31         self.setupUi(self)
32         self.aa2=HH2(thht.biophyshh,thht.biophysna,
33         thht.biophysnak,thht.biophyskd)
34         '''Zona carga valores maximos y minimos default LineEdit
               '''
35
```

```
36          #General tab
37          self.hhgral_v_maxLabel.setText(
38          str(mxmn.biophyshhmxmn['v']['mx']))
39          self.hhgral_v_minLabel.setText(
40          str(mxmn.biophyshhmxmn['v']['mn']))
41          ...
42          self.initialtime_LineEdit.setText(str(thht.biosimula['it'
                ]))
43          ...
44          #Reset valores del objeto
45          self.resetall(biophyshh=thht.biophyshh,biophysna=thht.
                biophysna
46          ,biophyskd=thht.biophyskd,biophysnak=thht.biophysnak)
47          '''Zona conexiones sliders y botones'''
48          #step slider
49          QtCore.QObject.connect(self.winf_stepsize_Slider,
50          QtCore.SIGNAL(_fromUtf8("valueChanged(int)")),
51          lambda: self.myFloatSetNum(self.winf_stepsize_Label,
52          self.winf_stepsize_Slider))
53          ...
54          #buttons
55          QtCore.QObject.connect(self.general_reset_PushButton,
56          QtCore.SIGNAL(_fromUtf8("clicked()")),
57          lambda: self.resetgral(biophyshh=thht.biophyshh))
58          ...
59          QtCore.QObject.connect(self.load_PushButton,
60          QtCore.SIGNAL(_fromUtf8("clicked()")),
61          lambda: self.loadandsaveDialog('load'))
62          ...
63          #value LineEdit
64          QtCore.QObject.connect(self.hhgral_v_LineEdit,
65          QtCore.SIGNAL(_fromUtf8("editingFinished()")),
66          lambda: self.updateValue(self.hhgral_v_LineEdit,
67          self.hhgral_v_maxLabel,self.hhgral_v_minLabel))
68          ...
69          #Conectar Tooltip
70          self.hhgral_v_LineEdit.setToolTip(str(self.getValDef('v')
                ))
71          ...
72      #Metodo para slider con numeros float
73      def myFloatSetNum(self, worklabel, slider):
74          mynum=0.0
75          mynum=slider.value()/10.0
76          worklabel.setNum(mynum)
77      #metodo obtener definicion de valor de diccionario
78      def getValDef(self, label):
79          if label in vdf.valuedef:
```

Código 2.9 (Cont.): Ejemplo archivo conexión interfaz.

```
80                return vdf.valuedef[label]
81            else:
82                return "Sin␣definir"
83    #comprobador valores en LineEdit
84    def updateValue(self, linedit, max, min):
85        #linedit.setFocus()
86        try:
87            if float(linedit.text()) >= float(min.text()) and
88            float(linedit.text()) <= float(max.text()):
89                linedit.setStyleSheet
90                ("background-color:rgb(144,238,144)")#green
91            else :
92                #linedit.setFocus()
93                linedit.setStyleSheet
94                ("background-color:rgb(255,99,71)")#red
95                window2=errorwindow("Atencion!:Error␣en␣valores")
96                linedit.setFocus()
97        #break
98        except ValueError:
99            linedit.setStyleSheet
100           ("background-color:rgb(255,99,71)")#red
101           errorexecute=True
102           print errorexecute
103   #zona reset de tabs
104   def resetgral(self,biophyshh):
105       self.hhgral_v_LineEdit.setText(str(biophyshh['v']))
106       ...
107       self.hhgral_v_LineEdit.setStyleSheet
108       ("background-color:rgb(245,245,245)")#gray
109       ...
110   def resetall(self,biophyshh,biophysna,biophyskd,biophysnak):
111       self.resetgral(biophyshh)
112       self.resetna(biophysna)
113       self.resetkd(biophyskd)
114       self.resetnak(biophysnak)
115       errorexecute=False
116       ...
117   #Metodo de graficacion recogiendo los valores de loa interfaz
118   def generalGraph(self):
119       #zona para if de ventana de error
120       #self.hide()
121       self.simulation_Frame.setEnabled(False)
122       self.simulation_CheckBox.setEnabled(False)
123       QtGui.QApplication.processEvents()
124       #Actual values general
125       self.aa2.v=float(self.hhgral_v_LineEdit.text())
126       self.aa2.w=float(self.hhgral_w_LineEdit.text())
```

Código 2.9 (Cont.): Ejemplo archivo conexión interfaz.

```
127             ...
128             if self.channelinclude_CheckBox.isChecked():
129                 self.aa2.rundemo(
130                 t0=float(self.initialtime_LineEdit.text()),
131                 tf=float(self.finaltime_LineEdit.text()),
132                 h=float(self.stepsize_LineEdit.text()),choption=1)
133             else:
134                 self.aa2.rundemo(t0=float(self.initialtime_LineEdit.
                        text()),
135                 tf=float(self.finaltime_LineEdit.text())
136                 ,h=float(self.stepsize_LineEdit.text()),choption=0)
137             self.simulation_CheckBox.setEnabled(True)
138             self.simulation_CheckBox.setChecked(False)
139             self.channelinclude_CheckBox.setChecked(False)
140
141         #metodo graficar winf
142         def graphwinf(self):
143             auxv0=float(self.winf_inflimit_Label.text())
144             ...
145             self.aa2.qtgraphcfunc(v0=auxv0,vf=auxvf,h=auxstep,
146             vr=auxvw,g=auxnw,b=auxbw,vt=auxvt,r=auxrw)
147             self.winf_Frame.setEnabled(False)
148             self.winf_CheckBox.setChecked(False)
149         #zona graficacion de canales
150         def graphna(self,opt):
151             if opt==0:#funcion f
152                 auxv0=float(self.naffun_inflim_Label.text())
153                 auxvf=float(self.naffun_suplim_Label.text())
154                 auxstep=float(self.naffun_stepsize_Label.text())
155                 self.aa2.Na.halfact=float(self.hhna_halfact_LineEdit.
                        text())
156                 self.aa2.Na.gain=float(self.hhna_gain_LineEdit.text()
                        )
157             if opt==1:#funcion s
158                 auxv0=float(self.nasfun_inflim_Label.text())
159                 auxvf=float(self.nasfun_suplim_Label.text())
160                 auxstep=float(self.nasfun_stepsize_Label.text())
161                 self.aa2.Na.s=float(self.hhna_s_LineEdit.text())
162                 self.aa2.Na.a=float(self.hhna_a_LineEdit.text())
163                 self.aa2.Na.nu=float(self.hhna_nu_LineEdit.text())
164             auxvt=float(self.hhgral_vt_LineEdit.text())
165             auxinNa=float(self.hhgral_inna_LineEdit.text())
166             auxoutNa=float(self.hhgral_outna_LineEdit.text())
167             auxvna=vNernst(vT=auxvt, cIn=auxinNa, cOut=auxoutNa, val
                    =1.0)
168             self.aa2.Na.qtgraph(v0=auxv0,vf=auxvf,step=auxstep,vt=
                    auxvt,
```

Código 2.9 (Cont.): Ejemplo archivo conexión interfaz.

```
169           vna=auxvna , opt=opt )
170           self . nasfun_Frame . setEnabled ( False )
171           self . nasfun_CheckBox . setChecked ( False )
172           self . naffun_Frame . setEnabled ( False )
173           self . naffun_CheckBox . setChecked ( False )
174           ...
175       def graphsfFunction ( self , wich , opt ) :
176           return {
177           'na ': self . graphna ( opt ) ,
178           'kd ': self . graphkd ( opt ) ,
179           'nak ': self . graphnak ( opt )
180           } . get ( wich , lambda : 'nothing ')
181       #zona de carga y guardado de valores
182       def loadandsaveDialog ( self , opt ) :
183           if opt== 'save ':
184               fileName = QtGui . QFileDialog . getSaveFileName ( self ,
185               'Guardar␣Archivo ' , '/save/hhpersonal . npy ' ,
186               selectedFilter= '∗. npy ')
187               if fileName :
188                   dictsave={ 'biophyshh ':{ 'v ': self . hhgral_v_LineEdit
                          . text ( ) ,
189                                                  'w ': self . hhgral_w_LineEdit
                                                          . text ( ) ,
190                   ...
191                   np . save ( str ( fileName ) , dictsave )
192                   #print fileName
193           if opt== 'load ':
194               try :
195                   fileName = QtGui . QFileDialog . getOpenFileName ( self
                          ,
196                   'Cargar␣Biblioteca ' , '/save/∗. npy ' ,
197                   selectedFilter= '∗. npy ')
198                   if fileName :
199                       readdict = np . load ( str ( fileName ) ,
200                       allow_pickle=True ) . item ( )
201                       self . resetall ( biophyshh=readdict [ 'biophyshh '
                              ] ,
202                       biophysna=readdict [ 'biophysna '] ,
203                       biophyskd=readdict [ 'biophyskd '] ,
204                       biophysnak=readdict [ 'biophysnak '] )
205                       #print fileName
206               except KeyError :
207                   window2=errorwindow ( "Error␣en␣carga␣de␣valores " )
```

Código 2.9 (Cont.): Ejemplo archivo conexión interfaz.

A partir de los requerimientos y de establecer los resultados mínimos esperados de entrega, es que se ha descrito la forma de estructurar los archivos y de también, escribir los archivos de este trabajo. Este proyecto pretende que la programación sea sencilla, aprovecha un conjunto de bibliotecas de interfaz que amplía la funcionalidad del proyecto y deja la posibilidad de agregar otras aplicaciones con la misma facilidad sin modificar una gran cantidad de código. Por ejemplo se podría agregar la posibilidad de simular protocolos experimentales como curvas intensidad frecuencia.

Capitulo 3: Resultados del proyecto

En este capítulo se describe una ejecución del programa resultante. Se exponen las funcionalidades de las pantallas para su uso y la generación de las gráficas de varios tipos celulares, así como una breve comparación de los resultados obtenidos en el sistema y los resultado de experimentos reales.

3.1. Resultados de la ejecución del proyecto

Pantalla de bienvenida:

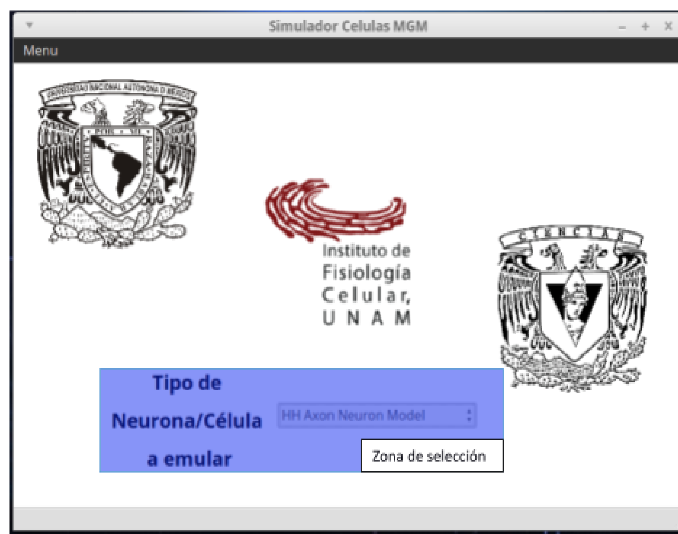


Figura 3.6: Pantalla de bienvenida.

Al iniciar la interfaz gráfica aparece la pantalla de bienvenida (ver Figura 3.6), contiene el logotipo de la Universidad, el de la Facultad de Ciencias y el del Instituto de Fisiología Celular, seguido de una zona de selección donde se despliega un menú donde es posible escoger las células disponibles para la simulación. Al seleccionar una de las opciones una nueva ventana aparece encima de ésta. La pantalla de la célula se muestra a continuación.

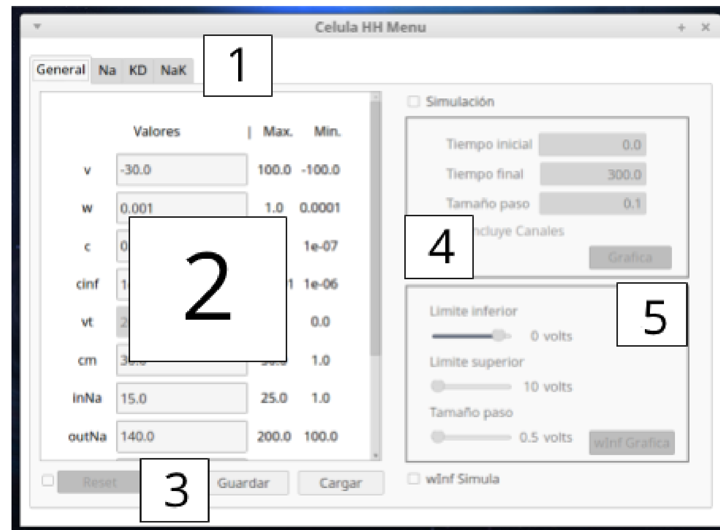


Figura 3.7: Pantalla de la célula.

En la pantalla de la célula (ver la Figura 3.7) tenemos cinco zonas designadas, un selector de pestañas (zona 1) que permite elegir entre un panel general o el panel de cada uno de los canales en la célula. En la pestaña “*General*” encontramos un listado de los valores propios de la célula (zona 2). Para cada valor tenemos nombre de la variable, un cuadro de texto editable con el valor predeterminado insertado, el valor máximo y mínimo de la variable, entre los cuales se puede asignar un nuevo valor.

Los máximos y mínimos corresponden a los límites de los sistemas biológicos. En la zona 3, existen tres botones:

1. El botón “*Reset*“ con el cual se restablecen los valores predeterminados, se debe activar la casilla a la izquierda del botón para su uso, esto evita que el usuario borre los datos por accidente.

2. El botón “*Guardar*” despliega una ventana que permite escoger el lugar donde guardar el archivo, recolecta los valores de todas las variables en ese momento y los almacena en un archivo con extensión *txt*.

3. El botón “*Cargar*” despliega una ventana donde permite escoger el archivo de extensión *txt* válido para establecer parámetros usado anteriormente en alguna simulación.

La zona 4 (ver la Figura 3.7) es el área destinada para la ejecución de la célula/-neurona, presenta tres cajas de texto editables para establecer valores de tiempo de inicio, final y tamaño del paso de tiempo. Contiene el botón *”Gráfica”* el cual inicia la simulación por el tiempo seleccionado, se incluye una casilla que permite graficar los valores de corriente de los diferentes canales de la célula.

Del lado derecho de la pantalla en la zona 5 (ver la Figura 3.7) se encuentra el cálculo de la función del estado estable de la variable de recuperación w o winf en función del voltaje. Para seleccionar el rango de voltaje y el tamaño de paso para la simulación existen controles deslizantes.

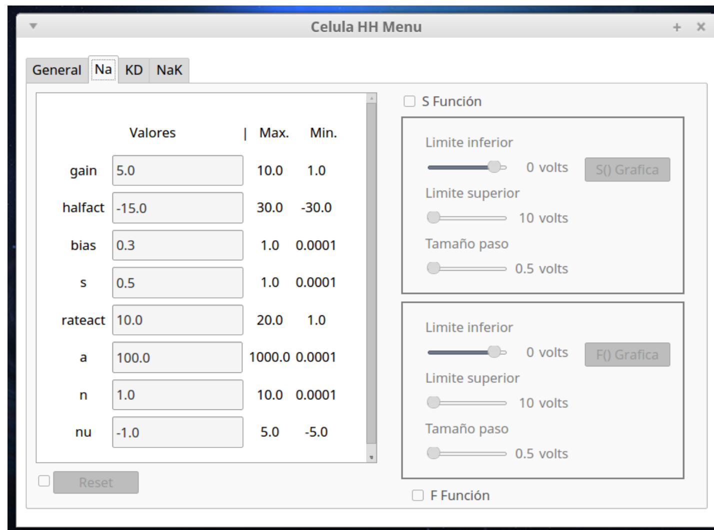


Figura 3.8: Pantalla de canal.

La pantalla de la Figura 3.8 se encuentran de lado izquierdo con los valores de las diferentes variables a su vez relacionadas con los canales, de la misma forma que en la célula tenemos el nombre de la variable, cuadro de texto para insertar un valor y los valores mínimos y máximos. Se puede observar que no tiene botones de carga y guardado de valores ya que los valores de cada canal se guardan junto con los valores de la célula. En la parte superior derecha, se puede graficar el flujo en función del voltaje para cada canal S (ecuación **??**). Y en la parte inferior la fracción de canales abiertos en función del voltaje F (ecuación **??**). Para cada una hay controles deslizables que permiten variar los valores de límite inferior al límite superior y el rango de voltaje.

En caso de que algún valor de las diferentes pestañas se encuentre fuera de los intervalos establecidos, el sistema despliega una ventana de advertencia y el texto en dicha casilla se torna rojo, permitiendo al usuario realizar las correcciones para continuar con la ejecución. El resultado de la ejecución de la simulación se expone a continuación.

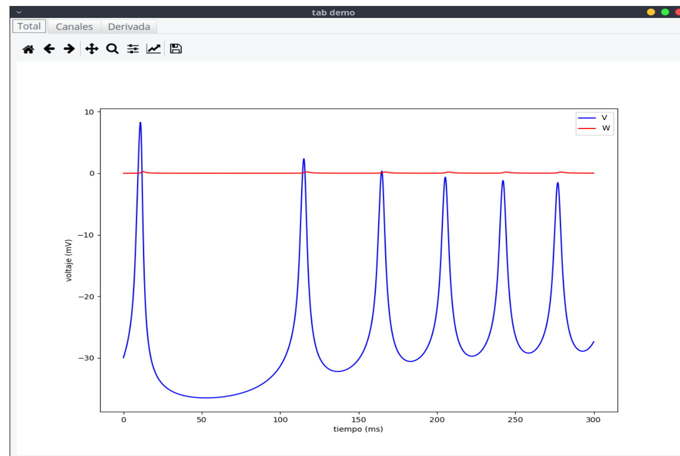


Figura 3.9: Pantalla de calculo Total.

La Figura 3.9 contiene la gráfica del resultado de la ejecución de la célula/neurona con los parámetros establecidos. Muestra los valores de voltaje obtenidos en función del tiempo. Si se marca la casilla de “*Incluye canales*” entonces también se despliega la Figura 3.10

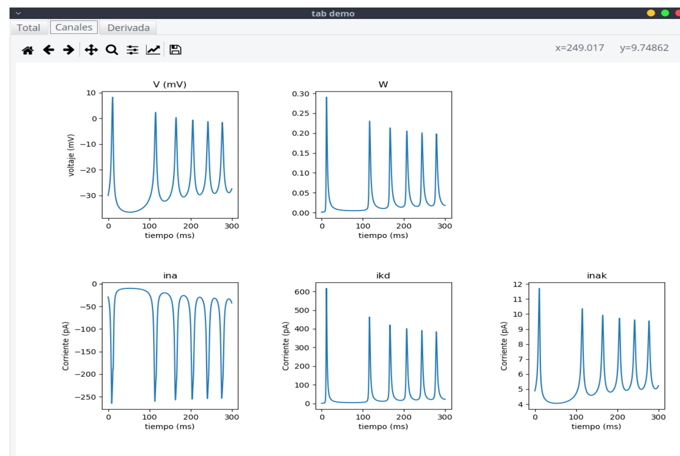


Figura 3.10: Pantalla de calculo de canales.

En la parte superior de la Figura 3.10 está la gráfica de valores de voltaje (v), la variable de recuperación (w) y la concentración intracelular de Calcio (c) en función del tiempo. Y en la parte inferior tenemos el valor de corriente de cada uno de los canales de la célula también en función del tiempo.

Para poder analizar mejor los resultados, como se muestra en la Figura 3.11, desplegamos el calculo de la derivada resultante para los valores V, W y C (en caso de ser utilizado)

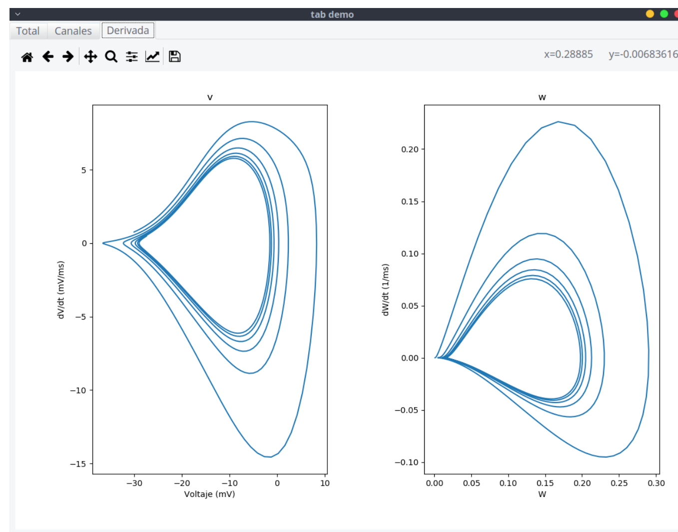


Figura 3.11: Pantalla derivada.

3.2. Simulación de células excitables

En esta sección se presentan las simulaciones generadas por el programa en los tres distintos tipos de células excitables modeladas, una comparación con registro obtenidos experimentalmente al registrar los cambios de voltaje de diferentes células.

3.2.1. Célula de nodo sinoauricular

Un tipo celular excitable que se ha modelado previamente es la célula de nodo sinoauricular (Verkerk et al. (2013)). El modelo implementado se ha utilizado para simular el comportamiento particular de este tipo celular (Herrera-Valdez (2018b)). Se duplicó esta simulación para validar esta implementación. La gráfica de la Figura 3.12 muestra el voltaje en función del tiempo del modelado de una célula de nodo sinoauricular. En el que se nota que hay 3 potenciales de acción, que son la subida en el voltaje.

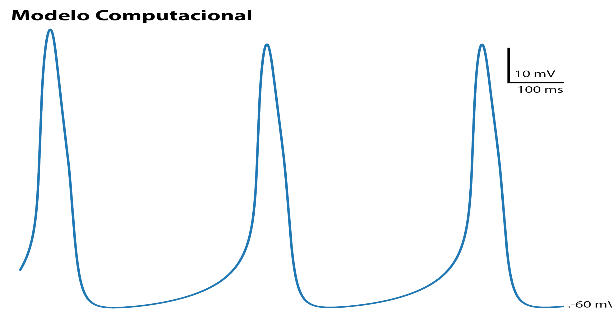


Figura 3.12: Registro Cardiaca.

Esta simulación se incluye debido a concuerda con lo reportado anteriormente para el modelo en Herrera-Valdez (2018b) validando la implementación. Además, se puede ver que es muy similar a los registros reales reportados en literatura previa (Verkerk et al. (2013)). Esta simulación se realizó sin utilizar la interfaz gráfica lo que muestra que la interfaz gráfica y la implementación pueden funcionar de manera independiente.

3.2.2. Interneurona estriatal de disparo rápido

Para mostrar el uso del modelo se modeló una interneurona de disparo rápido o Fast Spiking estriatal, lo cual se observa en la Figura 3.13. Primero se muestra el registro obtenido de manera experimental del voltaje en función del tiempo de una en respuesta a un pulso de corriente despolarizante de una neurona de este tipo (A). Se caracteriza por presentar un disparo a alta frecuencia (Assous and Tepper (2019)). Para reproducir este comportamiento se usa un modelo simple con tres objetos canal: Sodio, Potasio y bomba de Sodio-Potasio. También se encuentra el voltaje en función del tiempo obtenido de la simulación (B)

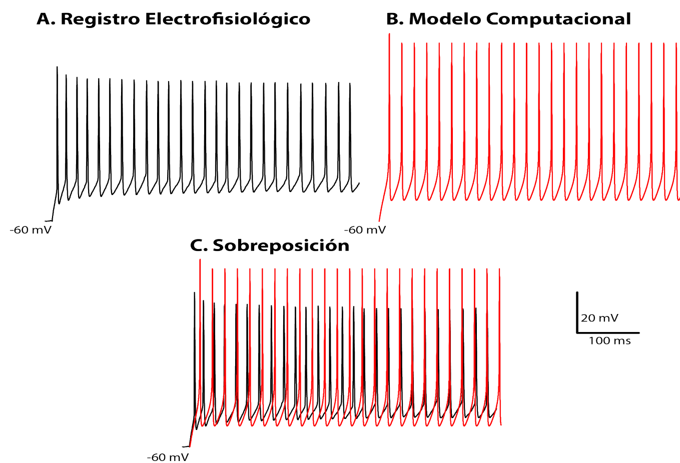


Figura 3.13: Fast Spiking voltaje-tiempo.

Se observa en la Figura 3.13 que la frecuencia de disparo es similar a la observada de manera experimental (C); sin embargo, el comportamiento no es idéntico, esto se debe a la sencillez del modelo. Para obtener un disparo más apegado al registro es necesario incluir más variables como canales, que se pueden integrar a la implementación como se encuentra, así como entradas sinápticas que permitan añadirse en futuras implementaciones.

3.2.3. Interneurona colinérgica

La interneuronas colinérgicas presentan un disparo tónico a baja frecuencia, se relacionan con procesos de atención y recompensa (Deng et al. (2007)) así como regulación en la enfermedad de Parkinson (Ztaou and Amalric (2019)).

La Figura 3.14 Muestra el registro experimental del voltaje en función del tiempo obtenido de una de ellas (A)

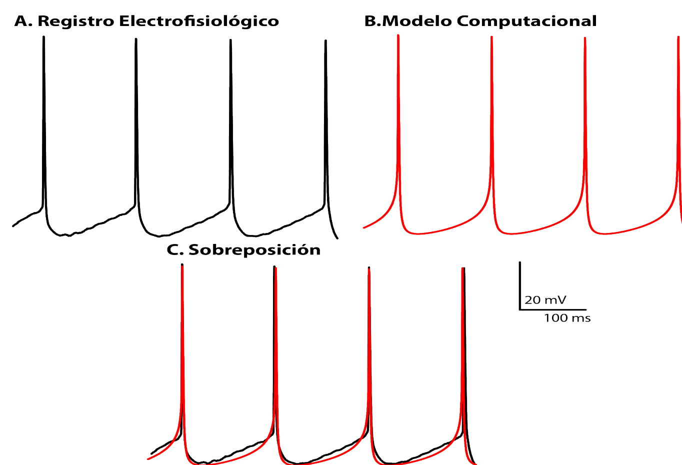


Figura 3.14: Interneurona Colinergica.

En la Figura 3.14 en la parte B se muestra la simulación generada por el modelo, se puede observar el conjunto de disparos de dicha neurona. La simulación de este tipo neuronal incluye el cálculo del cambio de la concentración de Calcio intracelular en función del tiempo así como canales de Calcio e intercambiador de Sodio Calcio en adición a los objetos incluidos para modelar a la interneurona de disparo rápido, ya que se sabe que una corriente de Calcio es responsable de promover el disparo tónico de estas neuronas.

Se ha conseguido un producto con muy buenos resultados, en primer lugar, la interfaz va a permitir un uso del programa adecuado, y en segundo lugar, las gráficas permiten simular un comportamiento similar a los experimentos reales sin dejar de olvidar el error que lleva el calculo reflejado en la gráfica de cada célula modelada.

Conclusiones

Con los resultados antes descritos se ha conseguido implementar el modelo termodinámico (Herrera-Valdez (2018a), Herrera-Valdez (2015)) para modelar el potencial transmembranal de varios tipos de células excitables utilizando el lenguaje Python.

- El entorno para aprendizaje y experimentación con el modelo termodinámico de potencial transmembranal de deriva-difusión se ha generado con un modelo desarrollado en paradigma orientado a objetos y se integró a una interfaz gráfica.
- Con las ventanas que se mostraron en el la zona de resultados, se muestra la integración de la implementación a una interfaz gráfica de usuario programada en pyQT, lo que permite la manipulación amigable del modelo para personas no familiarizadas con algún tipo de escritura de código.
- Se implementó el proyecto utilizando el Modelo Vista Controlador de manera que tanto el entorno cómo el modelo matemático permanecen por un lado como estructuras independientes y por el otro con la cercanía necesaria para realizar una aplicación compleja con un intercambio correcto de mensajes.
- Nos permitió modelar de forma eficiente diferentes tipos celulares basados en datos experimentales, comprobando que el código es consistente porque mantiene las características propuestas del modelo.

El programa preserva la estructura suficiente para eventualmente simular redes de células formadas por diferentes poblaciones celulares, esto hará que las redes celulares sean más apegadas a la realidad que las existentes, la mayoría de las redes celulares que se encuentran en la literatura no contemplan diferentes poblaciones celulares que forman un tejido.

Apéndice

4.1. Biblioteca de valores ejemplo

El siguiente Código (4.10) es un ejemplo de los valores que contiene una célula modelada, estos valores se encuentran contenidos en la estructura de Python llamada diccionario y cada uno corresponde a los valores de un canal que compone la célula (listin 4.10 linea 36 a 68) ó parámetros propios de la célula (listin 4.10 linea 19 a 35). Cada valor corresponde a un parámetro biofísico, por lo que esa es la razón de cada diccionario comience con la palabra *biophys* y agregamos después un identificador de canal o célula. Este archivo de ejemplo puede ser usado para probar el funcionamiento del modelo sin la necesidad de utilizar la interfaz gráfica, utilizar como ejemplo en el listin 4.10 linea 75 a 80.

Código 4.10: Ejemplo valores célula proyecto.

```
1  """
2  TANTest.py
3  Martin Garcia Mata
4  """
5  #La siguientes lineas son usadas para compatibilidad en el
6  #sistema de archivos
7  import sys
8  import os.path
9
10 sys.path.append(os.path.join(os.path.dirname(__file__), '..'))
11
12 #Importar neurona modelada
13 from Neuron.TAN import *
14 from PyQt4.QtGui import QApplication
15
16 #Zona de datos predeterminados
17 #Parametros generales propiedad de la celula/neurona
18 global biophystan
19 biophystan={'v':-80.0,#voltaje inicial
20            'w':0.01,#w inicial
```

```
21            'c':0.1,#concentracion de Calcio inicial
22            'leak':0.0,#corriente constante que se agrega a la
                  celula
23            'cinf':0.1,#estado estable de Calcio
24            'vt':26.5,
25            #constante de los gases*t(en Kelvin)/constante de
                  Faraday
26            'cm':30.0,#capacitancia de la membrana
27            'inNa':12.0,#10concentracion interna de sodio
28            'outNa':140.0,#150concentracion externa de sodio
29            'inK':130.0,#concentracion de potasio
30            'outK':5.0,#concentracion externa de potasio
31            'vATP':-450.0,#450potencial del ATP
32            'rc':0.05,#ratecalciumrecovery
33            'kc':20.0,#ratecalciumentry AGREGAR
34            'Faraday':96485.33289,#constante de Faraday
35            'outCa':1500#concentracion externa de Calcio
36            }
37 #Propiedades caracteristicas del canal de Sodio
38 global biophysna
39 biophysna={'gain':4.0, #6.0,
40            'halfact':-20.0,
41            's':0.5,#0.5
42            'a':1100.0,#100
43            'nu':-1.0}#-1.0
44 #Propiedades caracteristicas del canal de Calcio
45 global biophysca
46 biophysca={'gain':5.5, #5.0,
47            'halfact':-35.0,
48            's':0.5,#0.5
49            'a':2.0,#100
50            'nu':-2.0}#
51 #Propiedades caracteristicas del canal de Potasio
52 global biophyskd
53 biophyskd={'gain':2.5, #4.0,
54            'halfact':-20.0,#0.0
55            'bias':0.6,#0.6
56            's':0.9,#0.5
57            'rateact':0.8,#1.0
58            'a':1000.0,#400.0
59            'nu':1.0}#1.0
60 #Propiedades caracteristicas de intercambiador Sodio-Potasio
61 global biophysnak
62 biophysnak={'s':0.5,#0.5
63            'a':5.0, #2.5
64            'nu':1.0}#1.0
65 #Propiedades caracteristicas de intercambiador Sodio-Calcio
```

Código 4.10 (Cont.): Ejemplo valores célula proyecto.

```
66 global biophysnaca
67 biophysnaca={'s':0.5,#0.5
68            'a':2.0, #2.5
69            'nu':-1.0}#1.0
70 #Propiedades predeterminadas para simulacion
71 global biosimula
72 biosimula={'it':0.0,
73              'ft':300.0,
74              'stpsz':0.1}
75
76 if __name__ == '__main__':
77     app = QApplication(sys.argv)
78     aa2=TAN(biophystan=biophystan,biophysna=biophysna,
79     biophyskd=biophyskd,biophysnak=biophysnak,
80     biophysca=biophysca, biophysnaca=biophysnaca)
81     aa2.rundemo(0.0,500.0,0.1,1)
82     sys.exit(app.exec_())
```

Código 4.10 (Cont.): Ejemplo valores célula proyecto.

Bibliografía

Ascher, U. and Petzold, L. (1998). *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics.

Assous, M. and Tepper, J. M. (2019). Excitatory extrinsic afferents to striatal interneurons and interactions with striatal microcircuitry. *European Journal of Neuroscience*, 49(5):593–603.

Av-Ron, E., Parnas, H., and Segel, L. A. (1991). A minimal biophysical model for an excitable and oscillatory neuron. *Biological Cybernetics*, 65(6):487–500.

Becerril Espinosa, J. V. and Elizarraraz Martínez, D. (2004). Ecuaciones Diferenciales. Técnicas de Solución y Aplicaciones. pages 1–252.

Bucanek, J. (2009). *Learn Objective-C for Java Developers*. Apress, Berkeley, CA.

Clay, J. R. (2005). Axonal excitability revisited. *Progress in Biophysics and Molecular Biology*, 88(1):59–90.

Cole, K., Antosiewicz, H., and Rabinowitz, P. (1955). Automatic computation of nerve excitation. *Journal of the Society for Industrial and Applied Mathematics*, 3(3):153–172.

Davison, A. P., Hines, M. L., and Muller, E. (2009). Trends in programming languages for neuroscience simulations. *Frontiers in Neuroscience*, 3(DEC):374–380.

Deng, P., Zhang, Y., and Xu, Z. C. (2007). Involvement of ih in dopamine modulation of tonic firing in striatal cholinergic interneurons. *Journal of Neuroscience*, 27(12):3148–3156.

Fraser, J. A. and Huang, C. L.-H. (2007). Quantitative techniques for steady-state calculation and dynamic integrated modelling of membrane potential and intracellular ion concentrations. *Progress in biophysics and molecular biology*, 94(3):336–372.

Freeman, A. (2015). The Model/View/Controller Pattern. *Pro Design Patterns in Swift*, (Mvc):527–552.

Galarraga, E. and Bargas-Díaz, J. (2010). Potencial de membrana y de acción. *Fisiología humana*, 1:62–73.

Gerstner, W., Kistler, W. M., Naud, R., and Paninski, L. (2014). *Neuronal dynamics: From single neurons to networks and models of cognition*.

Gómez, M. (2011). *Material Didáctico Notas Del Curso*.

Herrera-Valdez, M. A. (2012). Membranes with the same ion channel populations but different excitabilities. *PLoS ONE*, 7(4).

Herrera-Valdez, M. A. (2014). *Geometry and nonlinear dynamics underlying excitability phenotypes in biophysical models of membrane potential*. text; electronic dissertation, The university of Arizona.

Herrera-Valdez, M. A. (2015). A unifying theory to describe transmembrane transport derived from thermodynamic principles. *PrePrints*, pages 0–21.

Herrera-Valdez, M. A. (2018a). A thermodynamic description for physiological transmembrane transport. pages 1–25.

Herrera-Valdez, M. A. (2018b). A thermodynamic description for physiological transmembrane transport [version 2; referees: 2 approved]. *F1000Research*, 7(May):1–29.

Hines, M., Davison, A. P., and Muller, E. (2009). Neuron and python. *Frontiers in neuroinformatics*, 3:1.

Hodgkin, A. L. and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, pages 500–544.

Husband, Mark, Nguyen, Dung, and Wong, Stephen (2008). Principles of Object-Oriented Programming.

Iserles, A. (2012). Runge-Kutta Methods. In *A First Course in the Numerical Analysis of Differential Equations*, chapter 1.3, pages 32–52. Cambridge University Press.

Johnston and Wu (1995). *Foundations of cellular neurophysiology*.

Kandel, E., Kandel, E., Schwartz, J., and Jessell, T. (2000). *Principles of Neural Science, Fourth Edition*. McGraw-Hill Companies,Incorporated.

Kaw, A. (2011a). On Solving Higher Order Equations for Ordinary Differential Equations. In *Numerical Methods With Applications*, chapter 8.5, pages 1–9.

Kaw, A. (2011b). Runge-Kutta 4th Order Method for Ordinary Differential Equations. In *Numerical Methods With Applications*, chapter 8.4, page 7.

Kelly, S. (2016). *Python, PyGame and Raspberry Pi Game Development Python, PyGame and Raspberry Pi Game Development Python, PyGame and Raspberry Pi Game Development*.

Langtangen, H. P. (2004). *Python Scripting for Computational Science*, volume 3.

Mallot, H. A. (2013). *Computational Neuroscience*, volume 2 of *Springer Series in Bio-/Neuroinformatics*. Springer International Publishing, Heidelberg.

Miura, R. M. (2002). Analysis of excitable cell models. *Journal of Computational and Applied Mathematics*, 144(1-2):29–47.

Mulansky, M. and Ahnert, K. (2014). Odeint library. *Scholarpedia*, 9(12):32342.

Muller, E., Bednar, J. A., Diesmann, M., Gewaltig, M.-o., Hines, M., and Davison, A. P. (2015). *Python in Neuroscience*. Frontiers Research Topics. Frontiers Media SA.

Norris, M. and Rigby, P. (1994). *Ingenieria de Software Explicada / Software Engineering Explained*. Area: computación. Editorial Limusa S.A. De C.V.

Ortega Arjona, J. L. (2005). The Practical Design Method : A Software Design Method for a First Object-Oriented Project. 9:41–54.

Qt (2019). About Qt - Qt Wiki. `https://wiki.qt.io/About{_}Qt`.

Rinzel, J. (1985). Excitation dynamics: insights from simplified membrane models. In *Fed. Proc*, volume 44, pages 2944–2946.

Scipy (2019). scipy odeint. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html`.

Severance, C. (2015). Guido van Rossum: The early years of python. *Computer*, 48(2):7–9.

Stefik, M. and Bobrow, D. G. (1985). Object-oriented programming: Themes and variations. *AI magazine*, 6(4):40–40.

Stepleman, R. S. R. S. (1983). *Scientific computing : applications of mathematics and computing to the physical sciences*. North-Holland Pub. Co.

Summerfield, M. (2008). Rapid Gui Programming with Python and Qt: The Definite Guid to PyQt Programming. *Rapid GUI Programming with Python and Qt:*, 54(2):NP.

Tikidji-Hamburyan, R. A., Narayana, V., Bozkus, Z., and El-Ghazawi, T. A. (2018). Corrigendum: Software for Brain Network Simulations: A Comparative Study. *Frontiers in Neuroinformatics*, 12(July):1–16.

Torlak, E. (2016). Case use diagram. `https://courses.cs.washington.edu/courses/cse403/16au/lectures/L04.pdf`.

Verkerk, A. O., Van Borren, M. M., and Wilders, R. (2013). Calcium transient and sodium-calcium exchange current in human versus rabbit sinoatrial node pacemaker cells. *The Scientific World Journal*, 2013.

Weiss, T. F. (1996). *Cellular biophysics*, volume 1. MIT press Cambridge, Mass:.

Wikimedia (2019). Model-view-controller pattern. https://upload.wikimedia.org/wikipedia/commons/thumb/b/b4/MVC_Diagram_%28Model-View-Controller%29.svg/440px-MVC_Diagram_%28Model-View-Controller%29.svg.png.

Ztaou, S. and Amalric, M. (2019). Contribution of cholinergic interneurons to striatal pathophysiology in parkinson's disease. *Neurochemistry international*.