UNIVERSIDADE DE SÃO PAULO
FACULDADE DE FILOSOFIA, CIÊNCIAS E LETRAS DE RIBEIRÃO PRETO
PROGRAMA DE FÍSICA APLICADA À MEDICINA E BIOLOGIA

Programa de computador para simulação de modelos de neurônios: aplicação à célula mitral do bulbo olfatório

Rafael Arantes

Ribeirão Preto
2011

UNIVERSIDADE DE SÃO PAULO
FACULDADE DE FILOSOFIA, CIÊNCIAS E LETRAS DE RIBEIRÃO PRETO
PROGRAMA DE FÍSICA APLICADA À MEDICINA E BIOLOGIA

Programa de computador para simulação de modelos de neurônios: aplicação à célula mitral do bulbo olfatório

Rafael Arantes

Dissertação submetida ao Programa de Pós Graduação em Física Aplicada à Medicina e Biologia da Faculdade de Filosofia, Ciências e Letras de Ribeirão Preto da Universidade de São Paulo como parte dos requisitos para a obtenção do título de Mestre em Física Aplicada à Medicina e Biologia.

Orientador: Prof. Dr. Antônio Carlos Roque da Silva Filho

Ribeirão Preto, Março de 2011.

R. Arantes
Programa de computador para simulação de modelos de neurônios: aplicação à célula mitral do bulbo olfatório / Rafael Arantes; orientador Prof. Dr. Antônio Carlos Roque da Silva Filho. – Ribeirão Preto/SP, 2011.
121 p.

Dissertação (Mestrado – Programa de Pós-Graduação em Física Aplicada à Medicina e Biologia) – Faculdade de Filosofia, Ciências e Letras de Ribeirão Preto da Universidade de São Paulo.

Neurônio, modelo biofísico, canal iônico, equação diferencial, célula mitral, bulbo olfativo

Banca Examinadora:

Prof. Dr. Antonio Carlos Roque da Silva Filho - Orientador

Prof. Dr. Renato Tinós

Prof. Dr. Reynaldo Daniel Pinto

DEDICATÓRIA

A todos que direta ou indiretamente contribuiram para a realização deste trabalho e para o aprendizado que adquiri no seu decorrer.

Agradecimentos

Ao meu orientador pelo apoio, sugestões e ajudas, sem as quais eu não teria concluído este trabalho.

A todos os professores da minha graduação e pós pelos conhecimentos transmitidos, de vital importancia para que eu chegasse até aqui.

Aos meus pais pelo apoio incondicional e compreensão.

Aos amigos do laboratório pelo apoio e ajudas e, todos meus amigos pelas distrações necessárias para recuperar minhas energias.

Aos funcionários, sempre dedicados e prestativos.

Ao Departamento de Física e Matemática pelo suporte ao trabalho.

À Coordenadoria de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio financeiro.

Resumo

R. ARANTES. **Programa de computador para simulação de modelos de neurônios: aplicação à célula mitral do bulbo olfatório**. Dissertação (Mestrado) – Faculdade de Filosofia, Ciências e Letras de Ribeirão Preto, Universidade de São Paulo, Ribeirão Preto, 2011.

O presente trabalho descreve um programa de computador em linguagem Java que reproduz o modelo compartimental reduzido de célula mitral do bulbo olfativo construído por Davison, Feng e Brown (*Brain Res. Bull. 51:393-399, 2000*), como uma simplificação do modelo detalhado de Bhalla e Bower (*J. Neurophysiol., 69:1948-1965, 1993*). O modelo reduzido considera a célula mitral como composta por quatro compartimentos, modelados conforme a metodologia de HODGKIN e HUXLEY. Por seu baixo custo computacional, o modelo reduzido permite a construção de modelos de rede de grande porte para o bulbo olfativo. A implementação computacional feita em Java apresenta grande similaridade com a original, indicando uma robustez do modelo com relação a versões em plataformas distintas.

Palavras-chave: Neurônio, modelo biofísico, canal iônico, equação diferencial, célula mitral, bulbo olfativo

Abstract

R. ARANTES. **Computer program for neuron models simulation: application to the olfactory bulb mitral cell**. Dissertation (Master) – Faculdade de Filosofia, Ciências e Letras de Ribeirão Preto, Universidade de São Paulo, Ribeirão Preto, 2011.

This work describes a computer program written in Java, which reproduces the reduced compartimental model of the mitral cell of the olfactory bulb constructed by Davison, Feng and Brown (*Brain Res. Bull. 51:393-399, 2000*), as a simplified version of the detailed model of Bhalla and Bower (*J. Neurophysiol., 69:1948-1965, 1993*). The reduced model considers the mitral cell as composed of four compartiments modeled according to the Hodgkin-Huxley formalism. Due to its low computational cost, the reduced model allows the construction of large-scale network models of the olfactory bulb. The computer implementation made in Java shows great similarity with the original, indicating that the model is robust with respect to implementations in different platforms.

Keywords: Neuron, biophysical model, ionic channel, differentials equations, mitral cell, olfactory bulb

Lista de Figuras

Lista de Tabelas

Sumário

1 Introdução

A simulação computacional de neurônios necessita, para sua realização, primeiramente de conhecimentos sobre os próprios neurônios e, em segundo lugar, de modelos matemáticos que representem os mecanismos biofísicos por trás do funcionamento de neurônios. Esses modelos matemáticos são expressos em termos de equações diferenciais, o que torna necessário também o conhecimento de técnicas e ferramentas para a resolução de equações diferenciais. De posse desses conhecimentos, é possível construir simulações computacionais de neurônios para estudar como eles se comportam em diferentes situações e qual o papel dos diferentes parâmetros do modelo sobre o seu comportamento.

Existem excelentes pacotes computacionais para a simulação de neurônios, muitos deles de domínio público, como GENESIS (BOWER; BEEMAN, 1998) e NEURON (CARNEVALE; HINES, 2004). Tais pacotes foram feitos para ser executados nas plataformas computacionais mais populares usadas por grupos de pesquisa ao redor do mundo. Essas plataformas são, em geral, customizadas para atender as peculiaridades locais de cada grupo. Isso leva à possibilidade de que simulações computacionais de um modelo de neurônio implementadas por um grupo trabalhando em um determinado ambiente computacional não produzam exatamente os mesmos resultados quando implementadas por outro grupo em outro ambiente computacional.

O objetivo deste trabalho é implementar em outra linguagem, um modelo computacional de neurônio originalmente construído em NEURON. Com isso, será possível testar se a simulação do modelo nas duas lingagens apresenta os mesmos resultados.

A linguagem escolhida para a implementação do modelo deste trabalho é a linguagem Java (GOSLING et al., 2005). Ela foi escolhida por ser multi-plataforma, de maneira que um programa desenvolvido e compilado em um dado ambiente computacional pode ser executado em outro ambiente.

O modelo de neurônio escolhido é o da célula mitral do bulbo olfativo, desenvolvido por Davison, Feng e Brown (2000). Esse modelo é uma versão simplificada, mas bastante acurada, do modelo detalhado de Bhalla e Bower (1993). Escolheu-se esse modelo porque ele, apesar

de ser pequeno para permitir simulações de baixo custo, possui diversos elementos de razoável complexidade (por exemplo, mais de um compartimento, canais iônicos dependentes de cálcio) para tornar mais visíveis diferenças entre a simulação original e a deste trabalho, caso existam.

A seguir, faz-se uma breve descrição do estilo de modelagem de neurônios adotado neste trabalho (compartimental baseado em condutância) e de técnicas de soluções de equações diferenciais.

1.1 Modelagem de Neurônios

A simulação computacional de neurônios e redes de neurônios pode ser de vários tipos, sendo o baseado em condutância o tipo escolhido para este trabalho. Este tipo de modelo considera a dinâmica de funcionamento dos neurônios com base em variáveis como distribuição e comportamento de canais iônicos no neurônio. Eles são geralmente chamados modelos de neurônios *à la* Hodgkin-Huxley, em referência ao primeiro modelo desta linha feito por HODGKIN e HUXLEY (1952) a partir de medidas realizadas em um axônio gigante de lula.

Na ausência de canais iônicos a membrana pode ser considerada um isolante de cargas elétricas que separa as cargas, os íons, dos meios intracelular e extracelular. Esta característica determina que a membrana se comporta como um capacitor elétrico com uma dada capacitância C.

Porém, a membrana possui canais iônicos que permitem, com maior ou menor facilidade, a passagem de íons específicos, de acordo com a diferença entre o potencial de membrana atual (V_m) e potencial de Nernst do íon ($E_{íon}$). Isto pode ser modelado como um resistor de resistência $R_{íon}$ associado em série com uma bateria de potencial $E_{íon}$. Este modelo de canal iônico pode ser usado para reproduzir as características de cada tipo de canal iônico. Desta maneira, um modelo de circuito elétrico composto por um capacitor em paralelo com um resistor representando as diferentes correntes iônicas presentes na membrana constitui uma boa representação da membrana neuronal.

Dessa forma a corrente que atravessa a membrana para gerar os potenciais de ação fica dada por uma soma da componente capacitiva(I_C), devida ao rearranjo de cargas, e da componente resistiva ($I_{íon}$), devida a passagem de cargas pela membrana, ou seja, $I_m = I_C + I_{íon} = C \cdot \frac{dV_m(t)}{dt} + \sum_{íon} G_{íon} \cdot (V_m - E_{íon})$, onde $G_{íon} = \frac{1}{R_{íon}}$ é a condutância da membrana ao íon.

Esta condutância iônica, porém, não é constante, mas varia de acordo com a porcentagem de canais daquele tipo que estão abertos a cada instante. Por isso ela é modelada como o produto entre uma condutância máxima ($\overline{g_{íon}}$) e um conjunto de variáveis de ativação e inativação, cuja mudança de valor reflete a mudança da porcentagem de canais abertos a cada instante.

A partir dos experimentos realizados com o axônio gigante de lula, HODGKIN e HUXLEY (1952) puderam obter curvas fenomenológicas para ajustar a variação das condutâncias da membrana deste axônio aos íons de sódio (Na) e de potássio (K) em função da diferença de potencial V_m e do tempo. As equações a seguir descrevem o modelo de forma que se possa interpretar os seus parâmetros como será descrito:

$$C_m \cdot \frac{dV_m}{dt} = \overline{g_{Na}} \cdot m^3 \cdot h \cdot (V_m - E_{Na}) + \overline{g_K} \cdot n^4 \cdot (V_m - E_K) + \overline{g_{Vaz}} \cdot (V_m - E_{Vaz}) + I_{inj}(t) \quad (1.1)$$

$$\frac{dn}{dt} = \alpha_n \cdot (1-n) - \beta_n \cdot n \quad (1.2)$$

$$\alpha_n = \frac{10 - V_m}{100 \cdot \left(e^{\frac{10-V_m}{10}} - 1\right)} \quad (1.3)$$

$$\beta_n = 0,125 \cdot e^{\frac{-V_m}{80}} \quad (1.4)$$

$$\frac{dm}{dt} = \alpha_m \cdot (1-m) - \beta_m \cdot m \quad (1.5)$$

$$\alpha_m = \frac{25 - V_m}{10 \cdot \left(e^{\frac{25-V_m}{10}} - 1\right)} \quad (1.6)$$

$$\beta_m = 4 \cdot e^{\frac{-V_m}{18}} \quad (1.7)$$

$$\frac{dh}{dt} = \alpha_h \cdot (1-h) - \beta_h \cdot h \quad (1.8)$$

$$\alpha_h = 0,07 \cdot e^{\frac{-V_m}{20}} \quad (1.9)$$

$$\beta_h = \frac{1}{e^{\frac{30-V_m}{10}} + 1} \quad (1.10)$$

Nestas equações, $I_{inj}(t)$ é a corrente injetada na célula e os parâmetros m, n e h são as variáveis de ativação e inativação que controlam a probabilidade de um dado canal iônico estar aberto ou não. Caso essas variáveis estejam elevadas uma potência i, será dito que ela se repete i vezes naquele canal. Elas são dependentes de taxas variáveis descritas pelas funções α e β. Existem diversos outros canais iônicos já estudados e modelados e a maioria segue o esquema geral de modelagem apresentado acima.

O modelo de Hodgkin-Huxley representa um neurônio pontual, sem extensão espacial. Modelos com extensão espacial se baseiam no chamado modelo do cabo, que trata a célula como um cabo elétrico contínuo. Uma maneira de discretizar esse modelo é pela chamada técnica

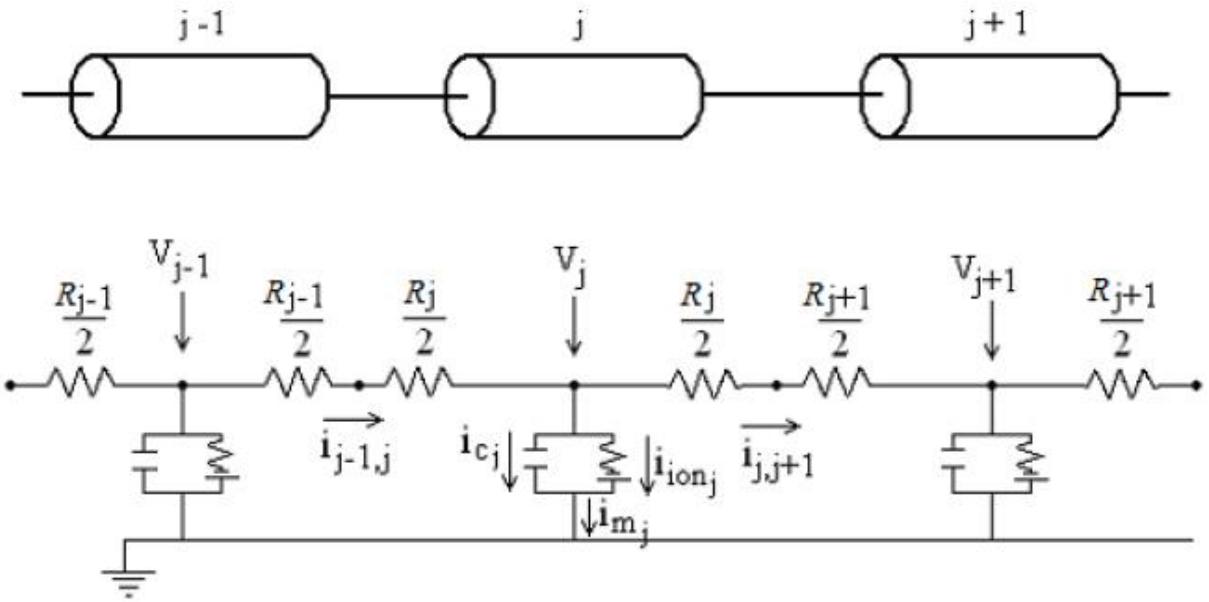

Figura 1.1: Esquema geral de modelo compartimental de neurônio

de compartimentalização de RALL (1959). Ela discretiza o modelo do cabo contínuo estabelecendo pedaços que podem ser aproximados por modelos pontuais interligados por resistências axiais. Um esquema desse modelo é apresentado na figura 1.1.

O pedaço de neurônio mostrado na figura 1.1 pode ser representado pelas seguintes equações:

$$I_{m_j} = I_{j-1,j} - I_{j,j+1} = C_{m_j} \cdot \frac{dV_j}{dt} + I_{\acute{i}on_j} \tag{1.11}$$

$$I_{j-1,j} = \frac{V_{j-1} - V_j}{R_{j-1,j}} \tag{1.12}$$

$$R_{j-1,j} = \frac{R_{j-1} + R_j}{2} \tag{1.13}$$

Combinando todas as equações acima, chega-se a equação que modela o potencial de membrana:

$$C_{m_j} \cdot \frac{dV_j}{dt} + I_{\acute{i}on_j} = \frac{V_{j-1} - V_j}{R_{j-1,j}} - \frac{V_j - V_{j+1}}{R_{j,j+1}} + I_{inj_j} = G_{j-1,j} \cdot \left(V_{j-1} - V_j\right) - G_{j,j+1} \cdot \left(V_j - V_{j+1}\right) + I_{inj_j} \tag{1.14}$$

1.2 Equações Diferenciais e seus Métodos de Resolução

Segundo Farlow (1994) uma equação diferencial ordinária de ordem n é uma equação que possui a seguinte fórmula geral:

$$F(x, y, y', y'', \ldots, y^{(n)}) = 0 \tag{1.15}$$

onde x é o termo independente, y é um termo que depende de x, e os termos restantes são as derivadas de y em relação a x, isto é, $\frac{dy}{dx}$, $\frac{d^2y}{dx^2}$, ..., $\frac{d^ny}{dx^n}$. Ainda segundo o mesmo autor estas equações são ditas lineares quando podem ser escritas na forma:

$$\sum_{i=0}^{n} a_i(x) \cdot \frac{d^i y}{dx^i} = f(x) \tag{1.16}$$

onde $\frac{d^0y}{dx^0}$ deve ser entendido como y, e $a_i(x)$ e $f(x)$ são funções de x. Estas equações são ditas homogêneas caso $f(x) = 0$ e não-homogêneas caso contrario.

Um sistema de equações diferenciais ordinárias é uma equação que siga a forma 1.15 exceto pelo fato de y ser um vetor $\vec{y}$ que pode ter sua dimensão maior que 1.

Sua resolução analítica é bastante dependente da forma particular de cada sistema e equação que o compõe. Com o propósito de tonar mais genérica a maneira de resolver esses problemas e desta forma sistematizar o processo a ponto de que ele possa ser implementado em um software, estudam-se métodos de aproximação da resolução chamados métodos de integração numérica.

Esses métodos utilizam um valor de $y(x_0) = y_0$ dado para calcular uma aproximação de $y_i \approx y(x_0 + \delta \cdot i)$ onde δ é o passo de integração utilizado pelo método. Eles se baseiam no teorema de Taylor, expresso por Butcher (2003) na forma

$$y(a+\delta) = y(a) + y'(a)(\delta) + \frac{1}{2!}y''(a)(\delta,\delta) + \cdots + \frac{1}{n!}y^{(n)}(a)(\delta,\delta,\cdots,\delta) + R_n$$

onde R_n é o termo (a ser ignorado na aproximação) referente ao erro cometido ao se realizar a aproximação e é da ordem de (δ^n).

Com base nesta fórmula é construído o método de Taylor de ordem n, que exige o conhecimento de todas as derivadas de ordem menor ou igual a n de y. Para contornar essa exigência foram desenvolvidos métodos que fazem aproximações desses valores por médias ponderadas dos valores da função e sua primeira derivada em pontos de coordenada $x \leq x_0 + \delta \cdot i$ na iteração em se pretende calcular o valor y_i.

Estes métodos se dividem entre os que utilizam apenas os pontos de coordenada $x \leq x_0 + \delta \cdot$

$(i-1)$ chamados métodos multi-passo e aos que não utilizam esses pontos chamados métodos de Runge-Kutta. Há ainda métodos que mesclam as abordagens multi-passo e Runge-Kutta.

Outra decisão a ser tomada pelo método é quanto ao tamanho do passo (δ) a ser dado a cada iteração da resolução. Quanto a esse critério há dois tipos de métodos: os que adotam passos de tamanho fixo e de tamanho variável, que muda de acordo com características da função no ponto em que se está realizando o cálculo. Esta variação geralmente é feita da seguinte forma:

1. escolhe-se o método (M_r) de ordem n a ser utilizado para resover;
2. escolhe-se o método (M_a) de ordem maior que n a ser utilizado para avaliar o erro;
3. estima-se o erro como a diferença entre os valores calculados por M_r e M_a;
4. de acordo com o erro estimado e uma tolerância pré-definida almenta-se, diminue-se ou aceita-se o tamanho atual do passo.

1.2.1 stiffness

Um tipo especial de sistema de equações diferenciais que é encontrado com bastante frequência em problemas científicos é o de sistemas que possuem forte estabilidade em muitos pontos, mas que podem se tornar fortemente instáveis em outros pontos (LAMBERT, 1991, Cap. 6 e 7). Esse tipo de comportamento é conhecido como *stiffness* e seu exemplo mais simples e clássico é a equação diferencial $y'(x) = \lambda \cdot y$. Observando o comportamento das equações diferenciais frequentemente utilizadas na área de neurociência computacional, percebe-se que elas se enquadram nesse tipo de caso (eqs. 1.1 - 1.10).

Como *stiffness* está relacionado a perturbações de uma solução de uma equação diferencial, pode-se tomar a equação diferencial genérica

$$y'(x) = f(x, y(x)) \tag{1.17}$$

e introduzir uma pequena perturbação, que também varie com x, na solução $y(x)$ substituindo-a por $y(x) + \varepsilon Y(x)$, onde ε é pequeno o suficiente para que se possa considerar $\varepsilon^\alpha \approx 0 \ \forall \alpha > 1$. Dessa forma, pode-se expandir f em série de Taylor e obter

$$y'(x) + \varepsilon Y'(x) = f(x, y(x)) + \varepsilon \frac{\partial f}{\partial y} Y(x) \tag{1.18}$$

Subtraindo 1.17 de 1.18 e cancelando ε obtém-se

$$Y'(x) = \frac{\partial f}{\partial y} Y(x) = J(x) \cdot Y(x)$$

onde J(x) é a matriz jacobiana do sistema $y(x)$.

Tomando um intervalo Δx de forma a causar alterações moderadas em $Y(x)$ e pouca alteração em $J(x)$ os autovalores de $J(x)$ determinarão o crescimento dos componentes da perturbação. A presença de um ou mais valores (λ) de $J(x)$ grandes e negativos quase certamente implicará em *stiffness* .

2 *Desenvolvimento*

Este capítulo está dividido em duas partes. Na primeira, apresenta-se o modelo de célula mitral proposto por Davison, Feng e Brown (2000). Na segunda, o programa implementado em Java para simular esse modelo é descrito.

2.1 O Modelo Original da Célula Mitral

O modelo de célula mitral de Davison, Feng e Brown (2000) foi construído originalmente no neurossimulador NEURON. O seu código esta disponível no ModelDB (HINES et al., 2004) como parte de um modelo do bulbo olfativo.

O modelo é composto por 4 compartimentos, a saber os dendritos glomerular, primário e distal e o soma. Há em cada um destes compartimentos canais iônicos de diferentes tipos e em diferentes concentrações. Cada tipo de canal é modelado por um conjunto de equações diferenciais e as concentrações destes canais são consideradas pela condutância máxima atribuída a cada canal.

A tabela 2.1 apresenta os tipos de canais presentes em cada compartimento com suas respectivas condutâncias máximas dadas em mS/cm^2. As linhas da tabela se referem ao compartimento e as colunas aos canais de sódio (Na), potássio com corrente rápida (Kfast), com corrente lenta (K), anômala (KA) e dependente da concentração de cálcio (KCa), canais de cálcio (Ca) e canais de vazamento. Além dessas informações, faz-se necessário para a construção dos modelos de canais, conhecer seus potenciais de reversão, que são: 45mV para os canais de sódio, -70mV para os canais de potássio com exceção do dependente de cálcio, onde é -80mV, 70mV para os canais de cálcio e -65mV para os canais de vazamento.

Cada canal é modelado de acordo com o formalismo de Hodgkin-Huxley. As variáveis de

	Na	Kfast	K	KA	KCa	Ca	Vazamento
Glomérulo	0	0	20	0	0	9,5	0,01
Primário	1,34	1,23	1,74	0	0	2,2	0,01
Soma	153,2	195,6	2,8	5,87	14,2	4	0,01
Distal	12,2	12,8	0	0	0	0	0,01

Tabela 2.1: Tipos de canais presentes em cada compartimento com suas respectivas condutâncias máximas dadas em mS/cm^2. As linhas se referem ao compartimento e as colunas aos canais de sódio (Na), canais de potássio com corrente rápida (Kfast), com corrente lenta (K), anômala (KA) e dependente da concentração de cálcio (KCa), canais de cálcio (Ca) e canais de vazamento

ativação e inativação do sódio são indicadas, respectivamente, por "m" e "h", sendo três da primeira e uma da segunda; as do cálcio, por "r" e "s". O canal de potássio dependente da concentração de cálcio só possui uma variável ("y"). As taxas de abertura e fechamento desses canais são modeladas pelas seguintes equações diferenciais:

$$\alpha_m = 0,32 \cdot \frac{V+42}{1-e^{\frac{V+42}{4}}} \tag{2.1}$$

$$\beta_m = 0,28 \cdot \frac{V+15}{e^{\frac{V+15}{5}}-1} \tag{2.2}$$

$$\alpha_h = \frac{0,128}{e^{\frac{V+38}{18}}} \tag{2.3}$$

$$\beta_h = \frac{4}{1+e^{\frac{V+15}{5}}} \tag{2.4}$$

$$\alpha_r = \frac{0,0068}{1+e^{\frac{V+30}{12}}} \tag{2.5}$$

$$\beta_r = \frac{0,06}{1+e^{\frac{-V}{11}}} \tag{2.6}$$

$$\alpha_s = \frac{7,5}{1+e^{\frac{13-V}{7}}} \tag{2.7}$$

$$\beta_s = \frac{1,65}{1+e^{\frac{V-14}{4}}} \tag{2.8}$$

$$\alpha_y = depV \cdot depCa = e^{\frac{V+70}{27}} \cdot \frac{500 \cdot (0,015 - min([Ca];0,01))}{e^{\frac{0,015-min([Ca];0,01)}{0,0013}}-1} \tag{2.9}$$

$$\beta_y = 0,05 \tag{2.10}$$

$$\tag{2.11}$$

onde $depCa$ denota a dependencia da concentração de cálcio ($[Ca]$), $depV$ a dependencia do potencial de membrana (V), e $min([Ca];0,01)$ significa o mínimo entre o valor da concentração

de cálcio e $0,01$.

A concentração de cálcio em cada compartimento é calculada considerando-se a entrada de cálcio no compartimento por meio da corrente no canal de cálcio (I_{Ca}) menos o seu decaimento causado pela difusão no meio. A equação diferencial 2.12 representa esta dinâmica, onde é considerando o limite mínimo da concentração ($[Ca]_\infty$) igual a 10^{-5} e a constate de tempo (τ) igual a 10.

$$[Ca]' = fluxo - decaimento = \frac{I_{Ca} \cdot 10^4}{2 \cdot 96154} - \frac{[Ca] - [Ca]_\infty}{\tau} \tag{2.12}$$

As variáveis de ativação e inativação dos canais restantes são descritos por outro tipo de equação que não as baseadas em α e β, mas sim em valores estacionários e tempos característicos de cada uma. As dos canais lentos de potássio são indicadas por "a", que aparece uma vez, e "b", que aparece duas vezes. Os canais de potássio rápido também possuem três variáveis, sendo uma do tipo indicado por "k" e outras duas do tipo indacado por "n". Já os canais de potássio anômalos possuem duas variáveis, indicadas por "p" e "q". Estas variáveis são modeladas pelas seguintes equações:

$$\frac{da}{dt} = \frac{a_\infty - a}{\tau_a} \tag{2.13}$$
$$\tau_a = 200 \tag{2.14}$$
$$\frac{db}{dt} = \frac{b_\infty - b}{\tau_b} \tag{2.15}$$
$$\frac{dk}{dt} = \frac{k_\infty - k}{\tau_k} \tag{2.16}$$
$$\tau_k = 50 \tag{2.17}$$
$$\frac{dn}{dt} = \frac{n_\infty - n}{\tau_n} \tag{2.18}$$
$$\frac{dp}{dt} = \frac{p_\infty - p}{\tau_p} \tag{2.19}$$
$$p_\infty = \frac{1}{1 + e^{-\frac{V+42}{13}}} \tag{2.20}$$
$$\tau_p = 1,38 \tag{2.21}$$
$$\frac{dq}{dt} = \frac{q_\infty - q}{\tau_q} \tag{2.22}$$
$$q_\infty = \frac{1}{1 + e^{\frac{V+110}{18}}} \tag{2.23}$$

$$\tau_q = 150 \tag{2.24}$$

$$\tag{2.25}$$

Os comportamentos de a_∞, b_∞, τ_b, k_∞, n_∞ e τ_n são dados por valores tabelados, não havendo equações para calculá-los.

2.2 O Programa

O programa desenvolvido foi dividido em duas partes, uma responsável pelo modelo em si, considerando a célula, os compartimentos, os canais e as variáveis de ativação/inativação aqui denominadas portões, e a outra responsável pelas necessidades básicas da simulação e não diretamente ligada ao modelo, como solucionadores de equações diferenciais e construtores de gráficos. A primeira parte foi denominada pacote `mitral` e a segunda `util`.

2.2.1 Pacote `util`

O pacote `util` é responsável pelas necessidades básicas da simulação e não está diretamente ligado ao modelo, ele é composto pela interface `Mapeavel` e as classes `FimSimulacao`, `Disparo`, `Grafico` e `Tabela`, contendo ainda o pacote `math.calculoNumerico`.

A interface `Mapeavel` deve ser implementada por toda classe que necessite ter sua evolução representada em um gráfico, seu código é o seguinte.

```
1  package util;
2
3  /**
4   * Mapeavel do pacote util do projeto tcc.<br>
5   * Interface que representa objetos que relacionam a cada valor x um valor y
6   * @author Rafael Arantes
7   */
8  public interface Mapeavel
9  {
10     /**
11      * @return o valor de x considerado pelo objeto
12      */
13     public double getX();
14
```

```
15      /**
16       * @return o valor de y considerado pelo objeto
17       */
18      public double getY();
19
20      /**
21       * @return o nome do objeto para identifica-lo
22       */
23      public String getNome();
24  }
```

A classe `FimSimulacao` é responsável por indicar o término de uma simulação, a seguir apresento seu código.

```
1   package util;
2
3   import java.awt.event.ActionEvent;
4
5   /**
6    * FimSimulacao do pacote util do projeto tcc.<br>
7    * @author Rafael Arantes
8    */
9   public class FimSimulacao extends ActionEvent
10  {
11      /**
12       * long que representa serialVersionUID
13       */
14      private static final long serialVersionUID = 1L;
15
16      /**
17       * Constroi um novo FimSimulacao
18       * @param fonte objeto que gerou este evento
19       * @param id identificação do evento
20       * @param explicacao texto explicativo sobre o evento
21       */
22      public FimSimulacao(Object fonte, int id, String explicacao)
23      {
24          super(fonte, id, explicacao);
25      }
26  }
```

A classe `Disparo` é responsável por indicar a ocorrencia de um potencial de ação no neurônio em uma simulação, a seguir apresento seu código.

```
1   package util;
2
```

```
3   import java.awt.event.ActionEvent;
4
5   /**
6    * Disparo do pacote util do projeto tcc.<br>
7    * @author Rafael Arantes
8    */
9   public class Disparo extends ActionEvent
10  {
11      /**
12       * long que representa serialVersionUID
13       */
14      private static final long serialVersionUID = 1L;
15
16      /**
17       * Constroi um novo Disparo
18       * @param fonte objeto que gerou este evento
19       * @param id identificação do evento
20       * @param explicacao texto explicativo sobre o evento
21       */
22      public Disparo(Object fonte, int id, String explicacao)
23      {
24          super(fonte, id, explicacao);
25      }
26  }
```

A classe `Grafico` é responsável por disponibilizar gráficos para plotar os potenciais dos compartimento ao longo da simulação, ela realiza uma mera comunicação entre a simulação e dois pacotes para construção de gráficos disponíveis em http://www.jfree.org/index.html. Seu código pode ser visto a seguir.

```
1   package util;
2
3   import java.awt.BorderLayout;
4   import java.io.File;
5   import java.io.IOException;
6
7   import javax.swing.JFrame;
8
9   import org.jfree.chart.ChartFactory;
10  import org.jfree.chart.ChartPanel;
11  import org.jfree.chart.ChartUtilities;
12  import org.jfree.chart.JFreeChart;
13  import org.jfree.chart.plot.PlotOrientation;
14  import org.jfree.chart.title.LegendTitle;
15  import org.jfree.data.xy.XYDataset;
16  import org.jfree.data.xy.XYSeries;
17  import org.jfree.data.xy.XYSeriesCollection;
18
```

```
19  /**
20   * Grafico do pacote util do projeto tcc.<br>
21   * Classe que implementa as funcionalidades nescessarias para a exibição de um
22   * grafico.
23   * @author Rafael Arantes
24   */
25  public class Grafico
26  {
27      private JFreeChart chart;
28      private XYSeriesCollection dataset;
29      private XYSeries serie;
30      private LegendTitle legenda;
31      private boolean hold;
32      private JFrame janela;
33      private ChartPanel painel;
34
35      /**
36       * Constroi um novo grafico plotando os dados pelos seus indices.
37       * @param dados dados q serem plotados no grafico.
38       */
39      public Grafico(double[] dados)
40      {
41          // createDataset
42          dataset = new XYSeriesCollection();
43          createSerie(dados);
44          painel = createPanel(dataset);
45          criaJanela(painel);
46      }
47
48      /**
49       * Constroi um novo grafico plotando os pontos passados.
50       * @param pontos as coordenadas x que devem ser plotadas.
51       * @param valores os valores a serem plotados nesses pontos.
52       */
53      public Grafico(double[] pontos, double[] valores)
54      {
55          // createDataset
56          dataset = new XYSeriesCollection();
57          createSerie(pontos, valores);
58          painel = createPanel(dataset);
59          criaJanela(painel);
60      }
61
62      /**
63       * Constroi um novo Grafico
64       * @param rotuloX significado do eixo x do grafico
65       * @param rotuloY significado do eixo y do grafico
66       */
67      public Grafico(String rotuloX, String rotuloY)
68      {
69          // createDataset
```

```
70          dataset = new XYSeriesCollection();
71          createSerie();
72          painel = createPanel(dataset, rotuloX, rotuloY);
73          criaJanela(painel);
74      }
75
76      private void criaJanela(ChartPanel painel)
77      {
78          janela = new JFrame("Grafico");
79          janela.setLayout(new BorderLayout());
80          janela.getContentPane().add(painel, BorderLayout.CENTER);
81          janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
82          janela.pack();
83          janela.setVisible(true);
84      }
85
86      /**
87       * Altera o titulo do grafico que é exibido na barra de titulo da janela que
88       * apresenta o grafico
89       * @param titulo novo titulo do grafico
90       */
91      public void setTitle(String titulo)
92      {
93          chart.setTitle(titulo);
94          janela.setTitle(titulo);
95      }
96
97      /**
98       * Cria um painel com o grafico, plotando nele os dados recebidos.
99       * @param dataset dados a serem plotados no grafico ja no formato requerido.
100      * @return painel criado.
101      */
102     private ChartPanel createPanel(XYDataset dataset, String rotuloX,
103             String rotuloY)
104     {
105         chart = ChartFactory.createXYLineChart(null, // titulo do grafico
106                 // (null permitted).
107                 rotuloX, // titulo do eixo x (null permitted).
108                 rotuloY, // titulo do eixo y.
109                 dataset, // dados a serem plotados (null permitted).
110                 PlotOrientation.VERTICAL, // orientação do grafico (horizontal
111                 // ou vertical) (null not
112                 // permitted).
113                 true, // se a legenda deve aparecer.
114                 true, // se os tool tips estão habilitados
115                 false); // configure chart to generate URLs?);
116         legenda = chart.getLegend();
117         // createFrame
118         return new ChartPanel(chart);
119     }
120
```

```
121         /**
122          * Cria um painel com o grafico, plotando nele os dados recebidos.
123          * @param dataset dados a serem plotados no grafico ja no formato requerido.
124          * @return painel criado.
125          */
126         private ChartPanel createPanel(XYDataset dataset)
127         {
128             return createPanel(dataset, "eixo X", "eixo Y");
129         }
130 
131         /**
132          * Coloca os dados no formato utilizado pelo grafico.
133          * @param pontos vetor das coordenadas x a serem plotadas.
134          * @param valores valor a ser plotado em cada ponto (coordenada y).
135          */
136         public void createSerie(double[] pontos, double[] valores)
137         {
138             createSerie();
139             int tam;
140             if (pontos.length > valores.length)
141                 tam = valores.length;
142             else
143                 tam = pontos.length;
144             for (int i = 0; i < tam; i++)
145                 serie.add(pontos[i], valores[i]);
146         }
147 
148         /**
149          * Coloca os dados no formato utilizado pelo grafico.
150          * @param dados vetor de dados a serem plotaods, o eixo x seguira o indice
151          *        dos dados.
152          */
153         public void createSerie(double[] dados)
154         {
155             createSerie();
156             for (int i = 0; i < dados.length; i++)
157                 serie.add(i, dados[i]);
158         }
159 
160         /**
161          * Cria uma nova serie de pontos no grafico
162          * @param nome o nome da serie a ser criada
163          */
164         public void createSerie(String nome)
165         {
166             if (!hold)
167                 dataset.removeAllSeries();
168             if (nome == null)
169                 nome = "Serie" + dataset.getSeriesCount();
170             XYSeries nova = new XYSeries(nome);
171             dataset.addSeries(nova);
```

```
172             serie = nova;
173         }
174 
175         /**
176          * Cria uma nova serie de pontos para o grafico sem nome
177          */
178         public void createSerie()
179         {
180             createSerie((String) null);
181         }
182 
183         /**
184          * Adiciona o ponto (x,y) ao grafico
185          * @param x coordenada x do ponto
186          * @param y coordenada y do ponto
187          */
188         public void addPonto(double x, double y)
189         {
190             serie.add(x, y);
191         }
192 
193         /**
194          * Torna a i-ésima série a serie ativa para os comandos de inserção de
195          * pontos
196          * @param i indice da serie a ser colocada como a ativa
197          */
198         public void setSerie(int i)
199         {
200             serie = dataset.getSeries(i);
201         }
202 
203         /**
204          * Segura as series que estão no grafico nele, permitindo adicionar novas
205          * ser altera-las
206          */
207         public void travar()
208         {
209             hold = true;
210         }
211 
212         /**
213          * Libera as séries do grafico, quando uma nova for adicionada as antigas
214          * serão perdidas
215          */
216         public void destravar()
217         {
218             hold = false;
219         }
220 
221         /**
222          * Exibe a legenda do grafico
```

```
223         */
224        public void mostraLegenda()
225        {
226            chart.addLegend(legenda);
227        }
228
229        /**
230         * Esconde a legenda do grafico
231         */
232        public void ocultaLegenda()
233        {
234            chart.removeLegend();
235        }
236
237        /**
238         * Torna a série <code>nome</code> a série ativa para os comandos de
239         * inserção de pontos
240         * @param nome serie que se deseja tornar ativa
241         */
242        public void setSerie(String nome)
243        {
244            for (int i = 0; i < dataset.getSeriesCount(); i++)
245                if (dataset.getSeries(i).getKey().equals(nome))
246                    setSerie(i);
247        }
248
249        /**
250         * Salva o grafico como uma imagem PNG
251         */
252        public void salvar()
253        {
254            int width = painel.getWidth();
255            String filename = chart.getTitle().getText() + ".png";
256            File file = new File("arquivos/" + filename);
257            int height = painel.getHeight();
258            try
259            {
260                ChartUtilities.saveChartAsPNG(file, chart, width, height);
261            }
262            catch (IOException e)
263            {
264                System.out.println("Erro ao salvar " + filename);
265                e.printStackTrace();
266            }
267        }
268
269        /**
270         * @return titulo do grafico
271         */
272        public String getTitulo()
273        {
```

```
274             return chart.getTitle().getText();
275         }
276     }
```

A classe `Tabela` é responsável por carregar e recuperar valores para as variáveis i_∞ e τ_i nos portões onde esses valores são dados por tabela, seu código é o seguinte.

```
1   package util;
2
3   import java.util.Vector;
4
5   /**
6    * Tabela do pacote util do projeto tcc.<br>
7    * Esta classe representa uma tabela com duas colunas de números reais, chave e
8    * valor. A coluna chave não adimite valores repetidos e as linhas serão
9    * inseridas na tabela de forma que ela fique ordenada pela chave. Desta forma é
10   * possivel recuperar um valor correspondente a uma chave se estiver na tabela
11   * ou ainda valores intermediarios entre dois quando a chave for intermediaria
12   * entre duas existentes.
13   * @author Rafael Arantes
14   */
15  public class Tabela
16  {
17      /**
18       * Vector<Double> que guarda as chaves
19       */
20      private Vector<Double> chave;
21      /**
22       * Vector<Double> que guarda os valores
23       */
24      private Vector<Double> valor;
25
26      /**
27       * Cria uma nova Tabela
28       */
29      public Tabela()
30      {
31          chave = new Vector<Double>();
32          valor = new Vector<Double>();
33      }
34
35      /**
36       * Realiza uma busca binaria na tabela pela chave passada, caso encontre
37       * retorna o valor correspondente a ela, caso contrario usa os valores
38       * correspondentes às chave vizinhas para calcular o valor intermediario a
39       * ser retornado.
40       * @param chave chave a ser buscada na tabela
41       * @return valor encontrado.
42       */
```

```
43      public double get(double chave)
44      {
45          int limInf = 0;
46          int limSup = this.chave.size() - 1;
47          if (chave > this.chave.lastElement())
48              return valor.lastElement();
49          int indice = (limSup + limInf) / 2;
50          do
51              if (chave == this.chave.get(indice))
52                  return valor.get(indice);
53              else
54              {
55                  if (chave > this.chave.get(indice))
56                      limInf = indice;
57                  else
58                      limSup = indice;
59                  indice = (limSup + limInf) / 2;
60              }
61          while (limSup - limInf > 1);
62          // valor intermediario na reta que passa pelo anterior e proximo
63          // |c1 v1 1|
64          // |c2 v2 1| = 0
65          // |ch va 1|
66          double c1 = this.chave.get(limInf);
67          double c2 = this.chave.get(limSup);
68          double v1 = valor.get(limInf);
69          double v2 = valor.get(limSup);
70          return (c2 * v1 - c1 * v2 - (v1 - v2) * chave) / (c2 - c1);
71      }
72
73      /**
74       * Insere por "insert-sort binaria" a nova chave com seu valor na tabela
75       * @param chave chave a ser inserida
76       * @param valor valor correspondente a chave
77       */
78      public void put(double chave, double valor)
79      {
80          if (this.chave.isEmpty())
81          {
82              this.chave.add(chave);
83              this.valor.add(valor);
84              return;
85          }
86          int limInf = 0;
87          int limSup = this.chave.size();
88          int indice = (limSup + limInf) / 2;
89          do
90              if (chave == this.chave.get(indice))
91              {
92                  this.chave.add(indice, chave);
93                  this.valor.add(indice, valor);
```

```
94                  return;
95              }
96              else
97              {
98                  if (chave > this.chave.get(indice))
99                      limInf = indice;
100                 else
101                     limSup = indice;
102                 indice = (limSup + limInf) / 2;
103             }
104         while (limSup - limInf > 1);
105         this.chave.add(limSup, chave);
106         this.valor.add(limSup, valor);
107     }
108 }
```

2.2.1.1 Pacote `math.calculoNumerico`

O pacote `math.calculoNumerico` reúne classes para manipulações matemáticas. Ele é composto pela interface `ODE` e as classes `ODESolver`, `ModifiedMidpoint`, `PrevisorCorretor`, `RungeKutta1`, `Euler`, `RungeKutta2`, `Heun`, `RungeKutta3`, `RungeKutta4` e `RungeKutta5`.

A interface `ODE` deve ser implementada por toda classe que deseje representar uma equação diferencial ordinária, seu código é o seguinte.

```
1  package util.math.calculoNumerico;
2
3  /**
4   * ODE do pacote util.math.calculoNumerico do projeto tcc.<br>
5   * Interface que representa uma equação diferencial a ser resolvida
6   * @author Rafael Arantes
7   */
8  public interface ODE
9  {
10
11     /**
12      * @return valor atual da função
13      */
14     double getY();
15
16     /**
17      * Expecifica a derivada dy/dx
18      * @param x valor do parametro da função
```

```
19          * @param y valor da função
20          * @return valor da derivada de y com relação a x no ponto (x,y) dado.
21          */
22         double fX(double x, double y);
23
24         /**
25          * @return valor de dt
26          */
27         double getDeltaT();
28
29         /**
30          * @return valor atual do parametro da função.
31          */
32         double getX();
33
34         /**
35          * Deve alterar o tamanho do passo de tempo
36          * @param dt novo tamanho do passo de tempo
37          */
38         void setDeltaT(double dt);
39
40         /**
41          * Deve alterar o valor atual da função
42          * @param y novo valor da função
43          */
44         void atualiza(double y);
45     }
```

A classe `ODESolver` representa todos os tipos de resolvedores de equações diferenciais, implementando os métodos comuns a todos eles e definindo os que devem ser implementados individualmente por cada resolvedor. Apresenta-se a seguir o seu código.

```
1   package util.math.calculoNumerico;
2
3   /**
4    * ODESolver do pacote util.math.calculoNumerico do projeto tcc.<br>
5    * Classe que representa todos os tipos de resolvedores de funções diferenciais,
6    * implementando os métodos comum a todos eles e definindo a os necessários que
7    * cada um implemente.
8    * @author Rafael Arantes
9    */
10  public abstract class ODESolver
11  {
12
13      private ODE ode;
14
15      /**
16       * Constroi um novo ODESolver
```

```
17          * @param ode equação diferencial ordinaria a ser resolvida
18          */
19         public ODESolver(ODE ode)
20         {
21             super();
22             this.ode = ode;
23         }
24
25         /**
26          * @return a equação diferencial ordinaria a ser resolvida
27          */
28         public ODE getODE()
29         {
30             return ode;
31         }
32
33         /**
34          * @return o valor aproximado de f(x) para o proximo passo, onde a equação
35          *         diferencial ordinaria é f'(x)*dx
36          */
37         public abstract double evaluateODEStep();
38     }
```

As classes cujos códigos são apresentados a seguir são tipos especiais de `ODESolver`, no caso, elas especificam as caracteristicas particulares de cada resolvedor de equação diferencial ordinária. São elas `ModifiedMidpoint`, `PrevisorCorretor`, `RungeKutta1`, sua subclasse idêntica `Euler`, `RungeKutta2` e sua subclasse `Heun`, `RungeKutta3`, `RungeKutta4` e `RungeKutta5`.

O resolvedor `ModifiedMidpoint` dá passos duas vezes maiores que o tamanho definido, mas utiliza como ponto de partida para o cálculo de y_{n+1} o valor de y_{n-1}, ou seja, $y_{n+1} = y_{n-1} + (x_{n+1} - x_{n-1})y'(x_n, y_n)$. Sua implementação é mostrada a seguir.

```
1   package util.math.calculoNumerico;
2
3   /**
4    * ModifiedMidpoint do pacote util.math.calculoNumerico do projeto tcc.<br>
5    * Classe que resolve equações diferenciais ordinarias pelo metodo do ponto
6    * médio modificado com erro de ordem 2.
7    * @author Rafael Arantes
8    */
9   public class ModifiedMidpoint extends ODESolver
10  {
11      private Double yAnterior;
12
```

```
13      /**
14       * Constroi um novo ModifiedMidpoint
15       * @param ode equação diferencial ordinaria a ser resolvida.
16       */
17      public ModifiedMidpoint(ODE ode)
18      {
19          super(ode);
20          yAnterior = new Double("NaN");
21      }
22
23      /**
24       * @return proximo valor de y calculado pelo metodo do ponto médio
25       *         modificado.
26       * @see util.math.calculoNumerico.ODESolver#evaluateODEStep()
27       */
28      @Override
29      public double evaluateODEStep()
30      {
31          ODE ode = getODE();
32          double y = ode.getY();
33          double x = ode.getX();
34          double h = ode.getDeltaT();
35          double yn;
36          if (yAnterior.isNaN())
37              yn = y + h * ode.fX(x, y); // First step.
38          else
39              yn = yAnterior + 2 * h * ode.fX(x, y);
40          yAnterior = y;
41          ode.atualiza(yn);
42          return yn;
43      }
44
45  }
```

O resolvedor `PrevisorCorretor` implementa um método de 4ªordem que utiliza o método Adams-Bashforth para calcular uma primeira aproximação para o valor no próximo ponto, e então corrige esse valor executando o método de Adams-Moulton até que o erro (diferença entre duas execuções seguidas do método) seja menor que um certo valor de tolerância (BURDEN; FAIRES, 2003, Algoritmo 5.4). A seguir apresenta-se seu código.

```
1   package util.math.calculoNumerico;
2
3   import java.util.Vector;
4
5   /**
6    * PrevisorCorretor do pacote util.math.calculoNumerico do projeto tcc.<br>
7    * Implementa um resolvedor de dequações diferenciais por método
```

```
 8    * Previsor-Corretor
 9    * @author Rafael Arantes
10    */
11   public class PrevisorCorretor extends ODESolver
12   {
13       private final int ITLIMITE;
14       private Vector<Double> anteriores = new Vector<Double>();
15       private double tolerancia;
16
17       /**
18        * @param ode equação diferencial ordinaria a ser resolvida
19        * @param itlimite quantidade limite de iterações
20        * @param tolerancia tolerancia da estimativa de erro
21        */
22       public PrevisorCorretor(ODE ode, final int itlimite, double tolerancia)
23       {
24           super(ode);
25           ITLIMITE = itlimite;
26           this.tolerancia = tolerancia;
27       }
28
29       /**
30        * @param n quantos passos atras deve retornar
31        * @return o valor da derivada calculado a n passos atras
32        */
33       private double getAnterior(int n)
34       {
35           int i = anteriores.size() - n;
36           if (anteriores.isEmpty())
37               return getODE().fX(getODE().getX(), getODE().getY());
38           if (i < 0)
39               return anteriores.get(0);
40           return anteriores.get(i);
41       }
42
43       /**
44        * @return proximo valor de y calculado pelo metodo do previsor-corretor
45        * @see util.math.calculoNumerico.ODESolver#evaluateODEStep()
46        */
47       @Override
48       public double evaluateODEStep()
49       {
50           ODE ode = getODE();
51           double yn = ode.getY();
52           double xn = ode.getX();
53           double h = ode.getDeltaT();
54           double fn = ode.fX(xn, yn);
55           double fn_1 = getAnterior(1);
56           double fn_2 = getAnterior(2);
57           double fn_3 = getAnterior(3);
58           double yn1 = yn + h * (55 * fn - 59 * fn_1 + 37 * fn_2 - 9 * fn_3) / 24;
```

```
59          double apYn1;
60          int it = 0;
61          do
62          {
63              double fn1 = ode.fX(xn + h, yn1);
64              apYn1 = yn1;
65              yn1 = yn + h * (9 * fn1 + 19 * fn - 5 * fn_1 + fn_2) / 24;
66              it++;
67          } while ((tolerancia < Math.abs(yn1 - apYn1)) && (it < ITLIMITE));
68          ode.atualiza(yn1);
69          return yn1;
70      }
71
72  }
```

Os resolvedores a seguir são da familia de métodos de Runge-Kutta, que recebem esse nome em referência a seus principais contribuidores (RUNGE, 1895) e (KUTTA, 1901). Eles podem ser descritos pela tabela

c_1	$a_{1,1}$	$a_{1,2}$	$\cdots$	$a_{1,s}$
c_2	$a_{2,1}$	$a_{2,2}$	$\cdots$	$a_{2,s}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
c_s	$a_{s,1}$	$a_{s,2}$	$\cdots$	$a_{s,s}$
	b_1	b_2	$\cdots$	b_s

associada às equações

$$t_n = t_0 + n \cdot h$$

$$v_n = w_{n-1} + h \sum_{j=1}^{s} a_{i,j} \cdot k_{n,j}$$

$$k_{n,j} = f(t_{n-1} + c_j \cdot h, v_{n,j})$$

$$w_n = w_{n-1} + h \sum_{j=1}^{s} b_j \cdot k_{n,j}, w_0 = y(t_0)$$

O resolvedor `RungeKutta1` tem os seguintes valores para esses parametros $s = 1$, $c_1 = 0$, $b_1 = 1$ e $a_{1,1} = 0$, seu código é apresentado a seguir.

```
1  package util.math.calculoNumerico;
2
3  /**
4   * RungeKutta1 do pacote util.math.calculoNumerico do projeto tcc.<br>
5   * Classe que resolve equações diferenciais pelo metodo de Runge Kutta de
6   * primeira ordem.
7   * @author Rafael Arantes
8   */
9  public class RungeKutta1 extends ODESolver
```

```
10  {
11
12      /**
13       * Constroi um novo RungeKutta1
14       * @param ode equação difenrencial a ser resolvida
15       */
16      public RungeKutta1(ODE ode)
17      {
18          super(ode);
19      }
20
21      /**
22       * @return proximo valor de y calculado pelo metodo de Runge Kutta de
23       *         primeira ordem.
24       * @see util.math.calculoNumerico.ODESolver#evaluateODEStep()
25       */
26      @Override
27      public double evaluateODEStep()
28      {
29          ODE ode = getODE();
30          double xn = ode.getX();
31          double yn = ode.getY();
32          double h = ode.getDeltaT();
33          double inc = ode.fX(xn, yn) * h;
34          yn += inc;
35          ode.atualiza(yn);
36          return yn;
37      }
38
39  }
```

O resolvedor `Euler` é identico ao anterior, sendo tratado apenas como um apelido para a classe anterior. Como pode ser visto no código a seguir ele herda todos os métodos do resolvedor `RungeKutta1` sem fazer nenhuma alteração.

```
1   package util.math.calculoNumerico;
2
3   /**
4    * Euler do pacote util.math.calculoNumerico do projeto tcc.<br>
5    * Representa um resolvedor de equações diferenciais ordinarias por método de
6    * Euler.
7    * @author Rafael Arantes
8    */
9   public class Euler extends RungeKutta1
10  {
11      /**
12       * Constroi um novo Euler
13       * @param ode equação diferencial ordinaria a ser resolvida
```

```
14          */
15         public Euler(ODE ode)
16         {
17             super(ode);
18         }
19
20     }
```

O resolvedor `RungeKutta2`, foi baseado em Ruggiero e Lopes (2004) e tem a seguinte tabela

0	0	0
b_1	b_2	0
	a_1	a_2

e o seguinte código.

```
1   package util.math.calculoNumerico;
2
3   /**
4    * RungeKutta2 do pacote util.math.calculoNumerico do projeto tcc.<br>
5    * Classe que resolve equações diferenciais pelo metodo de Runge Kutta de
6    * segunda ordem.
7    * @author Rafael Arantes
8    */
9   public class RungeKutta2 extends ODESolver
10  {
11      private double a1;
12      private double a2;
13      private double b1;
14      private double b2;
15
16      /**
17       * Constroi um novo RungeKutta2
18       * @param ode equação difenrencial a ser resolvida
19       * @param a1 parametro do método chamado de a1 no livro Calculo Numérico:
20       *        Aspectos Teóricos e Computacionais
21       * @param a2 parametro do método chamado de a2 no livro Calculo Numérico:
22       *        Aspectos Teóricos e Computacionais
23       * @param b1 parametro do método chamado de b1 no livro Calculo Numérico:
24       *        Aspectos Teóricos e Computacionais
25       * @param b2 parametro do método chamado de b2 no livro Calculo Numérico:
26       *        Aspectos Teóricos e Computacionais
27       */
28      public RungeKutta2(ODE ode, double a1, double a2, double b1, double b2)
29      {
30          super(ode);
31          this.a1 = a1;
32          this.a2 = a2;
33          this.b1 = b1;
34          this.b2 = b2;
```

```
35      }
36
37      /**
38       * @return proximo valor de y calculado pelo metodo de Runge Kutta de
39       *         segunda ordem com os parametros dados no seu construtor.
40       * @see util.math.calculoNumerico.ODESolver#evaluateODEStep()
41       */
42      @Override
43      public double evaluateODEStep()
44      {
45          ODE ode = getODE();
46          double xn = ode.getX();
47          double yn = ode.getY();
48          double h = ode.getDeltaT();
49          double derivada = ode.fX(xn, yn);
50          yn += h * a1 * derivada + h * a2
51                  * ode.fX(xn + b1 * h, yn + b2 * h * derivada);
52          ode.atualiza(yn);
53          return yn;
54      }
55  }
```

O resolvedor `Heun` é um exemplo de resolvedor de `RungeKutta2`, sua tabela é

0	0	0
1	1	0
	1/2	1/2

e seu código é.

```
1   package util.math.calculoNumerico;
2
3   /**
4    * Heun do pacote util.math.calculoNumerico do projeto tcc.<br>
5    * Representa um resolvedor de equações diferenciais ordinarias por método de
6    * Heun, também conhecido como método de Euler aperfeiçoado.
7    * @author Rafael Arantes
8    */
9   public class Heun extends RungeKutta2
10  {
11      /**
12       * Constroi um novo Heun
13       * @param ode equação diferencial ordinaria a ser resolvida
14       */
15      public Heun(ODE ode)
16      {
17          super(ode, 0.5, 0.5, 1, 1);
18      }
19
20  }
```

O resolvedor `RungeKutta3`, foi baseado em Ruggiero e Lopes (2004) e tem a seguinte tabela

0	0	0	0
1/2	1/2	0	0
3/4	0	3/4	0
	2/9	1/3	4/9

e o seguinte código.

```
1   package util.math.calculoNumerico;
2
3   /**
4    * RungeKutta3 do pacote util.math.calculoNumerico do projeto tcc.<br>
5    * Classe que resolve equações diferenciais pelo metodo de Runge Kutta de
6    * terceira ordem.
7    * @author Rafael Arantes
8    */
9   public class RungeKutta3 extends ODESolver
10  {
11
12      /**
13       * Constroi um novo RungeKutta3
14       * @param ode equação difenrencial a ser resolvida
15       */
16      public RungeKutta3(ODE ode)
17      {
18          super(ode);
19      }
20
21      /**
22       * @return proximo valor de y calculado pelo metodo de Runge Kutta de
23       *         terceira ordem.
24       * @see util.math.calculoNumerico.ODESolver#evaluateODEStep()
25       */
26      @Override
27      public double evaluateODEStep()
28      {
29          ODE ode = getODE();
30          double xn = ode.getX();
31          double yn = ode.getY();
32          double h = ode.getDeltaT();
33          double k1 = h * ode.fX(xn, yn);
34          double k2 = h * ode.fX(xn + (h / 2), yn + k1 / 2);
35          double k3 = h * ode.fX(xn + h * 0.75, yn + k2 * 0.75);
36          yn += k1 * 2 / 9 + k2 / 3 + k3 * 4 / 9;
37          ode.atualiza(yn);
38          return yn;
39      }
40
```

```
41  }
```

O resolvedor `RungeKutta4`, foi baseado em Ruggiero e Lopes (2004) e é o mais frequentemente utilizado método de Runge-Kutta, ele tem a seguinte tabela

0	0	0	0	0
1/2	1/2	0	0	0
1/2	0	1/2	0	0
1	0	0	1	0
	1/6	2/6	2/6	1/6

e o seguinte código.

```
1   package util.math.calculoNumerico;
2
3   /**
4    * RungeKutta4 do pacote util.math.calculoNumerico do projeto tcc.<br>
5    * Classe que resolve equações diferenciais pelo metodo de Runge Kutta de quarta
6    * ordem.
7    * @author Rafael Arantes
8    */
9   public class RungeKutta4 extends ODESolver
10  {
11
12      /**
13       * Constroi um novo RungeKutta4
14       * @param ode equação difenrencial a ser resolvida
15       */
16      public RungeKutta4(ODE ode)
17      {
18          super(ode);
19      }
20
21      /**
22       * @return proximo valor de y calculado pelo metodo de Runge-Kutta de quarta
23       *         ordem
24       * @see util.math.calculoNumerico.ODESolver#evaluateODEStep()
25       */
26      @Override
27      public double evaluateODEStep()
28      {
29          ODE ode = getODE();
30          double yn = ode.getY();
31          double xn = ode.getX();
32          double h = ode.getDeltaT();
33          double k1 = h * ode.fX(xn, yn);
34          double k2 = h * ode.fX(xn + (h / 2), yn + k1 / 2);
35          double k3 = h * ode.fX(xn + (h / 2), yn + k2 / 2);
```

```
36            double k4 = h * ode.fX(xn + h, yn + k3);
37            yn += (k1 + 2 * k2 + 2 * k3 + k4) / 6;
38            ode.atualiza(yn);
39            return yn;
40        }
41    }
```

O resolvedor `RungeKutta5` implementa um método de Runge-Kutta de 5ªordem no estilo previsor-corretor encontrado em (PRESS et al., 1992, seção 16.2). Os métodos previsor e corretor tem mesmo vetor c e matriz a, mudando apenas seus vetores b, a tabela a seguir apresenta esses valores, sendo sua ultima linha o corretor e penultima previsor

0	0	0	0	0	0	0
1/5	$\frac{1}{5}$	0	0	0	0	0
3/10	$\frac{3}{40}$	$\frac{9}{40}$	0	0	0	0
3/5	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$	0	0	0
1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$	0	0
7/8	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	0
	$\frac{2825}{27648}$	0	$\frac{18575}{48384}$	$\frac{13525}{55296}$	$\frac{277}{14336}$	$\frac{1}{4}$
	$\frac{37}{378}$	0	$\frac{250}{621}$	$\frac{125}{597}$	0	$\frac{512}{1771}$

e o seguinte código.

```
1   package util.math.calculoNumerico;
2
3   /**
4    * RungeKutta5 do pacote util.math.calculoNumerico do projeto tcc.<br>
5    * Classe que resolve equações diferenciais ordinarias pelo metodo de
6    * Runge-Kutta de quinta ordem.
7    * @author Rafael Arantes
8    */
9   public class RungeKutta5 extends ODESolver
10  {
11      private double eps; // valor da precisão desejada
12
13      private double yerr; // utilizado por dois métodos...
14
15      /**
16       * Constroi um novo RungeKutta5
17       * @param ode equação diferencial ordinaria a ser resolvida.
18       * @param eps valor da precisão desejada
19       */
20      public RungeKutta5(ODE ode, double eps)
21      {
22          super(ode);
23          this.eps = eps;
```

```
24      }
25
26      /**
27       * @return proximo valor de y calculado pelo metodo de Runge-Kutta de quinta
28       *         ordem.
29       * @see util.math.calculoNumerico.ODESolver#evaluateODEStep()
30       */
31      @Override
32      public double evaluateODEStep()
33      {
34          ODE ode = getODE();
35          double y = ode.getY(); // valor atual do y (variavel dependente)
36          double x = ode.getX(); // valor atual do x (variavel independente)
37          double dydx = ode.fX(x, y); // valor atual da derivada de y(x)
38          double htry = ode.getDeltaT(); // valor do passo pedido
39          y = rkqs(y, dydx, x, htry, ode);
40          return y;
41      }
42
43      /**
44       * Baseado no Numerical Recipes In C.<br>
45       * Fifth-order Runge-Kutta step with monitoring of local truncation error to
46       * ensure accuracy and adjust stepsize.
47       * @param y valor atual do y (variavel dependente)
48       * @param dydx valor atual da derivada de y(x)
49       * @param x valor atual do x (variavel independente)
50       * @param htry valor do passo pedido
51       * @param ode equação diferencial ordinaria que se deseja resolver
52       */
53      private double rkqs(double y, double dydx, double x, double htry, ODE ode)
54      {
55          double SAFETY = 0.9;
56          double PGROW = -0.2;
57          double PSHRNK = -0.25;
58          double ERRCON = 1.89e-4;// equals (5/SAFETY) elevado a (1/PGROW).
59          double errmax, h, htemp, xnew;
60          double hdid, hnext;
61          double ytemp;
62          h = htry;// Set stepsize to the initial trial value.
63          for (;;)
64          {
65              ytemp = rkck(y, dydx, x, h, ode); // Take a step.
66              errmax = 0.0; // Evaluate accuracy.
67              errmax = Math.max(errmax, Math.abs(yerr));
68              errmax /= eps; // Scale relative to required tolerance.
69              if (errmax <= 1.0)
70                  break; // Step succeeded. Compute size of next
71              // step.
72              htemp = SAFETY * h * Math.pow(errmax, PSHRNK);
73              // Truncation error too large, reduce stepsize.
74              h = (h >= 0.0) ? Math.max(htemp, 0.1 * h) : Math
```

```
75                      .min(htemp, 0.1 * h);
76              // No more than a factor of 10.
77              xnew = x + h;
78              if (xnew == x)
79                  System.err.println("stepsize underflow in rkqs");
80          }
81          if (errmax > ERRCON)
82              hnext = SAFETY * h * Math.pow(errmax, PGROW);
83          else
84              hnext = 5.0 * h; // No more than a factor of 5 increase.
85          hdid = h;
86          x += h;
87          y = ytemp;
88          ode.setDeltaT(hdid);
89          ode.atualiza(y);
90          ode.setDeltaT(hnext);
91          return y;
92      }
93
94      /**
95       * Given values for n variables y[1..n] and their derivatives dydx[1..n]
96       * known at x, use the fth-order Cash-Karp Runge-Kutta method to advance the
97       * solution over an interval h and return the incremented variables as
98       * yout[1..n]. Also return an estimate of the local truncation error in yout
99       * using the embedded fourth-order method. The user supplies the routine
100      * derivs(x,y,dydx), which returns derivatives dydx at x.
101      * @param y valor atual do y (variavel dependente)
102      * @param dydx valor atual da derivada de y(x)
103      * @param x valor atual do x (variavel independente)
104      * @param h passo que será dado
105      * @param yout novo valor de y calculado
106      * @param yerr estimativa de erro do y calculado, usando comparação com um
107      *        RungeKutta de ordem 4.
108      * @param ode equação diferencial ordinaria que se deseja resolver
109      */
110     private double rkck(Double y, Double dydx, double x, double h, ODE ode)
111     {
112         final double a2 = 0.2, a3 = 0.3, a4 = 0.6, a5 = 1.0, a6 = 0.875, b21 = 0.2, b31 =
113         final double dc1 = c1 - 2825.0 / 27648.0, dc3 = c3 - 18575.0 / 48384.0, dc4 = c4
114         double ak2, ak3, ak4, ak5, ak6, ytemp;
115         // First step.
116         ytemp = y + h * b21 * dydx;
117         ak2 = ode.fX(x + a2 * h, ytemp); // Second step.
118         ytemp = y + h * (b31 * dydx + b32 * ak2);
119         ak3 = ode.fX(x + a3 * h, ytemp); // Third step.
120         ytemp = y + h * (b41 * dydx + b42 * ak2 + b43 * ak3);
121         ak4 = ode.fX(x + a4 * h, ytemp); // Fourth step.
122         ytemp = y + h * (b51 * dydx + b52 * ak2 + b53 * ak3 + b54 * ak4);
123         ak5 = ode.fX(x + a5 * h, ytemp); // Fifth step.
124         ytemp = y + h
125                 * (b61 * dydx + b62 * ak2 + b63 * ak3 + b64 * ak4 + b65 * ak5);
```

```
126             ak6 = ode.fX(x + a6 * h, ytemp); // Sixth step.
127             // Accumulate increments with proper weights.
128             yerr = h * (dc1 * dydx + dc3 * ak3 + dc4 * ak4 + dc5 * ak5 + dc6 * ak6); // dc5=0
129             return y + h * (c1 * dydx + c3 * ak3 + c4 * ak4 + c6 * ak6); // intracelular=c5=0
130             // Estimate error as di erence between fourth and fth order methods.
131         }
132     }
```

2.2.2 Pacote `mitral`

O pacote `mitral`, onde foi implementado o modelo da célula mitral, foi subdividido nos pacotes `compartimento`, `canal` e `portao`, contendo ainda a interface `Neuronio` e as classes `DecaiCa`, `NeuronioMT` e `TestMitral`.

A interface `Neuronio` define os métodos que devem ser implementados por qualquer modelo de neurônio. Seu código é apresentado na sequência, dispensando maiores informações, já que os comentarios no código o explicam.

```
1   package mitral;
2
3   import java.awt.event.ActionEvent;
4   import java.awt.event.ActionListener;
5
6   /**
7    * Neuronio do pacote mitral do projeto tcc.<br>
8    * Define o que é necessario para representar um neuronio qualquer
9    * @author Rafael Arantes
10   */
11  public interface Neuronio
12  {
13
14      /**
15       * Adiciona um ActionListener no neuronio.
16       * @param a ActionListener a ser adicionado
17       */
18      public abstract void addListener(ActionListener a);
19
20      /**
21       * Notifica a todos os ActionListener do neuronios a ocorrencia de um
22       * evento.
23       * @param evento o que aconteceu
24       */
25      public abstract void notify(ActionEvent evento);
```

```
26
27      /**
28       * Deve informar a diferença de potencial atual através da membrana, em caso
29       * de varios valores informar o mais próximo do cone de implantação.
30       * @return diferença de potencial atual na membrana
31       */
32      public abstract double getDDP();
33
34      /**
35       * Deve informar o valor da diferença de potencial através da membrana
36       * quando o neuronio estiver em repouso.
37       * @return potencial de repouso do neuronio
38       */
39      public abstract double getVRepouso();
40
41  }
```

A classe `DecaiCa` é responsável pela modelagem da dinâmica da concentração de cálcio nos compartimentos em que esta tenha importancia. A seguir é apresentado o seu código. Note que ela, assim como várias outras classes desta parte do sistema, implementa a interface `EDO`, o que lhe permite ser resolvida por um dos resolvedores implementados. Seu principal método é o `public double fX(double x, double y)` que calcula sua derivada em cada ponto.

```
1   package mitral;
2
3   import mitral.canal.CanalCa;
4   import util.math.calculoNumerico.Euler;
5   import util.math.calculoNumerico.ODE;
6   import util.math.calculoNumerico.ODESolver;
7
8   /**
9    * DecaiCa do pacote mitral do projeto tcc.<br>
10   * Classe que modela a variação da concentração de calcio de um compartimento
11   * @author Rafael Arantes
12   */
13  public class DecaiCa implements ODE
14  {
15      /**
16       * DecaiCa que representa canalCa
17       */
18      private CanalCa canalCa;
19
20      /**
21       * int que representa a constante de decaimento
22       */
23      private final int tau = 10;
24
```

```
25      /**
26       * double que representa o valor da concentração no infinito (o minimo)
27       */
28      private final double inf = 1e-5;
29      /**
30       * ODESolver utilizado para resolver a equação diferencial que representa o
31       * decaimento da concentração de calsio.
32       */
33      private ODESolver resolvedor;
34      /**
35       * double que representa a concentração atual de calcio
36       */
37      private double concCa;
38
39      /**
40       * Cria um novo DecaiCa
41       * @param canalCa canal de Calcio presente no compartimento
42       */
43      public DecaiCa(CanalCa canalCa)
44      {
45          this.canalCa = canalCa;
46          concCa = inf;// concentração inicial de calcio no compartimento
47          resolvedor = new Euler(this);
48      }
49
50      /**
51       * Da um passo na simulação
52       */
53      public void proxPasso()
54      {
55          resolvedor.evaluateODEStep();
56      }
57
58      /**
59       * atualiza o valor da concentração
60       * @param y novo valor
61       * @see util.math.calculoNumerico.ODE#atualiza(double)
62       */
63      public void atualiza(double y)
64      {
65          concCa = y;
66      }
67
68      /**
69       * Derivada da equação que relaciona a concentração de calcio no
70       * compartimento com o tempo da simulação.<br>
71       * dy/dx = corrente que entra menos decaimento por dispersão, onde:
72       * <ul>
73       * <li>o que entra é getCorrente()*1e4/(2*96154)</li>
74       * <li>e o que decai é (concCa-inf)/tau</li>
75       * </ul>onde foi considerado inf = 1e-5 e tau = 10
```

```
76          * @param x tempo (variavel independente da simulação)
77          * @param y concentração de calsio (variavel dependente)
78          * @return valor da derivada no ponto (x,y)
79          * @see util.math.calculoNumerico.ODE#fX(double, double)
80          */
81         public double fX(double x, double y)
82         {
83             double fluxo = canalCa.getCorrente() * 1e4 / (2 * 96154);
84             fluxo = (fluxo < 0) ? 0 : fluxo;
85             return fluxo - (concCa - inf) / tau;
86         }
87
88         /**
89          * @return tamanho do passo de tempo da simulação em milissegundos
90          * @see util.math.calculoNumerico.ODE#getDeltaT()
91          */
92         public double getDeltaT()
93         {
94             return canalCa.getDeltaT();
95         }
96
97         /**
98          * @return tempo atual da simulação em milissegundos
99          * @see util.math.calculoNumerico.ODE#getX()
100         */
101        public double getX()
102        {
103            return canalCa.getTempo();
104        }
105
106        /**
107         * @return concentração atual de calcio
108         * @see util.math.calculoNumerico.ODE#getY()
109         */
110        public double getY()
111        {
112            return concCa;
113        }
114
115        /**
116         * Deveria alterar o tamanho do passo de tempo, mas não esta altorizado a
117         * fazer nada.
118         * @param dt seria o novo tamanho do passo de tempo em milissegundos
119         * @see util.math.calculoNumerico.ODE#setDeltaT(double)
120         */
121        public void setDeltaT(double dt)
122        {
123        }
124    }
```

A classe `NeuronioMT` reúne os compartimentos implementados para formar um único neurônio,

ela é responsável pela definição das condutâncias entre compartimentos vizinhos e de parametros gerais, como capacitância e potencial de repouso, além de métodos para a integração dos componentes do modelo em um único neurônio. Seu código é apresentado a seguir.

```
1  package mitral;
2
3  import java.awt.event.ActionEvent;
4  import java.awt.event.ActionListener;
5  import java.util.Vector;
6
7  import mitral.compartimento.Compartimento;
8  import mitral.compartimento.DendritoDistal;
9  import mitral.compartimento.DendritoPrimario;
10 import mitral.compartimento.Glomerular;
11 import mitral.compartimento.Soma;
12 import util.FimSimulacao;
13
14 /**
15  * NeuronioMT do pacote mitral do projeto tcc.<br>
16  * Reconstrução em java do modelo de Andrew P Davison de neuronio mitral
17  * @author Rafael Arantes
18  */
19 public class NeuronioMT implements Neuronio
20 {
21     /**
22      * Glomerular que representa o compartimento glomerular deste neuronio.
23      */
24     private Glomerular glomerular;
25     /**
26      * DendritoPrimario que representa o dendrito primario deste neuronio.
27      */
28     private DendritoPrimario dendritoPrimario;
29     /**
30      * DendritoDistal que representa o dendrito distal deste neuronio.
31      */
32     private DendritoDistal dendritoDistal;
33     /**
34      * Soma que representa o soma deste neuronio.
35      */
36     private Soma soma;
37     /**
38      * double que representa o tamanho do passo de tempo, em milissegundos, da
39      * simulação deste neuronio.
40      */
41     private double dt;
42     /**
43      * double que representa o tempo atual, em milissegundos, na simulação deste
44      * neuronio
45      */
46     private double tempo;
```

```
47      /**
48       * double que representa o limite final do tempo, em milissegundos, da
49       * simução.
50       */
51      private double fim;
52      /**
53       * condutancia da area de contato entre os compartimentos glomerular e de
54       * dendrito primario
55       */
56      private final double gpg = 0.0586;
57      /**
58       * condutancia da area de contato entre os compartimentos soma e dendrito
59       * primario
60       */
61      private final double gsp = 0.0547;
62      /**
63       * condutancia da area de contato entre os compartimentos soma e de dendrito
64       * distal
65       */
66      private final double gsd = 0.194;
67      /**
68       * double que representa o potencia de repouso da membrana deste
69       * compartimento.
70       */
71      protected final double vRepouso = -65;
72      /**
73       * Vector<ActionListener> que guarda uma lista dos objetos que devem ser
74       * notificados da ocorrencia de eventos nesse objeto.
75       */
76      protected Vector<ActionListener> listeners;
77      private double corrente;
78      private double inicioCorrente;
79      private Compartimento localCorrente;
80
81      /**
82       * Cria um novo NeuronioMT
83       */
84      public NeuronioMT()
85      {
86          // definições de tempo: inicial, passo e final
87          tempo = 0;
88          dt = 1e-3;
89          fim = 100;
90          // capacitancia comum a todos os compartimentos, 1 microFarad por
91          // centimetro quadrado.
92          double capacitancia = 1;
93          // criação dos compartimentos
94          glomerular = new Glomerular(capacitancia, this);
95          dendritoPrimario = new DendritoPrimario(capacitancia, this);
96          soma = new Soma(capacitancia, this);
97          dendritoDistal = new DendritoDistal(capacitancia, this);
```

```
98          // deixa o Vector pronto
99          listeners = new Vector<ActionListener>();
100     }
101
102     /**
103      * Constroi um novo NeuronioMT
104      * @param dt tamanho do passo de tempo
105      * @param fim limite final do tempo da simulção
106      * @param corrente corrente injetada
107      * @param localCorrente onde a corrente sera injetada (1- glomerulo, 2-
108      *        soma)
109      * @param inicioCorrente tempo em que a corrente começa a ser injetada
110      */
111     public NeuronioMT(double dt, double fim, double corrente,
112             int localCorrente, double inicioCorrente)
113     {
114         // definições de tempo: inicial, passo e final
115         tempo = 0;
116         this.dt = dt;
117         this.fim = fim;
118         // capacitancia comum a todos os compartimentos, 1 microFarad por
119         // centimetro quadrado.
120         double capacitancia = 1;
121         // criação dos compartimentos
122         glomerular = new Glomerular(capacitancia, this);
123         dendritoPrimario = new DendritoPrimario(capacitancia, this);
124         soma = new Soma(capacitancia, this);
125         dendritoDistal = new DendritoDistal(capacitancia, this);
126         // deixa o Vector pronto
127         listeners = new Vector<ActionListener>();
128         this.corrente = corrente;
129         this.localCorrente = (localCorrente == 1) ? glomerular : soma;
130         this.inicioCorrente = inicioCorrente;
131     }
132
133     /**
134      * Da um passo na simulação calculando e atualizando os valores.
135      */
136     public void proxPasso()
137     {
138         soma.proxPasso();
139         glomerular.proxPasso();
140         dendritoPrimario.proxPasso();
141         dendritoDistal.proxPasso();
142         tempo += dt;
143         if ((inicioCorrente - dt <= tempo) && (tempo <= inicioCorrente))
144             localCorrente.setCorrenteInjetada(corrente);
145         if (tempo >= fim)
146             notify(new FimSimulacao(this, 0, "terminei"));
147     }
148
```

```
149         /**
150          * Adiciona um objeto na lista dos que devem ser informados quando ocorrer
151          * um evento em algum compartimento
152          * @param a objeto a ser adicionado
153          * @see Glomerular#addListener(ActionListener)
154          * @see DendritoPrimario#addListener(ActionListener)
155          * @see Soma#addListener(ActionListener)
156          * @see DendritoDistal#addListener(ActionListener)
157          */
158         public void addListener(ActionListener a)
159         {
160             glomerular.addListener(a);
161             dendritoPrimario.addListener(a);
162             soma.addListener(a);
163             dendritoDistal.addListener(a);
164             listeners.add(a);
165         }
166 
167         /**
168          * @return tamanho do passo de tempo da simulação em milissegundos
169          */
170         public double getDt()
171         {
172             return dt;
173         }
174 
175         /**
176          * @return tempo atual da simulação em milissegundos
177          */
178         public double getTempo()
179         {
180             return tempo;
181         }
182 
183         /**
184          * Altera o tamanho do passo de tempo da simulação
185          * @param dt novo tamanho em milissegundos
186          */
187         public void setDt(double dt)
188         {
189             this.dt = dt;
190         }
191 
192         /**
193          * @return o compartimento glomerular deste neuronio.
194          */
195         public Compartimento getCompartimentoGlomerular()
196         {
197             return glomerular;
198         }
199 
```

```
200         /**
201          * @return o dendrito primario deste neuronio.
202          */
203         public DendritoPrimario getDendritoPrimario()
204         {
205             return dendritoPrimario;
206         }
207
208         /**
209          * @return o dendrito distal deste neuronio.
210          */
211         public DendritoDistal getDendritoDistal()
212         {
213             return dendritoDistal;
214         }
215
216         /**
217          * @return o soma deste neuronio.
218          */
219         public Soma getSoma()
220         {
221             return soma;
222         }
223
224         /**
225          * @return a condutancia da area de contato entre o soma e o dendrito
226          *         primario
227          */
228         public double getGsp()
229         {
230             return gsp;
231         }
232
233         /**
234          * @return a condutancia da area de contato entre o soma e o dendrito distal
235          */
236         public double getGsd()
237         {
238             return gsd;
239         }
240
241         /**
242          * @return a condutancia da area de contato entre os compartimentos
243          *         glomerular e de dendrito primario
244          */
245         public double getGpg()
246         {
247             return gpg;
248         }
249
250         /**
```

```
251         * @return valor do potencial de repouso do neuronio
252         * @see Neuronio#getVRepouso()
253         */
254        public double getVRepouso()
255        {
256            return vRepouso;
257        }
258
259        /**
260         * Informa a diferença de potencial atual através da membrana no soma do
261         * neuronio.
262         * @return diferença de potencial atual no soma
263         * @see Neuronio#getDDP()
264         */
265        public double getDDP()
266        {
267            return soma.getDDP();
268        }
269
270        /**
271         * Notifica a ocorrencia de um evento a quem deve ser notificado
272         * @param evento o que aconteceu
273         * @see #addListener(java.awt.event.ActionListener)
274         * @see Neuronio#notify(java.awt.event.ActionEvent)
275         */
276        public void notify(ActionEvent evento)
277        {
278            for (ActionListener listener : listeners)
279                listener.actionPerformed(evento);
280        }
281    }
```

A classe `TestMitral` executa a simulação do modelo para varias situações de interesse. Seu código é o seguinte.

```
1   package mitral;
2
3   import java.awt.event.ActionEvent;
4   import java.awt.event.ActionListener;
5   import java.util.Vector;
6
7   import util.FimSimulacao;
8   import util.Grafico;
9   import util.Mapeavel;
10
11  /**
12   * TestMitral do pacote mitral do projeto tcc.<br>
13   * Classe criada para testar o NeuronioMT
14   * @see NeuronioMT
```

```
15   * @author Rafael Arantes
16   */
17  public class TestMitral extends Thread implements ActionListener
18  {
19      private Grafico grafico;
20      private Vector<String> strCampos;
21      private NeuronioMT n;
22      private final double espaco = 620;
23      private int maxPassosAtePlotar;
24      private int passosAtePlotar = 0;
25      private boolean plota = true;
26      private boolean escreve = false;
27      private Vector<Object[]> espera;
28
29      /**
30       * Constroi um novo TestMitral
31       */
32      public TestMitral()
33      {
34          espera = new Vector<Object[]>();
35          espera.add(new Object[] { 1e-3, 400.0, 0.2, 1, 10.0,
36                  "0.2 uA no Glomérulo" });
37          espera.add(new Object[] { 1e-3, 210.0, 1.6, 1, 50.0,
38                  "1.6 uA no Glomérulo" });
39          espera
40                  .add(new Object[] { 1e-3, 200.0, 0.4, 2, 50.0, "0.4 uA no Soma" });
41          espera
42                  .add(new Object[] { 1e-3, 150.0, 0.8, 2, 50.0, "0.8 uA no Soma" });
43          espera
44                  .add(new Object[] { 1e-3, 450.0, 0.2, 2, 10.0, "0.2 uA no Soma" });
45          espera
46                  .add(new Object[] { 1e-3, 100.0, 1.6, 2, 50.0, "1.6 uA no Soma" });
47          simulaProx();
48      }
49
50      /**
51       * Passa para a proxima simulação agendada
52       */
53      private void simulaProx()
54      {
55          if (espera.isEmpty())
56              System.exit(0);
57          strCampos = new Vector<String>();
58          Object[] param = espera.remove(0);
59          double dt = (Double) param[0];
60          double fim = (Double) param[1];
61          maxPassosAtePlotar = (int) (fim / dt / espaco);
62          n = new NeuronioMT(dt, fim, (Double) param[2], (Integer) param[3],
63                  (Double) param[4]);
64          n.addListener(this);
65          criaGrafico();
```

```
66          if (grafico == null)
67              System.out.println("Grafico nulo!!");
68          grafico.setTitle((String) param[5]);
69      }
70
71      /**
72       * Adiciona um mapeavel que deve ser ouvido por este ActionListener
73       * @param l mapeavel que deve ser ouvido
74       * @return valor da id que o evento gerado pelo mapeavel deve ter
75       */
76      public int addMapeado(Mapeavel l)
77      {
78          int retorno = strCampos.size();
79          strCampos.add(l.getNome());
80          return retorno;
81      }
82
83      /**
84       * Cria o grafico que apresentarï¿½ as alteraï¿½ï¿½es de valores do que for
85       * mapeado
86       */
87      private void criaGrafico()
88      {
89          if (strCampos.size() == 0)
90              return;
91          if (strCampos.size() == 1)
92          {
93              grafico = new Grafico("tempo em milissegundos", "valor de "
94                      + strCampos.get(0));
95              grafico.ocultaLegenda();
96              return;
97          }
98          grafico = new Grafico("tempo em milissegundos", "ddp da membrana em mV");
99          grafico.destravar();
100         for (String strCampo : strCampos)
101         {
102             grafico.createSerie(strCampo);
103             grafico.travar();
104         }
105     }
106
107     /**
108      * Método envocado quando ocorre um evento em um objeto que esta sendo
109      * mapeado. Adiciona o ponto do novo valor no grafico
110      * @param evento o que ocorreu
111      * @see java.awt.event.ActionListener#actionPerformed(java.awt.event.ActionEvent)
112      */
113     public void actionPerformed(ActionEvent evento)
114     {
115         if (evento.getSource() instanceof Mapeavel)
116         {
```

```
117                 Mapeavel fonte = (Mapeavel) evento.getSource();
118                 if (plota)
119                     if (passosAtePlotar == 0)
120                     {
121                         grafico.setSerie(strCampos.get(evento.getID()));
122                         grafico.addPonto(fonte.getX(), fonte.getY());
123                         if (evento.getID() == strCampos.size() - 1)
124                             passosAtePlotar = maxPassosAtePlotar;
125                     }
126                     else
127                         if (evento.getID() == strCampos.size() - 1)
128                             passosAtePlotar--;
129                 if (escreve)
130                     System.out.println(fonte.getNome() + "(" + fonte.getX() + ", "
131                             + fonte.getY() + ")");
132             }
133             if (evento instanceof FimSimulacao)
134             {
135                 grafico.salvar();
136                 simulaProx();
137             }
138         }
139 
140         /**
141          * Executa a simulação.
142          * @see java.lang.Thread#run()
143          */
144         @Override
145         public void run()
146         {
147             while (Thread.currentThread() == this)
148                 n.proxPasso();
149         }
150 
151         /**
152          * Cria um teste e executa
153          * @param args
154          */
155         public static void main(String[] args)
156         {
157             new TestMitral().start();
158         }
159     }
```

2.2.2.1 Pacote `compartimento`

O pacote `compartimento` é composto pelas classes `Compartimento`, `Glomerular`, `DedritoPrimario`, `DendritoDistal` e `Soma`.

A classe `Compartimento` define atributos e métodos comuns a todo compartimento implementado. Seu código é apresentado a seguir

```
1  package mitral.compartimento;
2
3  import java.awt.event.ActionEvent;
4  import java.awt.event.ActionListener;
5  import java.util.Vector;
6
7  import mitral.NeuronioMT;
8  import mitral.TestMitral;
9  import mitral.canal.Canal;
10 import util.Mapeavel;
11 import util.math.calculoNumerico.Euler;
12 import util.math.calculoNumerico.ODE;
13 import util.math.calculoNumerico.ODESolver;
14
15 /**
16  * Compartimento do pacote mitral.compartimento do projeto tcc.<br>
17  * Representa um compartimento genérico, implementando os métodos que são
18  * necessarios a todo compartimento.
19  * @author Rafael Arantes
20  */
21 public abstract class Compartimento implements ODE, Mapeavel
22 {
23     /**
24      * Vector<Canal> que representa os canais ionicos deste compartimento.
25      */
26     protected Vector<Canal> canais;
27     /**
28      * Neuronio ao qual pertence este compartimento.
29      */
30     protected NeuronioMT neuronio;
31     /**
32      * double que representa potencial de membrana atual em milivolts.
33      */
34     protected double ddp;
35     /**
36      * double que representa capacitancia da membrana do compartimento em
37      * microFarad por centimetro quadrado.
38      */
39     protected double capacitancia;
40     /**
41      * double que representa a corrente ionioca que atravessa a membrana em
42      * microAmpere por centimetro quadrado.
43      */
44     protected double corrente;
```

```
45      /**
46       * ODESolver utilizado para resolver a equação diferencial que representa o
47       * funcionamento deste compartimento.
48       */
49      protected ODESolver resolvedor;
50      /**
51       * Vector<ActionListener> que guarda uma lista dos objetos que devem ser
52       * notificados da ocorrencia de eventos nesse objeto.
53       */
54      protected Vector<ActionListener> listeners;
55      /**
56       * double que representa a concentração de Calsio na célula.
57       */
58      private double concCa;
59      /**
60       * double que representa a corrente injetada no compartimento em microAmpere
61       * por centimetro quadrado.
62       */
63      protected double correnteInjetada;
64      private int id;
65
66      /**
67       * Constroi um novo compartimento genérico
68       * @param capacitancia valor da capacitancia da membrana nesse compartimento
69       * @param neuronio neuronio ao qual pertence este compartimento
70       */
71      public Compartimento(double capacitancia, NeuronioMT neuronio)
72      {
73          this.capacitancia = capacitancia;
74          this.neuronio = neuronio;
75          resolvedor = new Euler(this);// a equação diferencial representada
76          // nesta classe sera relovida por método
77          // de Euler.
78          ddp = neuronio.getVRepouso();// o potencial de membrana inicial será
79          // igual o seu potenical de repouso.
80          concCa = 1e-5;// concentração inicial de calsio no compartimento, pego
81          // pelo seu limite inferior em Bhalla e Bower
82          listeners = new Vector<ActionListener>();// deixa o Vector pronto
83          // para inserções.
84          canais = getCanais();
85          corrente = calcCorrenteIonica();// valor inicial da corrente ionica na
86          // membrana.
87          correnteInjetada = 0.0;// valor inicial da corrente injetada no
88          // compartimento.
89      }
90
91      /**
92       * Deve ser implementada por cada compartimento para criar os canais
93       * especificos daquele compartimento
94       * @return os canais presentes no compartimento
95       */
```

```
96      public abstract Vector<Canal> getCanais();
97
98      /**
99       * @return valor da corrente ionica calculada.
100      */
101     public double calcCorrenteIonica()
102     {
103         double retorno = 0;
104         for (Canal canal : getCanais())
105             retorno -= canal.getCorrente();
106         corrente = retorno;
107         return retorno;
108     }
109
110     /**
111      * Da um passo na simulação (calcula os novos valores e notifica)
112      */
113     public void proxPasso()
114     {
115         for (Canal canal : getCanais())
116             canal.proxPasso();
117         calcCorrenteIonica();
118         resolvedor.evaluateODEStep();
119         notify(new ActionEvent(this, getId(), ""));
120     }
121
122     /**
123      * @return numero de identificação do compartimento
124      */
125     public int getId()
126     {
127         return id;
128     }
129
130     /**
131      * @return o valor da corrente injetada no compartimento
132      */
133     public double getCorrenteInjetada()
134     {
135         return correnteInjetada;
136     }
137
138     /**
139      * Altera o valor de corrente injetada no compartimento
140      * @param correnteInjetada nova corrente injetada
141      */
142     public void setCorrenteInjetada(double correnteInjetada)
143     {
144         this.correnteInjetada = correnteInjetada;
145     }
146
```

```
147         /**
148          * Adiciona um objeto na lista dos que devem ser informados quando ocorrer
149          * um evento
150          * @param a objeto a ser adicionado
151          * @see #notify(java.awt.event.ActionEvent)
152          */
153         public void addListener(ActionListener a)
154         {
155             id = ((TestMitral) a).addMapeado(this);
156             listeners.add(a);
157         }
158
159         /**
160          * Notifica a ocorrencia de um evento a quem deve ser notificado
161          * @param evento o que aconteceu
162          * @see #addListener(java.awt.event.ActionListener)
163          */
164         public void notify(ActionEvent evento)
165         {
166             for (ActionListener listener : listeners)
167                 listener.actionPerformed(evento);
168         }
169
170         /**
171          * Altera o valor do potencial de membrana atual
172          * @param v novo valor do potencial
173          * @see util.math.calculoNumerico.ODE#atualiza(double)
174          */
175         public void atualiza(double v)
176         {
177             ddp = v;
178         }
179
180         /**
181          * Derivada da equação que relaciona a ddp no compartimento com o tempo da
182          * simulação.
183          * @param t tempo no ponto em que a equação é calculada (variavel
184          *        independente)
185          * @param v potencial de membrana no ponto em que a equação é calculada
186          *        (variavel dependente)
187          * @return valor da equação no ponto.
188          * @see util.math.calculoNumerico.ODE#fX(double, double)
189          */
190         public abstract double fX(double t, double v);
191
192         /**
193          * Altera o valor do passo de tempo.
194          * @param dt novo valor do passo de tempo
195          * @see util.math.calculoNumerico.ODE#setDeltaT(double)
196          */
197         public void setDeltaT(double dt)
```

```
198         {
199             neuronio.setDt(dt);
200         }
201 
202         /**
203          * @return potencial de membrana atual.
204          */
205         public double getDDP()
206         {
207             return ddp;
208         }
209 
210         /**
211          * @return valor do potencial de repouso do compartimento
212          */
213         public double getVRepouso()
214         {
215             return neuronio.getVRepouso();
216         }
217 
218         /**
219          * @return valor da corrente ionica que passa pelos canais.
220          */
221         public double getCorrenteIonica()
222         {
223             return corrente;
224         }
225 
226         /**
227          * @return tamanho do passo de tempo
228          * @see util.math.calculoNumerico.ODE#getDeltaT()
229          */
230         public double getDeltaT()
231         {
232             return neuronio.getDt();
233         }
234 
235         /**
236          * @return valor atual de tempo (variavel independente da equação
237          *         diferencial do funcionamento do compartimento)
238          * @see util.math.calculoNumerico.ODE#getX()
239          */
240         public double getX()
241         {
242             return neuronio.getTempo();
243         }
244 
245         /**
246          * @return valor do potencial de membrana atual
247          * @see #getDDP()
248          * @see util.math.calculoNumerico.ODE#getY()
```

```
249         */
250        public double getY()
251        {
252            return getDDP();
253        }
254
255        /**
256         * @return concentração de Calsio no compartimento
257         */
258        public double getConcCa()
259        {
260            return concCa;
261        }
262
263        /**
264         * Altera a concentração de Calsio no compartimento
265         * @param concCa novo valor da concentração
266         */
267        public void setConcCa(double concCa)
268        {
269            this.concCa = concCa;
270        }
271
272        /**
273         * @return o nome do compartimento
274         * @see util.Mapeavel#getNome()
275         */
276        public String getNome()
277        {
278            return getClass().getSimpleName();
279        }
280    }
```

As classes cujos códigos são apresentados a seguir são tipos especiais de `Compartimento`. No caso, elas especificam as caracteristicas particulares de cada compartimento do modelo. Na ordem em que aparecem seus códigos, são elas `Glomerular`, `DedritoPrimario`, `DendritoDistal` e `Soma`.

```
1   package mitral.compartimento;
2
3   import java.util.Vector;
4
5   import mitral.NeuronioMT;
6   import mitral.canal.Canal;
7   import mitral.canal.CanalCa;
8   import mitral.canal.CanalK;
9   import mitral.canal.CanalVazamento;
10  import util.Mapeavel;
```

```
11  import util.math.calculoNumerico.ODE;
12
13  /**
14   * CompartimentoGlomerular do pacote mitral2 do projeto Utilitarios Reconstrução
15   * em java do modelo de Andrew P Davison de compartimento glomerular de célula
16   * mitral
17   * @author Rafael Arantes
18   */
19  public class Glomerular extends Compartimento implements ODE, Mapeavel
20  {
21      /**
22       * double que representa a area deste compartimento dividida pela soma das
23       * areas de todos os compartimentos deste neuronio.
24       */
25      private final double areaRelativa = 0.084;
26      /**
27       * double que representa um fator de ajuste no valor de corrente injetada.
28       */
29      private final double alpha = 1.85;
30
31      /**
32       * Cria um novo CompartimentoGlomerular
33       * @param capacitancia capacitancia da membrana do compartimento em
34       *        microFarad por centimetro quadrado
35       * @param neuronio neuronio ao qual pertence este compartimento.
36       */
37      public Glomerular(double capacitancia, NeuronioMT neuronio)
38      {
39          super(capacitancia, neuronio);
40      }
41
42      /**
43       * Derivada da equação que relaciona a ddp no dendrito glomerular com o
44       * tempo da simulação.<br>
45       * dv/dt = somatoria das correntes / capacitancia
46       * @param t tempo no ponto em que a equação é calculada (variavel
47       *        independente)
48       * @param v potencial de membrana no ponto em que a equação é calculada
49       *        (variavel dependente)
50       * @return valor da equação no ponto.
51       * @see mitral.compartimento.Compartimento#fX(double, double)
52       */
53      @Override
54      public double fX(double t, double v)
55      {
56          // System.out.println("potencial no glomerulo = "+v);
57          double ionica = getCorrenteIonica();// corrente que passa pelos canais
58          double injetada = getCorrenteInjetada() * alpha / areaRelativa;// corrente
59          // injetada
60          // ajustada
61          double compVizinho = neuronio.getGpg()
```

```
62                  * (neuronio.getDendritoPrimario().getDDP() - getDDP())
63                  / areaRelativa;// corrente vinda do dendrito primario ajustada
64          double retorno = (ionica + injetada + compVizinho) / capacitancia;
65          return retorno;
66      }
67
68      /**
69       * Constroi os canais do dendrito glomerular, que são:
70       * <ul>
71       * <li>K com condutancia 20;</li>
72       * <li>Ca com condutancia 9.5; e</li>
73       * <li>de vazamento com potencial de reversão -54.4 e condutancia 0.01</li>
74       * </ul>
75       * @return os canais presentes no dendrito glomerular
76       * @see mitral.compartimento.Compartimento#getCanais()
77       */
78      @Override
79      public Vector<Canal> getCanais()
80      {
81          if (canais == null)
82          {// cria os canais presentes segundo Davison.
83              canais = new Vector<Canal>();
84              // condutancia de Davison - MATERIALS AND METHODS
85              canais.add(new CanalK(this, 20));
86              // condutancia de Bhalla e Bower - TABLE 1
87              canais.add(new CanalCa(this, 9.5));
88              // condutancia de Bhalla e Bower - RESULTS - Mitral cells - PASSIVE
89              // PROPERTIES
90              canais.add(new CanalVazamento(this, -54.4, 0.01));
91          }
92          return canais;
93      }
94  }
```

```
1   package mitral.compartimento;
2
3   import java.util.Vector;
4
5   import mitral.NeuronioMT;
6   import mitral.canal.Canal;
7   import mitral.canal.CanalKFast;
8   import mitral.canal.CanalNa;
9   import mitral.canal.CanalVazamento;
10  import util.Mapeavel;
```

```
11  import util.math.calculoNumerico.ODE;
12
13  /**
14   * CompartimentoDendritoDistal do pacote mitral2 do projeto Utilitarios
15   * Reconstrução em java do modelo de Andrew P Davison de compartimento
16   * representante do dendrito distal de célula mitral
17   * @author Rafael Arantes
18   */
19  public class DendritoDistal extends Compartimento implements ODE, Mapeavel
20  {
21      /**
22       * double que representa a area deste compartimento dividida pela soma das
23       * areas de todos os compartimentos deste neuronio.
24       */
25      private final double areaRelativa = 0.537;
26
27      /**
28       * Cria um novo CompartimentoDendritoDistal
29       * @param capacitancia capacitancia da membrana do compartimento em
30       *        microFarad por centimetro quadrado
31       * @param neuronio neuronio ao qual pertence este compartimento.
32       */
33      public DendritoDistal(double capacitancia, NeuronioMT neuronio)
34      {
35          super(capacitancia, neuronio);
36      }
37
38      /**
39       * Derivada da equação que relaciona a ddp no dendrito distal com o tempo da
40       * simulação.<br>
41       * dv/dt = somatoria das correntes / capacitancia
42       * @param t tempo no ponto em que a equação é calculada (variavel
43       *        independente)
44       * @param v potencial de membrana no ponto em que a equação é calculada
45       *        (variavel dependente)
46       * @return valor da equação no ponto.
47       * @see mitral.compartimento.Compartimento#fX(double, double)
48       */
49      @Override
50      public double fX(double t, double v)
51      {
52          double ionica = getCorrenteIonica();// corrente que passa pelos canais
53          double compVizinho = neuronio.getGsd()
54                  * (neuronio.getSoma().getDDP() - getDDP()) / areaRelativa;// corrente
55          // vinda
56          // do
57          // soma
58          // ajustada
59          double retorno = (ionica + compVizinho) / capacitancia;
60          return retorno;
61      }
```

```
62
63      /**
64       * Constroi os canais do dendrito distal, que são:
65       * <ul>
66       * <li>Kfast com condutancia 12.8;</li>
67       * <li>Na com condutancia 12.2; e</li>
68       * <li>de vazamento com potencial de reversão -54.4 e condutancia 0.01</li>
69       * </ul>
70       * @return os canais presentes no dendrito distal
71       * @see mitral.compartimento.Compartimento#getCanais()
72       */
73      @Override
74      public Vector<Canal> getCanais()
75      {
76          if (canais == null)
77          {// cria os canais presentes segundo Davison.
78              canais = new Vector<Canal>();
79              // condutancias extraidas do artigo de Bhalla e Bower Tabela 1 e
80              // RESULTS - Mitral cells - PASSIVE PROPERTIES
81              canais.add(new CanalKFast(this, 12.8));
82              canais.add(new CanalNa(this, 12.2));
83              canais.add(new CanalVazamento(this, -54.4, 0.01));
84          }
85          return canais;
86      }
87  }
```

```
1   package mitral.compartimento;
2
3   import java.util.Vector;
4
5   import mitral.NeuronioMT;
6   import mitral.canal.Canal;
7   import mitral.canal.CanalCa;
8   import mitral.canal.CanalK;
9   import mitral.canal.CanalKA;
10  import mitral.canal.CanalKCa;
11  import mitral.canal.CanalKFast;
12  import mitral.canal.CanalNa;
13  import mitral.canal.CanalVazamento;
14  import util.Disparo;
15  import util.Mapeavel;
16  import util.math.calculoNumerico.ODE;
17  import util.math.calculoNumerico.RungeKutta4;
18
19  /**
20   * CompartimentoSoma do pacote mitral2 do projeto Utilitarios Reconstrução em
21   * java do modelo de Andrew P Davison de compartimento somatico de célula mitral
22   * @author Rafael Arantes
```

```
23   */
24  public class Soma extends Compartimento implements ODE, Mapeavel
25  {
26      private static final double limiar = 0;
27      /**
28       * double que representa um fator de ajuste no valor de corrente injetada.
29       */
30      private final double alpha = 1;
31      /**
32       * double que representa a area deste compartimento dividida pela soma das
33       * areas de todos os compartimentos deste neuronio.
34       */
35      private final double areaRelativa = 0.051;
36
37      /**
38       * Cria um novo CompartimentoSoma
39       * @param capacitancia capacitancia da membrana do compartimento em
40       *        microFarad por centimetro quadrado
41       * @param neuronio neuronio ao qual pertence este compartimento.
42       */
43      public Soma(double capacitancia, NeuronioMT neuronio)
44      {
45          super(capacitancia, neuronio);
46          resolvedor = new RungeKutta4(this);// a equação diferencial
47          // representada nesta classe sera
48          // relovida por método de Euler.
49          corrente = calcCorrenteIonica();// valor inicial da corrente ionica na
50          // membrana.
51      }
52
53      /**
54       * Derivada da equação que relaciona a ddp no soma com o tempo da simulação.<br>
55       * dv/dt = somatoria das correntes / capacitancia
56       * @param t tempo no ponto em que a equação é calculada (variavel
57       *        independente)
58       * @param v potencial de membrana no ponto em que a equação é calculada
59       *        (variavel dependente)
60       * @return valor da equação no ponto.
61       * @see mitral.compartimento.Compartimento#fX(double, double)
62       */
63      @Override
64      public double fX(double t, double v)
65      {
66          double ionica = getCorrenteIonica();// corrente que passa pelos canais
67          double injetada = getCorrenteInjetada() * alpha / areaRelativa;// corrente
68          // injetada
69          // ajustada
70          double compVizinho = neuronio.getGsd()
71                  * (neuronio.getDendritoDistal().getDDP() - getDDP());// corrente
72          // vinda
73          // do
```

```
74          // dendrito
75          // distal
76          compVizinho += neuronio.getGsp()
77                  * (neuronio.getDendritoPrimario().getDDP() - getDDP());// corrente
78          // vinda
79          // do
80          // dendrito
81          // primario
82          compVizinho /= areaRelativa;// ajuste da corrente vinda de
83          // compartimentos vizinhos
84          double retorno = (ionica + injetada + compVizinho) / capacitancia;
85          return retorno;
86      }
87
88      /**
89       * Constroi os canais do dendrito primario, que são:
90       * <ul>
91       * <li>K com condutancia 2.8;</li>
92       * <li>Ca com condutancia 4;</li>
93       * <li>Kfast com condutancia 195.6;</li>
94       * <li>Na com condutancia 153.2;</li>
95       * <li>KA com condutancia 5.87;</li>
96       * <li>KCa com condutancia 14.2; e</li>
97       * <li>de vazamento com potencial de reversão -65 e condutancia 0.01</li>
98       * </ul>
99       * @return os canais presentes no dendrito primario
100      * @see mitral.compartimento.Compartimento#getCanais()
101      */
102     @Override
103     public Vector<Canal> getCanais()
104     {
105         if (canais == null)
106         {// cria os canais presentes segundo Davison.
107             canais = new Vector<Canal>();
108             // condutancias extraidas do artigo de Bhalla e Bower Tabela 1 e
109             // RESULTS - Mitral cells - PASSIVE PROPERTIES
110             canais.add(new CanalK(this, 2.8));
111             canais.add(new CanalCa(this, 4));
112             canais.add(new CanalKFast(this, 195.6));
113             canais.add(new CanalNa(this, 153.2));
114             canais.add(new CanalKA(this, 5.87));
115             canais.add(new CanalKCa(this, 14.2));
116             canais.add(new CanalVazamento(this, -54.4, 0.01));
117         }
118         return canais;
119     }
120
121     /**
122      * @see mitral.compartimento.Compartimento#proxPasso()
123      */
124     @Override
```

```
125         public void proxPasso()
126         {
127             double ddpAnterior = getDDP();
128             super.proxPasso();
129             if ((ddpAnterior < limiar) && (getDDP() >= limiar))
130                 notify(new Disparo(this, getId(), ""));
131         }
132     }
```

2.2.2.2 Pacote `canal`

O pacote `canal` possui as classes `Canal`, `CanalCa`, `CanalK`, `CanalKA`, `CanalKCa`, `CanalKfast`, `CanalNa` e `CanalVazamento`.

A classe `Canal` define atributos e métodos comuns a todo canal implementado e tem seu código reproduzido a seguir.

```
1   package mitral.canal;
2
3   import java.awt.event.ActionEvent;
4   import java.awt.event.ActionListener;
5   import java.util.Vector;
6
7   import mitral.TestMitral;
8   import mitral.compartimento.Compartimento;
9   import mitral.portao.Portao;
10  import util.Mapeavel;
11
12  /**
13   * Canal do pacote mitral.canal do projeto tcc.<br>
14   * Um canal iônico é uma passagem através da membrana de uma célula por onde
15   * podem passar determinados íons de acordo com seus gradientes eletroquímicos.
16   * Esta classe representa um canal iônico genérico, implementando os métodos que
17   * são necessarios a todo canal.
18   * @author Rafael Arantes
19   */
20  public abstract class Canal implements Mapeavel
21  {
22      /**
23       * condutancia do canal em miliSiemens por centimetro quadrado.
24       */
25      protected Double condutancia;
26      /**
27       * Portao[] que representa os portoes presentes nesse canal. Um canal só
```

```
28          * esta aberto se todos os seus portões estiverem abertos.
29          */
30         protected Portao[] portoes;
31
32         private Compartimento compartimento;
33         /**
34          * corrente do canal em microAmperes
35          */
36         private double corrente;
37
38         /**
39          * Potencial reversão para esse canal em milivolts
40          */
41         protected Double potencialReversao;
42         private Vector<ActionListener> listeners;
43         private int id;
44
45         // Construtores
46         /**
47          * Constroi um novo canal genérico.
48          * @param compartimento compartimento ao qual pertence este canal
49          */
50         public Canal(Compartimento compartimento)
51         {
52             this.compartimento = compartimento;
53             listeners = new Vector<ActionListener>();
54             condutancia = getCondutancia();
55             portoes = getPortoes();
56             potencialReversao = getPotencialReversao();
57             corrente = calcCorrente();
58         }
59
60         /**
61          * Calcula o valor da corrente ionica que passa pelo canal neste instante da
62          * simulação
63          * @return corrente ionica que passa pelo canal
64          */
65         public double calcCorrente()
66         {
67             double retorno = getCondutancia()
68                     * (compartimento.getDDP() - getPotencialReversao());
69             for (Portao portao : getPortoes())
70                 retorno *= portao.getValor();
71             corrente = retorno;
72             return retorno;
73         }
74
75         /**
76          * @return o valor do potencial de reversão do canal em milivolts
77          */
78         public abstract double getPotencialReversao();
```

```
79
80      /**
81       * @return o valor da condutancia maxima do canal em miliSiemens por
82       *         centimetro quadrado.
83       */
84      public abstract double getCondutancia();
85
86      /**
87       * @return compartimento ao qual pertence este canal.
88       */
89      public Compartimento getCompartimento()
90      {
91          return compartimento;
92      }
93
94      /**
95       * @return o nome do canal
96       * @see util.Mapeavel#getNome()
97       */
98      public String getNome()
99      {
100         return getClass().getSimpleName();
101     }
102
103     /**
104      * Da um passo na simulação do canal
105      */
106     public void proxPasso()
107     {
108         for (Portao portao : getPortoes())
109             portao.proxPasso();
110         calcCorrente();
111         notify(new ActionEvent(this, id, "passo dado"));
112     }
113
114     /**
115      * @return diferença de potencial entre os lados do canal.
116      */
117     public double getDDP()
118     {
119         return compartimento.getDDP();
120     }
121
122     /**
123      * @return potencial de repouso da membrana onde o canal esta.
124      */
125     public double getVRepouso()
126     {
127         return compartimento.getVRepouso();
128     }
129
```

```
130         /**
131          * Deve ser implementada por cada canal para criar os portões especificos
132          * daquele canal.
133          * @return valor de portoes.
134          */
135         public abstract Portao[] getPortoes();
136 
137         /**
138          * @return corrente do canal em microAmperes
139          */
140         public double getCorrente()
141         {
142             return corrente;
143         }
144 
145         /**
146          * @return tempo atual da simulção
147          * @see #getX()
148          */
149         public double getTempo()
150         {
151             return getX();
152         }
153 
154         /**
155          * @param evento o que aconteceu
156          */
157         public void notify(ActionEvent evento)
158         {
159             for (ActionListener listener : listeners)
160                 listener.actionPerformed(evento);
161         }
162 
163         /**
164          * @param a
165          */
166         public void addListener(ActionListener a)
167         {
168             id = ((TestMitral) a).addMapeado(this);
169             listeners.add(a);
170         }
171 
172         /**
173          * @return o tamanho do passo de tempo da simulação
174          * @see mitral.compartimento.Compartimento#getDeltaT()
175          */
176         public double getDeltaT()
177         {
178             return compartimento.getDeltaT();
179         }
180 
```

```
181         /**
182          * @return o tempo atual da simulção
183          * @see mitral.compartimento.Compartimento#getX()
184          * @deprecated substituido por {@link #getTempo()}
185          */
186         @Deprecated
187         public double getX()
188         {
189             return compartimento.getX();
190         }
191 
192         /**
193          * @return o valor atual da corrente ionica no canal em microAmpere por
194          *         centimetro quadrado
195          * @see util.Mapeavel#getY()
196          * @see #getCorrente()
197          * @deprecated substituido por {@link #getCorrente()}
198          */
199         @Deprecated
200         public double getY()
201         {
202             return getCorrente();
203         }
204 
205         /**
206          * Altera o tamanho do passo de tempo da simulação
207          * @param dt novo tamanho do passo de tempo
208          * @see mitral.compartimento.Compartimento#setDeltaT(double)
209          */
210         public void setDeltaT(double dt)
211         {
212             compartimento.setDeltaT(dt);
213         }
214     }
```

As classes cujo código é apresentado a seguir são tipos especiais de `Canal`, no caso, elas especificam as caracteristicas particulares de cada canal do modelo. Na ordem em que aparecem seus códigos, são elas `CanalCa`, `CanalK`, `CanalKA`, `CanalKCa`, `CanalKfast`, `CanalNa` e `CanalVazamento`.

```
1   package mitral.canal;
2   
3   import mitral.DecaiCa;
4   import mitral.compartimento.Compartimento;
5   import mitral.portao.Portao;
6   import mitral.portao.PortaoR;
7   import mitral.portao.PortaoS;
8   
```

```
 9  /**
10   * CanalCa do pacote mitral.canal do projeto tcc.<br>
11   * Representa o canal de Calcio do modelo de Davison.
12   * @author Rafael Arantes
13   */
14  public class CanalCa extends Canal
15  {
16      /**
17       * DecaiCa que representa a variação na concentração de calsio no
18       * compartimento onde esta este canal.
19       */
20      private DecaiCa decaiCa;
21
22      /**
23       * Cria um novo CanalCa
24       * @param compartimento onde o canal esta
25       * @param condutancia condutancia do canal em microSiemens por centimetro
26       *        quadrado.
27       */
28      public CanalCa(Compartimento compartimento, double condutancia)
29      {
30          super(compartimento);// constroi um canal generico
31          this.condutancia = condutancia;// altera a condutancia (que foi
32                                         // iniciada com 0)
33          decaiCa = new DecaiCa(this);
34      }
35
36      /**
37       * Da um passo na simulação, dando um passo do canal e um passo no
38       * decaimento da concentração de calsio
39       * @see Canal#proxPasso()
40       */
41      @Override
42      public void proxPasso()
43      {
44          super.proxPasso();
45          decaiCa.proxPasso();
46          super.getCompartimento().setConcCa(decaiCa.getY());
47      }
48
49      /**
50       * Pega a condutancia maxima do canal, e caso não tenha sido inicializada,
51       * inicializa com 0.
52       * @return condutancia maxima do canal em miliSiemens por centimetro
53       *         quadrado
54       * @see Canal#getCondutancia()
55       */
56      @Override
57      public double getCondutancia()
58      {
59          if (condutancia == null)
```

```
60              condutancia = 0.0;
61          return condutancia;
62      }
63
64      /**
65       * Pega o potencial de reversão do canal, e caso não tenha sido
66       * inicializado, inicializa com 70
67       * @return potencial de reversão do canal em milivols
68       * @see Canal#getPotencialReversao()
69       */
70      @Override
71      public double getPotencialReversao()
72      {
73          if (potencialReversao == null)
74              potencialReversao = 70.0;
75          return potencialReversao;
76      }
77
78      /**
79       * Pega os portões do canal, e caso não tenham sido criados, cria 1 do tipo
80       * S e 1 do tipo R
81       * @return os portões do canal
82       * @see Canal#getPortoes()
83       * @see PortaoS
84       * @see PortaoR
85       */
86      @Override
87      public Portao[] getPortoes()
88      {
89          if (portoes == null)
90          {
91              portoes = new Portao[2];
92              portoes[0] = new PortaoS(this);
93              portoes[1] = new PortaoR(this);
94          }
95          return portoes;
96      }
97  }
```

```
1   package mitral.canal;
2
3   import mitral.compartimento.Compartimento;
4   import mitral.portao.Portao;
5   import mitral.portao.PortaoA;
6   import mitral.portao.PortaoB;
7
8   /**
9    * CanalK do pacote mitral.canal do projeto tcc.<br>
10   * Representa o canal lento de potassio do modelo de Davison.
```

```
11   * @author Rafael Arantes
12   */
13  public class CanalK extends Canal
14  {
15      /**
16       * Cria um novo CanalK
17       * @param compartimento onde esta o canal
18       * @param condutancia condutancia do canal em microSiemens por centimetro
19       *        quadrado
20       */
21      public CanalK(Compartimento compartimento, double condutancia)
22      {
23          super(compartimento);
24          this.condutancia = condutancia;
25      }
26
27      /**
28       * Pega a condutancia maxima do canal, e caso não tenha sido inicializada,
29       * inicializa com 0
30       * @return condutancia maxima do canal em miliSiemens por centimetro
31       *         quadrado
32       * @see mitral.canal.Canal#getCondutancia()
33       */
34      @Override
35      public double getCondutancia()
36      {
37          if (condutancia == null)
38              condutancia = 0.0;
39          return condutancia;
40      }
41
42      /**
43       * Pega os portões do canal, e caso não tenham sido criados, cria 2 do tipo
44       * B e 1 do tipo A
45       * @return os portões do canal
46       * @see mitral.canal.Canal#getPortoes()
47       * @see PortaoB
48       * @see PortaoA
49       */
50      @Override
51      public Portao[] getPortoes()
52      {
53          if (portoes == null)
54          {
55              portoes = new Portao[3];
56              portoes[0] = new PortaoB(this);
57              portoes[1] = new PortaoB(this);
58              portoes[2] = new PortaoA(this);
59          }
60          return portoes;
61      }
```

```
62
63      /**
64       * Pega o potencial de reversão do canal, e caso não tenha sido
65       * inicializado, inicializa com -70
66       * @return potencial de reversão do canal em milivols
67       * @see mitral.canal.Canal#getPotencialReversao()
68       */
69      @Override
70      public double getPotencialReversao()
71      {
72          if (potencialReversao == null)
73              potencialReversao = -70.0;
74          return potencialReversao;
75      }
76  }
```

```
1   package mitral.canal;
2
3   import mitral.compartimento.Compartimento;
4   import mitral.portao.Portao;
5   import mitral.portao.PortaoP;
6   import mitral.portao.PortaoQ;
7
8   /**
9    * CanalKA do pacote mitral.canal do projeto tcc.<br>
10   * Representa o canal de potassio anômalo do modelo de Davison.
11   * @author Rafael Arantes
12   */
13  public class CanalKA extends Canal
14  {
15      /**
16       * Cria um novo CanalKA
17       * @param compartimento onde esta o canal
18       * @param condutancia condutancia do canal em microSiemens por centimetro
19       *        quadrado
20       */
21      public CanalKA(Compartimento compartimento, double condutancia)
22      {
23          super(compartimento);
24          this.condutancia = condutancia;
25      }
26
27      /**
28       * Pega a condutancia do canal, e caso não tenha sido inicializada,
29       * inicializa com 0
30       * @return condutancia do canal em miliSiemens por centimetro quadrado
31       * @see Canal#getCondutancia()
32       */
33      @Override
```

```
34      public double getCondutancia()
35      {
36          if (condutancia == null)
37              condutancia = 0.0;
38          return condutancia;
39      }
40
41      /**
42       * Pega os portões do canal, e caso não tenham sido criados, cria 1 do tipo
43       * P e 1 do tipo Q
44       * @return os portões do canal
45       * @see Canal#getPortoes()
46       * @see PortaoP
47       * @see PortaoQ
48       */
49      @Override
50      public Portao[] getPortoes()
51      {
52          if (portoes == null)
53          {
54              portoes = new Portao[2];
55              portoes[0] = new PortaoP(this);
56              portoes[1] = new PortaoQ(this);
57          }
58          return portoes;
59      }
60
61      /**
62       * Pega o potencial de reversão do canal, e caso não tenha sido
63       * inicializado, inicializa com -70
64       * @return potencial de reversão do canal em milivols
65       * @see Canal#getPotencialReversao()
66       */
67      @Override
68      public double getPotencialReversao()
69      {
70          if (potencialReversao == null)
71              potencialReversao = -70.0;
72          return potencialReversao;
73      }
74  }
```

```
1   package mitral.canal;
2
3   import mitral.compartimento.Compartimento;
4   import mitral.portao.Portao;
5   import mitral.portao.PortaoY;
6
7   /**
```

```
 8    * CanalKCa do pacote mitral.canal do projeto tcc.<br>
 9    * Representa o canal de potassio dependente de calsio do modelo de Davison.
10    * @author Rafael Arantes
11    */
12   public class CanalKCa extends Canal
13   {
14       /**
15        * Cria um novo CanalKCa
16        * @param compartimento onde esta o canal
17        * @param condutancia condutancia do canal em microSiemens por centimetro
18        *        quadrado
19        */
20       public CanalKCa(Compartimento compartimento, double condutancia)
21       {
22           super(compartimento);
23           this.condutancia = condutancia;
24       }
25
26       /**
27        * Pega a condutancia do canal, e caso não tenha sido inicializada,
28        * inicializa com 0
29        * @return condutancia do canal em miliSiemens por centimetro quadrado
30        * @see Canal#getCondutancia()
31        */
32       @Override
33       public double getCondutancia()
34       {
35           if (condutancia == null)
36               condutancia = 0.0;
37           return condutancia;
38       }
39
40       /**
41        * Pega os portões do canal, e caso não tenham sido criados, cria 1 do tipo
42        * Y
43        * @return os portões do canal
44        * @see Canal#getPortoes()
45        * @see PortaoY
46        */
47       @Override
48       public Portao[] getPortoes()
49       {
50           if (portoes == null)
51           {
52               portoes = new Portao[1];
53               portoes[0] = new PortaoY(this);
54           }
55           return portoes;
56       }
57
58       /**
```

```
59          * Pega o potencial de reversão do canal, e caso não tenha sido
60          * inicializado, inicializa com -80
61          * @return potencial de reversão do canal em milivols
62          * @see Canal#getPotencialReversao()
63          */
64         @Override
65         public double getPotencialReversao()
66         {
67             if (potencialReversao == null)
68                 potencialReversao = -80.0;
69             return potencialReversao;
70         }
71     }
```

```
1   package mitral.canal;
2
3   import mitral.compartimento.Compartimento;
4   import mitral.portao.Portao;
5   import mitral.portao.PortaoH;
6   import mitral.portao.PortaoM;
7
8   /**
9    * CanalNa do pacote mitral.canal do projeto tcc.<br>
10   * Representa o canal de sódio do modelo de Davison.
11   * @author Rafael Arantes
12   */
13  public class CanalNa extends Canal
14  {
15      /**
16       * Cria um novo CanalNa
17       * @param compartimento onde esta o canal
18       * @param condutancia condutancia do canal em microSiemens por centimetro
19       *        quadrado
20       */
21      public CanalNa(Compartimento compartimento, double condutancia)
22      {
23          super(compartimento);
24          this.condutancia = condutancia;
25      }
26
27      /**
28       * Pega a condutancia do canal, e caso não tenha sido inicializada,
29       * inicializa com 0
30       * @return condutancia do canal em miliSiemens por centimetro quadrado
```

```
31          * @see Canal#getCondutancia()
32          */
33         @Override
34         public double getCondutancia()
35         {
36             if (condutancia == null)
37                 condutancia = 0.0;
38             return condutancia;
39         }
40
41         /**
42          * Pega os portões do canal, e caso não tenham sido criados, cria 3 do tipo
43          * M e 1 do tipo H
44          * @return os portões do canal
45          * @see Canal#getPortoes()
46          * @see PortaoM
47          * @see PortaoH
48          */
49         @Override
50         public Portao[] getPortoes()
51         {
52             if (portoes == null)
53             {
54                 portoes = new Portao[4];
55                 portoes[0] = new PortaoM(this);
56                 portoes[1] = new PortaoM(this);
57                 portoes[2] = new PortaoM(this);
58                 portoes[3] = new PortaoH(this);
59             }
60             return portoes;
61         }
62
63         /**
64          * Pega o potencial de reversão do canal, e caso não tenha sido
65          * inicializado, inicializa com 45
66          * @return potencial de reversão do canal em milivols
67          * @see Canal#getPotencialReversao()
68          */
69         @Override
70         public double getPotencialReversao()
71         {
72             if (potencialReversao == null)
73                 potencialReversao = 45.0;
74             return potencialReversao;
75         }
76     }
```

```
1   package mitral.canal;
2
```

```
3  import mitral.compartimento.Compartimento;
4  import mitral.portao.Portao;
5
6  /**
7   * CanalVazamento do pacote mitral.canal do projeto tcc.<br>
8   * Implementa o canal de vazamento, por onde passam ions de diversos tipos.
9   * @author Rafael Arantes
10  */
11 public class CanalVazamento extends Canal
12 {
13     /**
14      * Construtor utilizado por Hodgkin-Huxley, controi um novo canal de
15      * vazamento com condutancia 0.3 e potencial de reversão -54.4
16      * @param compartimento compartimento ao qual pertence este canal de
17      *        vazamento
18      */
19     public CanalVazamento(Compartimento compartimento)
20     {
21         super(compartimento);
22     }
23
24     /**
25      * Constroi um novo canal de vazamento.
26      * @param compartimento compartimento ao qual pertence este canal de
27      *        vazamento
28      * @param potencialReversao Potencial de reversao do canal, quanto maior
29      *        mais corrente (positiva) passara por ele.
30      * @param condutancia condutancia do canal de vazamento.
31      */
32     public CanalVazamento(Compartimento compartimento,
33             double potencialReversao, double condutancia)
34     {
35         super(compartimento);
36         this.condutancia = condutancia;
37         this.potencialReversao = potencialReversao;
38     }
39
40     /**
41      * @return a condutancia do canal, o valor padrão é 0.3
42      * @see Canal#getCondutancia()
43      */
44     @Override
45     public double getCondutancia()
46     {
47         if (condutancia == null)
48             condutancia = 0.3;
49         return condutancia;
50     }
51
52     /**
53      * @return o potencial de reversão do canal, o valor padrão é -54.4
```

```
54          * @see Canal#getPotencialReversao()
55          */
56         @Override
57         public double getPotencialReversao()
58         {
59             if (potencialReversao == null)
60                 potencialReversao = -54.4;
61             return potencialReversao;
62         }
63
64         /**
65          * Constroi o Array de portões sem nenhuma posição, pois este canal não tem
66          * portões.
67          * @return o que foi construido
68          * @see Canal#getPortoes()
69          */
70         @Override
71         public Portao[] getPortoes()
72         {
73             if (portoes == null)
74                 portoes = new Portao[0];
75             return portoes;
76         }
77     }
```

2.2.2.3 Pacote `portao`

O pacote `portao` é constituído pelas classes `Portao`, `PortaoA`, `PortaoB`, `PortaoH`, `PortaoK`, `PortaoM`, `PortaoN`, `PortaoP`, `PortaoQ`, `PortaoR`, `PortaoS` e `PortaoY`.

A classe `Portao` define atributos e métodos comuns a todo portão (variável de ativação/inativação) implementado e seu código é mostrado a seguir.

```
1   package mitral.portao;
2
3   import java.awt.event.ActionEvent;
4   import java.awt.event.ActionListener;
5   import java.util.Vector;
6
7   import mitral.TestMitral;
8   import mitral.canal.Canal;
9   import util.Mapeavel;
10  import util.math.calculoNumerico.Euler;
11  import util.math.calculoNumerico.ODE;
```

```
12  import util.math.calculoNumerico.ODESolver;
13
14  /**
15   * Portao do pacote mitral.portao do projeto tcc.<br>
16   * Cada canal iônico é composto por portões, para que um determinado canal
17   * esteja aberto é necessario que todos os seus portões estejam abertos. Esta
18   * classe representa um portão genérico, implementando os métodos que são
19   * necessarios a todo portão.
20   * @see mitral.canal.Canal
21   * @author Rafael Arantes
22   */
23  public abstract class Portao implements ODE, Mapeavel
24  {
25      /**
26       * Canal que representa o canal ao qual esse portão pertence.
27       */
28      private Canal canal;
29
30      private double valor;
31
32      private Vector<ActionListener> listeners;
33
34      private ODESolver resolvedor;
35
36      private int id;
37
38      /**
39       * Constroi um novo Portao
40       * @param valor o valor inicial para o portão, o valor do portão representa
41       *        a porcentagem de portões que estam abertos em um determinado
42       *        momento.
43       * @param canal canal ao qual pertence este portão
44       */
45      public Portao(double valor, Canal canal)
46      {
47          this.canal = canal;
48          this.valor = valor;
49          resolvedor = new Euler(this);
50          listeners = new Vector<ActionListener>();
51      }
52
53      /**
54       * Constroi um novo Portao
55       * @param canal canal ao qual pertence este portão
56       */
57      public Portao(Canal canal)
58      {
59          this.canal = canal;
60          resolvedor = new Euler(this);
61          listeners = new Vector<ActionListener>();
62          valor = getAlpha() / (getAlpha() + getBeta());
```

```
63      }
64
65      /**
66       * Adiciona um objeto a lista de objetos que devem ser informados quando da
67       * ocorrencia de um evento.
68       * @param a objeto a ser adicionado
69       */
70      public void addListener(ActionListener a)
71      {
72          id = ((TestMitral) a).addMapeado(this);
73          listeners.add(a);
74      }
75
76      /**
77       * Informa os objetos que devem ser informados da ocorrencia do evento.
78       * @param evento o que aconteceu
79       */
80      public void notify(ActionEvent evento)
81      {
82          for (ActionListener listener : listeners)
83              listener.actionPerformed(evento);
84      }
85
86      /**
87       * Da um passo na simulação e notifica isso
88       */
89      public void proxPasso()
90      {
91          resolvedor.evaluateODEStep();
92          notify(new ActionEvent(this, id, "passo dado"));
93      }
94
95      /**
96       * Altera o valor do portão, ou seja a porcentagem de portões desse tipo
97       * abertos.
98       * @param valor novo valor do portão.
99       * @see util.math.calculoNumerico.ODE#atualiza(double)
100      */
101     public void atualiza(double valor)
102     {
103         this.valor = valor;
104     }
105
106     /**
107      * Derivada da equação que relaciona valor do portão com o tempo da
108      * simulação.
109      * @param tempo tempo no ponto em que a equação é calculada (variavel
110      *        independente)
111      * @param valor potencial de membrana no ponto em que a equação é calculada
112      *        (variavel dependente)
113      * @return valor da equação no ponto.
```

```
114         * @see util.math.calculoNumerico.ODE#fX(double, double)
115         */
116        public double fX(double tempo, double valor)
117        {
118            double retorno = getAlpha() * (1 - valor) - getBeta() * valor;
119            return retorno;
120        }
121
122        /**
123         * Alatera o tamanho do passo de tempo da simulação.
124         * @param dt novo tamanho do passo de tempo da simulação.
125         * @see util.math.calculoNumerico.ODE#setDeltaT(double)
126         * @see mitral.canal.Canal#setDeltaT(double)
127         */
128        public void setDeltaT(double dt)
129        {
130            canal.setDeltaT(dt);
131        }
132
133        /**
134         * @return o valor de alpha, ou seja, a probabilidade de um portão fechado
135         *         abrir.
136         */
137        public abstract double getAlpha();
138
139        /**
140         * @return o valor de beta, ou seja, a probabilidade de um portão aberto
141         *         fechar.
142         */
143        public abstract double getBeta();
144
145        /**
146         * @return o valor maximo possivel para o portão no potencial atual
147         */
148        public abstract double getInfinito();
149
150        /**
151         * @return o valor da constante de tempo para o potencial atual
152         */
153        public abstract double getTau();
154
155        /**
156         * @return o tamanho do passo de tempo da simulação
157         * @see util.math.calculoNumerico.ODE#getDeltaT()
158         * @see mitral.canal.Canal#getDeltaT()
159         */
160        public double getDeltaT()
161        {
162            return canal.getDeltaT();
163        }
164
```

```
165         /**
166          * @return tempo atual da simulação
167          * @see util.math.calculoNumerico.ODE#getX()
168          * @see util.Mapeavel#getY()
169          * @see mitral.canal.Canal#getTempo()
170          */
171         public double getX()
172         {
173             return canal.getTempo();
174         }
175 
176         /**
177          * @return valor atual do portão, ou seja a porcentagem de portões desse
178          *         tipo abertos
179          * @see util.math.calculoNumerico.ODE#getY()
180          * @see util.Mapeavel#getY()
181          * @deprecated substituido por {@link #getValor()}
182          */
183         @Deprecated
184         public double getY()
185         {
186             return getValor();
187         }
188 
189         /**
190          * @return canal ao qual este portão pertence.
191          */
192         public Canal getCanal()
193         {
194             return canal;
195         }
196 
197         /**
198          * @return valor atual do portão, ou seja a porcentagem de portões desse
199          *         tipo abertos
200          */
201         public double getValor()
202         {
203             return valor;
204         }
205 
206         /**
207          * @return Nome do portão.
208          * @see util.Mapeavel#getNome()
209          */
210         public String getNome()
211         {
212             return getClass().getSimpleName();
213         }
214     }
```

As classes cujos códigos são apresentados a seguir são tipos especiais de `Portao`. No caso, elas especificam as caracteristicas particulares de cada portão do modelo. Na ordem em que aparecem seus códigos, são elas `PortaoA`, `PortaoB`, `PortaoH`, `PortaoK`, `PortaoM`, `PortaoN`, `PortaoP`, `PortaoQ`, `PortaoR`, `PortaoS` e `PortaoY`.

```
1  package mitral.portao;
2
3  import java.io.BufferedReader;
4  import java.io.FileReader;
5  import java.io.IOException;
6
7  import mitral.canal.CanalK;
8  import util.Tabela;
9
10 /**
11  * PortaoA do pacote mitral.portao do projeto tcc.<br>
12  * Portão denominado k do canal de Potassio no modelo de Davison.
13  * @see mitral.canal.CanalK
14  * @author Rafael Arantes
15  */
16 public class PortaoA extends Portao
17 {
18     private Tabela infinitos;
19
20     /**
21      * Constroi um novo PortaoA
22      * @param canal canal ao qual pertence este portão
23      */
24     public PortaoA(CanalK canal)
25     {
26         super(canal);
27         if (infinitos == null)
28             carregaTabela();
29     }
30
31     /**
32      * Carrega a tabela de dos valores maximos do portão em cada potencial
33      */
34     private void carregaTabela()
35     {
36         infinitos = new Tabela();
37         try
38         {
39             BufferedReader arq = new BufferedReader(new FileReader(
40                     "arquivos/a.inf"));
41             String linha;
42             while ((linha = arq.readLine()) != null)
43                 infinitos.put(Double.valueOf(linha.split(" ", 2)[0]), Double
```

```
44                          .valueOf(linha.split(" ", 2)[1]));
45              arq.close();
46          }
47          catch (IOException e)
48          {
49              e.printStackTrace();
50          }
51      }
52
53      /**
54       * @return probabilidade de um portão fechado abrir, dada por
55       *         {@link #getInfinito()} / {@link #getTau()}
56       * @see Portao#getAlpha()
57       */
58      @Override
59      public double getAlpha()
60      {
61          return getInfinito() / getTau();
62      }
63
64      /**
65       * @return probabilidade de um portão aberto fechar, dada por (1 -
66       *         {@link #getInfinito()}) / {@link #getTau()}
67       * @see Portao#getBeta()
68       */
69      @Override
70      public double getBeta()
71      {
72          return (1 - getInfinito()) / getTau();
73      }
74
75      /**
76       * @return o valor maximo possivel para o portão no potencial atual
77       *         conseguido por consulta à tabela no arquivo a.inf
78       * @see Portao#getInfinito()
79       */
80      @Override
81      public double getInfinito()
82      {
83          double v = getCanal().getDDP();// potencial atual em milivolts
84          if (infinitos == null)
85              carregaTabela();
86          return infinitos.get(v);
87      }
88
89      /**
90       * @return o valor da constante de tempo para o potencial atual, ou seja,
91       *         200
92       * @see Portao#getTau()
93       */
94      @Override
```

```
95      public double getTau()
96      {
97          return 200;
98      }
99  }
```

```
1   package mitral.portao;
2
3   import java.io.BufferedReader;
4   import java.io.FileReader;
5   import java.io.IOException;
6
7   import mitral.canal.CanalK;
8   import util.Tabela;
9
10  /**
11   * PortaoB do pacote mitral.portao do projeto tcc.<br>
12   * Portão denominado n do canal de Potassio no modelo de Davison.
13   * @see mitral.canal.CanalK
14   * @author Rafael Arantes
15   */
16  public class PortaoB extends Portao
17  {
18      private Tabela infinitos;
19      private Tabela tau;
20
21      /**
22       * Constroi um novo PortaoB
23       * @param canal canal ao qual pertence este portão
24       */
25      public PortaoB(CanalK canal)
26      {
27          super(canal);
28          if ((infinitos == null) || (tau == null))
29              carregaTabelas();
30      }
31
32      /**
33       * Carrega a tabela de dos valores maximos do portão em cada potencial
34       */
35      private void carregaTabelas()
36      {
37          infinitos = new Tabela();
38          BufferedReader arq;
39          try
40          {
41              arq = new BufferedReader(new FileReader("arquivos/b.inf"));
42              String linha;
43              while ((linha = arq.readLine()) != null)
```

```
44                  infinitos.put(Double.valueOf(linha.split(" ", 2)[0]), Double
45                          .valueOf(linha.split(" ", 2)[1]));
46              arq.close();
47          }
48          catch (IOException e)
49          {
50              e.printStackTrace();
51          }
52          tau = new Tabela();
53          try
54          {
55              arq = new BufferedReader(new FileReader("arquivos/b.tau"));
56              String linha;
57              while ((linha = arq.readLine()) != null)
58                  tau.put(Double.valueOf(linha.split(" ", 2)[0]), Double
59                          .valueOf(linha.split(" ", 2)[1]));
60              arq.close();
61          }
62          catch (IOException e)
63          {
64              e.printStackTrace();
65          }
66      }
67
68      /**
69       * @return probabilidade de um portão fechado abrir, dada por
70       *         {@link #getInfinito()} / {@link #getTau()}
71       * @see Portao#getAlpha()
72       */
73      @Override
74      public double getAlpha()
75      {
76          return getInfinito() / getTau();
77      }
78
79      /**
80       * @return probabilidade de um portão aberto fechar, dada por (1 -
81       *         {@link #getInfinito()}) / {@link #getTau()}
82       * @see Portao#getBeta()
83       */
84      @Override
85      public double getBeta()
86      {
87          return (1 - getInfinito()) / getTau();
88      }
89
90      /**
91       * @return o valor maximo possivel para o portão no potencial atual
92       *         conseguido por consulta à tabela no arquivo b.inf
93       * @see Portao#getInfinito()
94       */
```

```
95      @Override
96      public double getInfinito()
97      {
98          double v = getCanal().getDDP();
99          if ((infinitos == null) || (tau == null))
100             carregaTabelas();
101         double retorno = infinitos.get(v);
102         return retorno;
103     }
104
105     /**
106      * @return o valor da constante de tempo no potencial atual conseguido por
107      *         consulta à tabela no arquivo b.tau
108      * @see Portao#getTau()
109      */
110     @Override
111     public double getTau()
112     {
113         double v = getCanal().getDDP();// potencial atual em milivolts
114         if ((infinitos == null) || (tau == null))
115             carregaTabelas();
116         double retorno = tau.get(v);
117         return retorno;
118     }
119 }
```

```
1   package mitral.portao;
2
3   import mitral.canal.CanalNa;
4
5   /**
6    * PortaoH do pacote mitral.portao do projeto tcc.<br>
7    * Mesmo portão h do modelo de Davison
8    * @see mitral.canal.CanalNa
9    * @author Rafael Arantes
10   */
11  public class PortaoH extends Portao
12  {
13      /**
14       * Constroi um novo PortaoH
15       * @param canal canal ao qual pertence este portão
16       */
17      public PortaoH(CanalNa canal)
18      {
19          super(canal);
20      }
21
22      /**
23       * @return a probabilidade de um portão fechado abrir, dada por
```

```
24          *         0.128/exp((v+38)/18)
25          * @see Portao#getAlpha()
26          */
27         @Override
28         public double getAlpha()
29         {
30             double v = getCanal().getDDP();
31             return 0.128 / Math.exp((v + 38) / 18);
32         }
33
34         /**
35          * @return probabilidade de um portão aberto fechar, dada por
36          *         4/(1+exp(-(v+15)/5))
37          * @see Portao#getBeta()
38          */
39         @Override
40         public double getBeta()
41         {
42             double v = getCanal().getDDP();
43             return 4 / (1 + Math.exp(-(v + 15) / 5));
44         }
45
46         /**
47          * @return o valor maximo possivel para o portão no potencial, dado por
48          *         {@link #getAlpha()} / ({@link #getAlpha()} + {@link #getBeta()})
49          * @see Portao#getInfinito()
50          */
51         @Override
52         public double getInfinito()
53         {
54             return getAlpha() / (getAlpha() + getBeta());
55         }
56
57         /**
58          * @return o valor da constante de tempo do portão no potencial, dado por 1 / ({@link
59          *         {@link #getBeta()})
60          * @see Portao#getTau()
61          */
62         @Override
63         public double getTau()
64         {
65             return 1.0 / (getAlpha() + getBeta());
66         }
67     }
```

```
1   package mitral.portao;
2
3   import java.io.BufferedReader;
4   import java.io.FileReader;
```

```
5  import java.io.IOException;
6
7  import mitral.canal.CanalKFast;
8  import util.Tabela;
9
10 /**
11  * PortaoK do pacote mitral.portao do projeto tcc.<br>
12  * Portão k do canal rapido de Potassio no modelo de Davison.
13  * @see mitral.canal.CanalKFast
14  * @author Rafael Arantes
15  */
16 public class PortaoK extends Portao
17 {
18     private Tabela infinitos;
19
20     /**
21      * Constroi um novo PortaoK
22      * @param canal canal ao qual pertence este portão
23      */
24     public PortaoK(CanalKFast canal)
25     {
26         super(canal);
27         if (infinitos == null)
28             carregaTabela();
29     }
30
31     private void carregaTabela()
32     {
33         infinitos = new Tabela();
34         try
35         {
36             BufferedReader arq = new BufferedReader(new FileReader(
37                     "arquivos/k.inf"));
38             String linha;
39             while ((linha = arq.readLine()) != null)
40                 infinitos.put(Double.valueOf(linha.split(" ", 2)[0]), Double
41                         .valueOf(linha.split(" ", 2)[1]));
42             arq.close();
43         }
44         catch (IOException e)
45         {
46             e.printStackTrace();
47         }
48     }
49
50     /**
51      * @return probabilidade de um portão fechado abrir, dada por
52      *         {@link #getInfinito()} / {@link #getTau()}
53      * @see Portao#getAlpha()
54      */
55     @Override
```

```
56      public double getAlpha()
57      {
58          return getInfinito() / getTau();
59      }
60
61      /**
62       * @return probabilidade de um portão aberto fechar, dada por (1 -
63       *         {@link #getInfinito()}) / {@link #getTau()}
64       * @see Portao#getBeta()
65       */
66      @Override
67      public double getBeta()
68      {
69          return (1 - getInfinito()) / getTau();
70      }
71
72      /**
73       * @return o valor maximo possivel para o portão no potencial atual
74       *         conseguido por consulta à tabela no arquivo k.inf
75       * @see Portao#getInfinito()
76       */
77      @Override
78      public double getInfinito()
79      {
80          double v = getCanal().getDDP();
81          // double v = (getCanal().getDDP() -
82          // getCanal().getVRepouso());//potencial atual - o de repouso em
83          // milivolts
84          if (infinitos == null)
85              carregaTabela();
86          return infinitos.get(v);
87      }
88
89      /**
90       * @return o valor da constante de tempo para o potencial atual, ou seja,
91       *         50
92       * @see Portao#getTau()
93       */
94      @Override
95      public double getTau()
96      {
97          return 50;
98      }
99  }
```

```
1   package mitral.portao;
2
3   import mitral.canal.CanalNa;
4
```

```
5   /**
6    * PortaoM do pacote mitral.portao do projeto tcc.<br>
7    * Mesmo portão m do modelo de Davison
8    * @see mitral.canal.CanalNa
9    * @author Rafael Arantes
10   */
11  public class PortaoM extends Portao
12  {
13      /**
14       * Constroi um novo PortaoM
15       * @param canal canal ao qual pertence este portão
16       */
17      public PortaoM(CanalNa canal)
18      {
19          super(canal);
20      }
21
22      /**
23       * @return a probabilidade de um portão fechado abrir, dada por
24       *         0.32*(v+42)/(1-exp(-(v+42)/4))
25       * @see Portao#getAlpha()
26       */
27      @Override
28      public double getAlpha()
29      {
30          double v = getCanal().getDDP();
31          double aux = v + 42;
32          return 0.32 * aux / (1 - Math.exp(-aux / 4));
33      }
34
35      /**
36       * @return probabilidade de um portão aberto fechar, dada por
37       *         0.28*(v+15)/(exp((v+15)/5)-1)
38       * @see Portao#getBeta()
39       */
40      @Override
41      public double getBeta()
42      {
43          double v = getCanal().getDDP();
44          double aux = v + 15;
45          return 0.28 * aux / (Math.exp(aux / 5) - 1);
46      }
47
48      /**
49       * @return o valor maximo possivel para o portão no potencial, dado por
50       *         {@link #getAlpha()} / ({@link #getAlpha()} + {@link #getBeta()})
51       * @see Portao#getInfinito()
52       */
53      @Override
54      public double getInfinito()
55      {
```

```
56          return getAlpha() / (getAlpha() + getBeta());
57      }
58
59      /**
60       * @return o valor da constante de tempo do portão no potencial, dado por 1 / ({@link
61       *         {@link #getBeta()})
62       * @see Portao#getTau()
63       */
64      @Override
65      public double getTau()
66      {
67          return 1.0 / (getAlpha() + getBeta());
68      }
69  }
```

```
1   package mitral.portao;
2
3   import java.io.BufferedReader;
4   import java.io.FileReader;
5   import java.io.IOException;
6
7   import mitral.canal.CanalKFast;
8   import util.Tabela;
9
10  /**
11   * PortaoN do pacote mitral.portao do projeto tcc.<br>
12   * Portão n do canal rapido de Potassio no modelo de Davison.
13   * @see mitral.canal.CanalKFast
14   * @author Rafael Arantes
15   */
16  public class PortaoN extends Portao
17  {
18      private Tabela infinitos;
19      private Tabela tau;
20
21      /**
22       * Constroi um novo PortaoN
23       * @param canal canal ao qual pertence este portão
24       */
25      public PortaoN(CanalKFast canal)
26      {
27          super(canal);
28          if ((infinitos == null) || (tau == null))
29              carregaTabelas();
30      }
31
32      private void carregaTabelas()
33      {
34          infinitos = new Tabela();
```

```
35          BufferedReader arq;
36          try
37          {
38              arq = new BufferedReader(new FileReader("arquivos/n.inf"));
39              String linha;
40              while ((linha = arq.readLine()) != null)
41                  infinitos.put(Double.valueOf(linha.split(" ", 2)[0]), Double
42                          .valueOf(linha.split(" ", 2)[1]));
43              arq.close();
44          }
45          catch (IOException e)
46          {
47              e.printStackTrace();
48          }
49          tau = new Tabela();
50          try
51          {
52              arq = new BufferedReader(new FileReader("arquivos/n.tau"));
53              String linha;
54              while ((linha = arq.readLine()) != null)
55                  tau.put(Double.valueOf(linha.split(" ", 2)[0]), Double
56                          .valueOf(linha.split(" ", 2)[1]));
57              arq.close();
58          }
59          catch (IOException e)
60          {
61              e.printStackTrace();
62          }
63      }
64
65      /**
66       * @return probabilidade de um portão fechado abrir, dada por
67       *         {@link #getInfinito()} / {@link #getTau()}
68       * @see mitral.portao.Portao#getAlpha()
69       */
70      @Override
71      public double getAlpha()
72      {
73          return getInfinito() / getTau();
74      }
75
76      /**
77       * @return probabilidade de um portão aberto fechar, dada por (1 -
78       *         {@link #getInfinito()}) / {@link #getTau()}
79       * @see mitral.portao.Portao#getBeta()
80       */
81      @Override
82      public double getBeta()
83      {
84          return (1 - getInfinito()) / getTau();
85      }
```

```
86
87      /**
88       * @return o valor maximo possivel para o portão no potencial atual
89       *         conseguido por consulta à tabela no arquivo n.inf
90       * @see mitral.portao.Portao#getInfinito()
91       */
92      @Override
93      public double getInfinito()
94      {
95          double v = getCanal().getDDP();
96          if ((infinitos == null) || (tau == null))
97              carregaTabelas();
98          return infinitos.get(v);
99      }
100
101     /**
102      * @return o valor da constante de tempo no potencial atual conseguido por
103      *         consulta à tabela no arquivo n.tau
104      * @see mitral.portao.Portao#getTau()
105      */
106     @Override
107     public double getTau()
108     {
109         double v = getCanal().getDDP();
110         if ((infinitos == null) || (tau == null))
111             carregaTabelas();
112         return tau.get(v);
113     }
114 }
```

```
1   package mitral.portao;
2
3   import mitral.canal.CanalKA;
4
5   /**
6    * PortaoP do pacote mitral.portao do projeto tcc.<br>
7    * Mesmo portão p do modelo de Davison
8    * @see mitral.canal.CanalKA
9    * @author Rafael Arantes
10   */
11  public class PortaoP extends Portao
12  {
13      /**
14       * Constroi um novo PortaoP
15       * @param canal canal ao qual pertence este portão
16       */
17      public PortaoP(CanalKA canal)
18      {
19          super(canal);
```

```
20      }
21
22      /**
23       * @return probabilidade de um portão fechado abrir, dada por
24       *         {@link #getInfinito()} / {@link #getTau()}
25       * @see Portao#getAlpha()
26       */
27      @Override
28      public double getAlpha()
29      {
30          return getInfinito() / getTau();
31      }
32
33      /**
34       * @return probabilidade de um portão aberto fechar, dada por (1 -
35       *         {@link #getInfinito()}) / {@link #getTau()}
36       * @see Portao#getBeta()
37       */
38      @Override
39      public double getBeta()
40      {
41          return (1 - getInfinito()) / getTau();
42      }
43
44      /**
45       * @return o valor maximo possivel para o portão no potencial atual, dada
46       *         por 1.0/(1+exp(-(v+42)/13.0))
47       * @see Portao#getInfinito()
48       */
49      @Override
50      public double getInfinito()
51      {
52          double v = getCanal().getDDP();
53          return 1.0 / (1 + Math.exp(-(v + 42) / 13.0));
54      }
55
56      /**
57       * @return o valor da constante de tempo para o potencial atual, ou seja,
58       *         1.38
59       * @see Portao#getTau()
60       */
61      @Override
62      public double getTau()
63      {
64          return 1.38;
65      }
66  }
```

```
1   package mitral.portao;
```

```
2
3   import mitral.canal.CanalKA;
4
5   /**
6    * PortaoQ do pacote mitral.portao do projeto tcc.<br>
7    * Mesmo portão q do modelo de Davison
8    * @see mitral.canal.CanalKA
9    * @author Rafael Arantes
10   */
11  public class PortaoQ extends Portao
12  {
13      /**
14       * Constroi um novo PortaoQ
15       * @param canal canal ao qual pertence este portão
16       */
17      public PortaoQ(CanalKA canal)
18      {
19          super(canal);
20      }
21
22      /**
23       * @return probabilidade de um portão fechado abrir, dada por
24       *         {@link #getInfinito()} / {@link #getTau()}
25       * @see Portao#getAlpha()
26       */
27      @Override
28      public double getAlpha()
29      {
30          return getInfinito() / getTau();
31      }
32
33      /**
34       * @return probabilidade de um portão aberto fechar, dada por (1 -
35       *         {@link #getInfinito()}) / {@link #getTau()}
36       * @see Portao#getBeta()
37       */
38      @Override
39      public double getBeta()
40      {
41          return (1 - getInfinito()) / getTau();
42      }
43
44      /**
45       * @return o valor maximo possivel para o portão no potencial atual, dada
46       *         por 1.0/(1+exp((v+110)/18))
47       * @see Portao#getInfinito()
48       */
49      @Override
50      public double getInfinito()
51      {
52          double v = getCanal().getDDP();
```

```
53          return 1.0 / (1 + Math.exp((v + 110) / 18));
54      }
55
56      /**
57       * @return o valor da constante de tempo para o potencial atual, ou seja,
58       *         150
59       * @see Portao#getTau()
60       */
61      @Override
62      public double getTau()
63      {
64          return 150.0;
65      }
66  }
```

```
1   package mitral.portao;
2
3   import mitral.canal.CanalCa;
4
5   /**
6    * PortaoR do pacote mitral.portao do projeto tcc.<br>
7    * Mesmo portão r do modelo de Davison
8    * @see mitral.canal.CanalCa
9    * @author Rafael Arantes
10   */
11  public class PortaoR extends Portao
12  {
13      /**
14       * Constroi um novo PortaoR
15       * @param canal canal ao qual pertence este portão
16       */
17      public PortaoR(CanalCa canal)
18      {
19          super(canal);
20      }
21
22      /**
23       * @return a probabilidade de um portão fechado abrir, dada por
24       *         0.0068/(1+exp((30+v)/12))
25       * @see Portao#getAlpha()
26       */
27      @Override
28      public double getAlpha()
29      {
30          double v = getCanal().getDDP();
31          return 0.0068 / (1 + Math.exp((30 + v) / 12));
32      }
33
34      /**
```

```
35          * @return probabilidade de um portão aberto fechar, dada por
36          *         0.06/(1+exp(-v/11))
37          * @see Portao#getBeta()
38          */
39         @Override
40         public double getBeta()
41         {
42             double v = getCanal().getDDP();
43             return 0.06 / (1 + Math.exp(-v / 11));
44         }
45
46         /**
47          * @return o valor maximo possivel para o portão no potencial, dado por
48          *         {@link #getAlpha()} / ({@link #getAlpha()} + {@link #getBeta()})
49          * @see Portao#getInfinito()
50          */
51         @Override
52         public double getInfinito()
53         {
54             return getAlpha() / (getAlpha() + getBeta());
55         }
56
57         /**
58          * @return o valor da constante de tempo do portão no potencial, dado por 1 / ({@link
59          *         {@link #getBeta()})
60          * @see Portao#getTau()
61          */
62         @Override
63         public double getTau()
64         {
65             return 1.0 / (getAlpha() + getBeta());
66         }
67     }
```

```
1   package mitral.portao;
2
3   import mitral.canal.CanalCa;
4
5   /**
6    * PortaoS do pacote mitral.portao do projeto tcc.<br>
7    * Mesmo portão s do modelo de Davison
8    * @see mitral.canal.CanalCa
9    * @author Rafael Arantes
10   */
11  public class PortaoS extends Portao
12  {
13      /**
14       * Constroi um novo PortaoS
15       * @param canal canal ao qual pertence este portão
```

```
16          */
17         public PortaoS(CanalCa canal)
18         {
19             super(canal);
20         }
21
22         /**
23          * @return a probabilidade de um portão fechado abrir, dada por 7.5 / (1 +
24          *         exp((13 - v) / 7))
25          * @see Portao#getAlpha()
26          */
27         @Override
28         public double getAlpha()
29         {
30             double v = getCanal().getDDP();
31             return 7.5 / (1 + Math.exp((13 - v) / 7));
32         }
33
34         /**
35          * @return probabilidade de um portão aberto fechar, dada por 1.65 / (1 +
36          *         Math.exp((v - 14) / 4))
37          * @see Portao#getBeta()
38          */
39         @Override
40         public double getBeta()
41         {
42             double v = getCanal().getDDP();
43             return 1.65 / (1 + Math.exp((v - 14) / 4));
44         }
45
46         /**
47          * @return o valor maximo possivel para o portão no potencial, dado por
48          *         {@link #getAlpha()} / ({@link #getAlpha()} + {@link #getBeta()})
49          * @see Portao#getInfinito()
50          */
51         @Override
52         public double getInfinito()
53         {
54             return getAlpha() / (getAlpha() + getBeta());
55         }
56
57         /**
58          * @return o valor da constante de tempo do portão no potencial, dado por 1 / ({@link
59          *         {@link #getBeta()})
60          * @see Portao#getTau()
61          */
62         @Override
63         public double getTau()
64         {
65             return 1.0 / (getAlpha() + getBeta());
66         }
```

```
67  }
```

```
1   package mitral.portao;
2
3   import mitral.canal.CanalKCa;
4
5   /**
6    * PortaoY do pacote mitral.portao do projeto tcc.<br>
7    * Mesmo portão y do modelo de Davison
8    * @see mitral.canal.CanalKCa
9    * @author Rafael Arantes
10   */
11  public class PortaoY extends Portao
12  {
13      /**
14       * Constroi um novo PortaoY
15       * @param canal canal ao qual pertence este portão
16       */
17      public PortaoY(CanalKCa canal)
18      {
19          super(canal);
20      }
21
22      /**
23       * @return a probabilidade de um portão fechado abrir, dada pela
24       *         multiplicação de um termo dependente da voltagem e outro
25       *         dependente da concentração de Calsio. A parcela dependente de V é
26       *         exp((v+70)/27). A dependente de concCa é
27       *         500*(0.015-concCa)/(exp((0.015-concCa)/0.0013)-1) se concCa<0.01
28       *         e 500*0.005/(Math.exp(0.005/0.0013)-1) caso contrario. Estas
29       *         equações foram copiadas diretamente do modelo de Davison.
30       * @see Portao#getAlpha()
31       */
32      @Override
33      public double getAlpha()
34      {
35          double vDep;
36          double concDep;
37          double v = getCanal().getDDP();
38          double concCa = ((CanalKCa) getCanal()).getCompartimento().getConcCa();
39          vDep = Math.exp((v + 70) / 27);
40          double aux = 0.015 - Math.min(0.01, concCa);
41          concDep = 500 * aux / (Math.exp(aux / 0.0013) - 1);
42          return vDep * concDep;
43      }
44
45      /**
46       * @return probabilidade de um portão aberto fechar, que é 0.05
47       * @see Portao#getBeta()
```

```
48          */
49         @Override
50         public double getBeta()
51         {
52             return 0.05;
53         }
54 
55         /**
56          * @return o valor maximo possivel para o portão no potencial, dado por
57          *         {@link #getAlpha()} / ({@link #getAlpha()} + {@link #getBeta()})
58          * @see Portao#getInfinito()
59          */
60         @Override
61         public double getInfinito()
62         {
63             return getAlpha() / (getAlpha() + getBeta());
64         }
65 
66         /**
67          * @return o valor da constante de tempo do portão no potencial, dado por 1 / ({@link
68          *         {@link #getBeta()})
69          * @see Portao#getTau()
70          */
71         @Override
72         public double getTau()
73         {
74             return 1.0 / (getAlpha() + getBeta());
75         }
76     }
```

A figura 2.1 apresenta um diagrama de classe simplificado do sistema, nele os métodos e atributos de cada classe foram suprimidos, assim como relações entre classes de pacotes distintos. Esta simplificação foi feita para que o diagrama tivesse um tamanho compatível com a pagina deste trabalho e não apresentasse muitas relações entre classes que só dificultariam a sua compreensão.

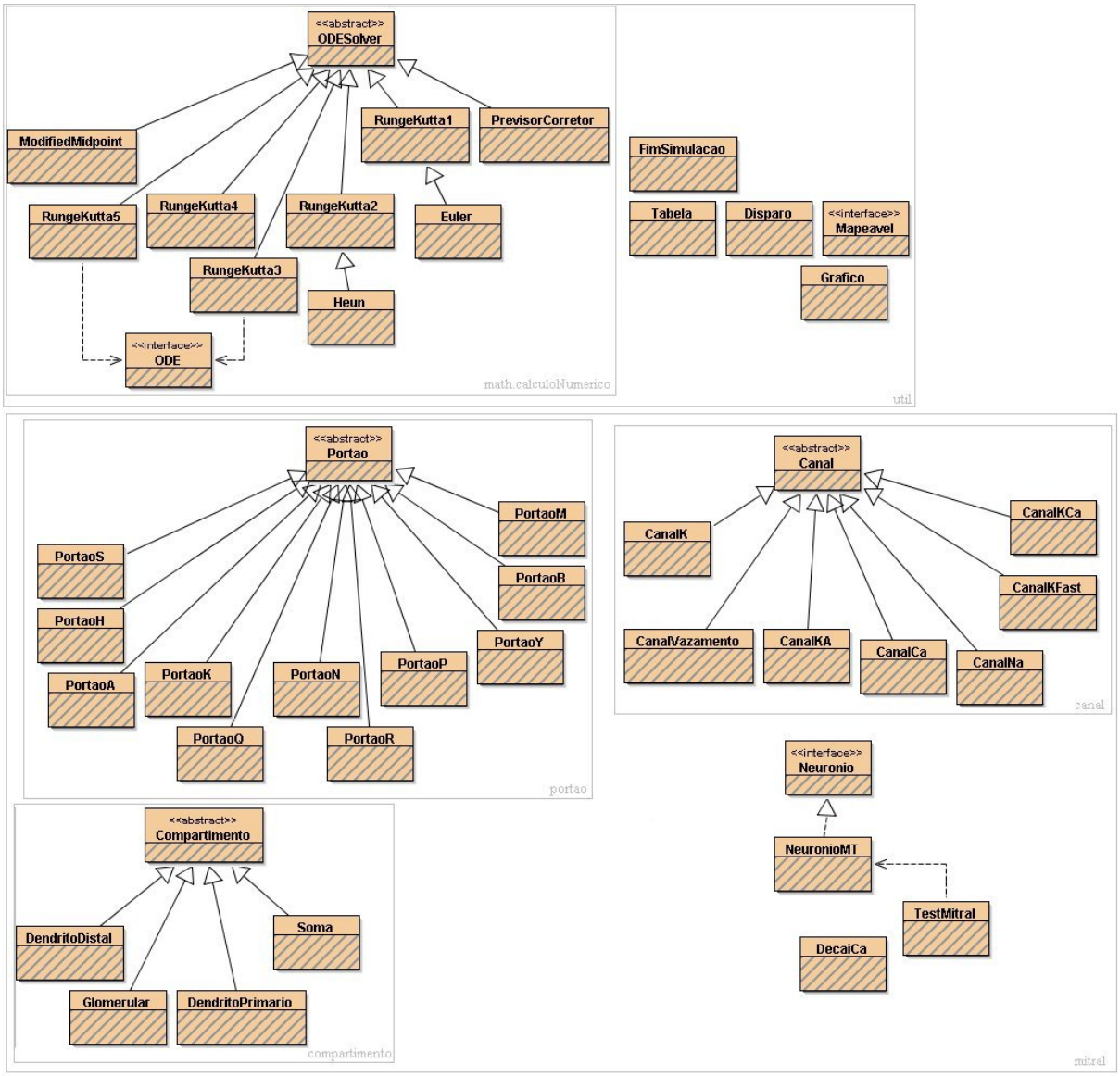

Figura 2.1: Diagrama de classe simplificado do sistema

3 Resultados

Uma vez implementado o modelo conforme descrito no capítulo anterior, foram realizados testes injetando-se diferentes valores de correntes nos compartimentos glomerular e soma. Os resultados permitiram a construção de gráficos da evolução temporal da diferença de potencial de membrana de cada compartimento do neurônio. Os valores de corrente foram escolhidos de forma que se pudesse comparar os resultados obtidos nessas simulações com aqueles obtidos por Davison, Feng e Brown (2000).

A resolução das equações diferenciais foi realizada pelo método de Euler utilizando-se passos de tempo de 0,001ms. Os gráficos foram plotados tomando-se alguns pontos amostrais durante a simulação, não sendo representados todos os pontos calculados, uma vez que a resolução espacial do monitor não permitiria tal plotagem e seu custo em temos de memória tornaria a simulação impraticável. A figura 3.1 apresenta o gráfico obtido ao se injetar a densidade de corrente de $0,2\mu A/cm^2$ no compartimento glomerular. A corrente começou a ser injetada no instante 10 ms e foi mantida constante por toda a duração da simulação. Comparando esse gráfico com o produzido por Davison, Feng e Brown (2000, FIG. 6B), nota-se uma leve diferença entre o comportamento dos dois quanto a freqüência e instante do primeiro disparo.

A figura 3.2 apresenta o gráfico obtido ao se injetar a densidade de corrente de $0,2\mu A/cm^2$ no compartimento somático. A corrente começou a ser injetada no instante 10 ms e foi mantida constante por toda a simulação. Comparando esse gráfico com o produzido por Davison, Feng e Brown (2000, FIG. 6A), nota-se uma leve diferença entre o comportamento dos dois quanto a freqüência e instante do primeiro disparo.

A figura 3.3 apresenta o gráfico obtido ao se injetar a densidade de corrente de $0,4\mu A/cm^2$ no compartimento somático. A corrente começou a ser injetada no instante 50 ms e foi mantida constante por toda a simulação. Comparando esse gráfico com o produzido por Davison, Feng e

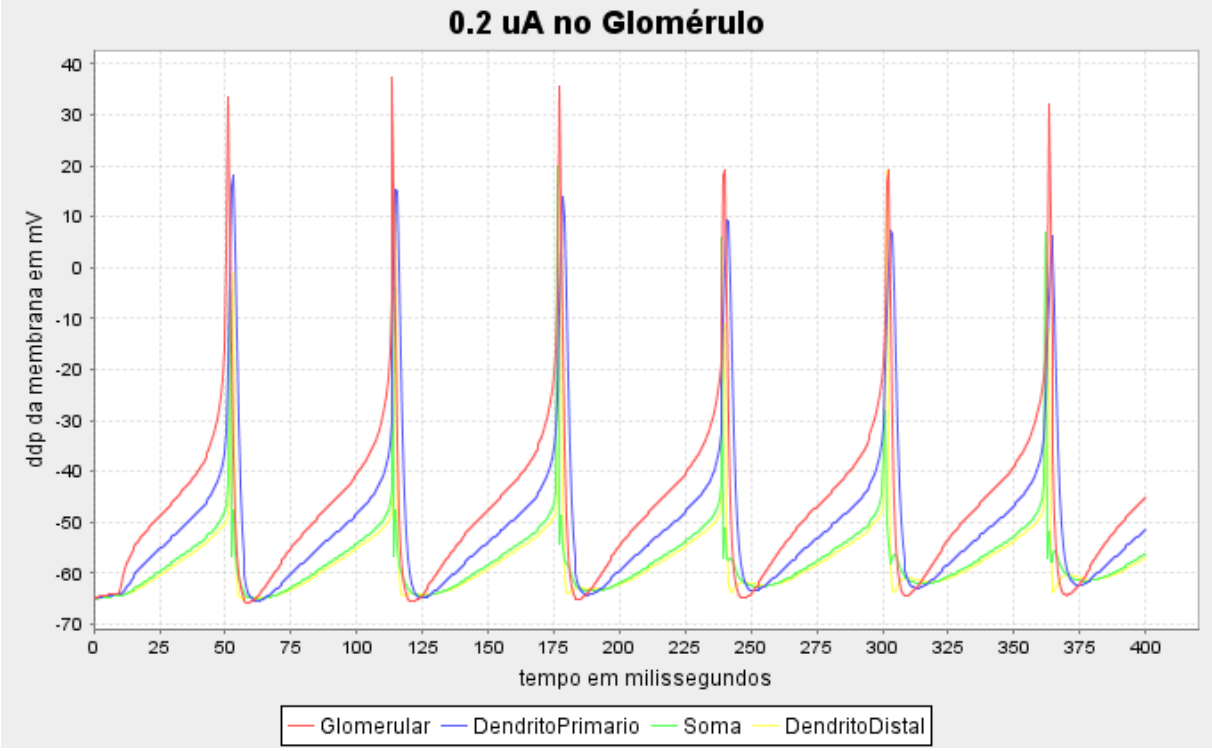

Figura 3.1: Gráfico obtido ao se injetar a densidade de corrente de $0,2\mu A/cm^2$ no compartimento glomerular. A corrente começou a ser injetada no instante 10 ms e durou por toda a simulação

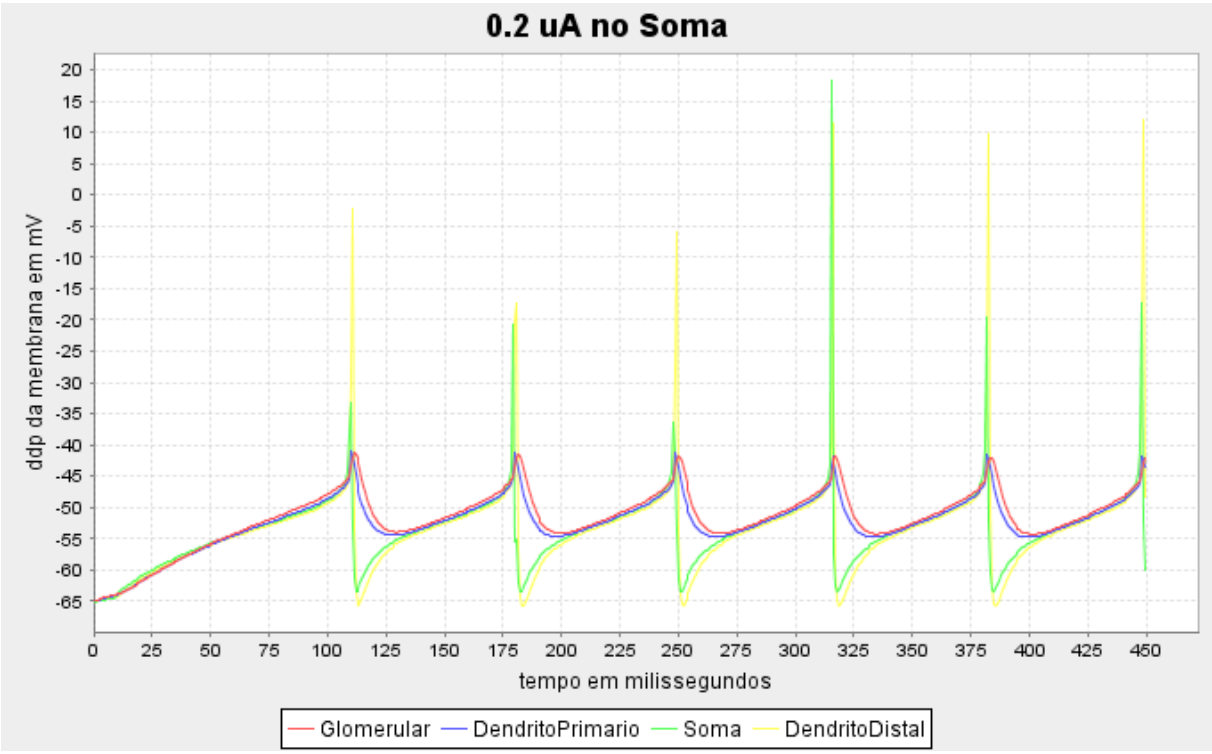

Figura 3.2: Gráfico obtido ao se injetar a densidade de corrente de $0,2\mu A/cm^2$ no compartimento somático. A corrente começou a ser injetada no instante 10 ms e durou por toda a simulação.

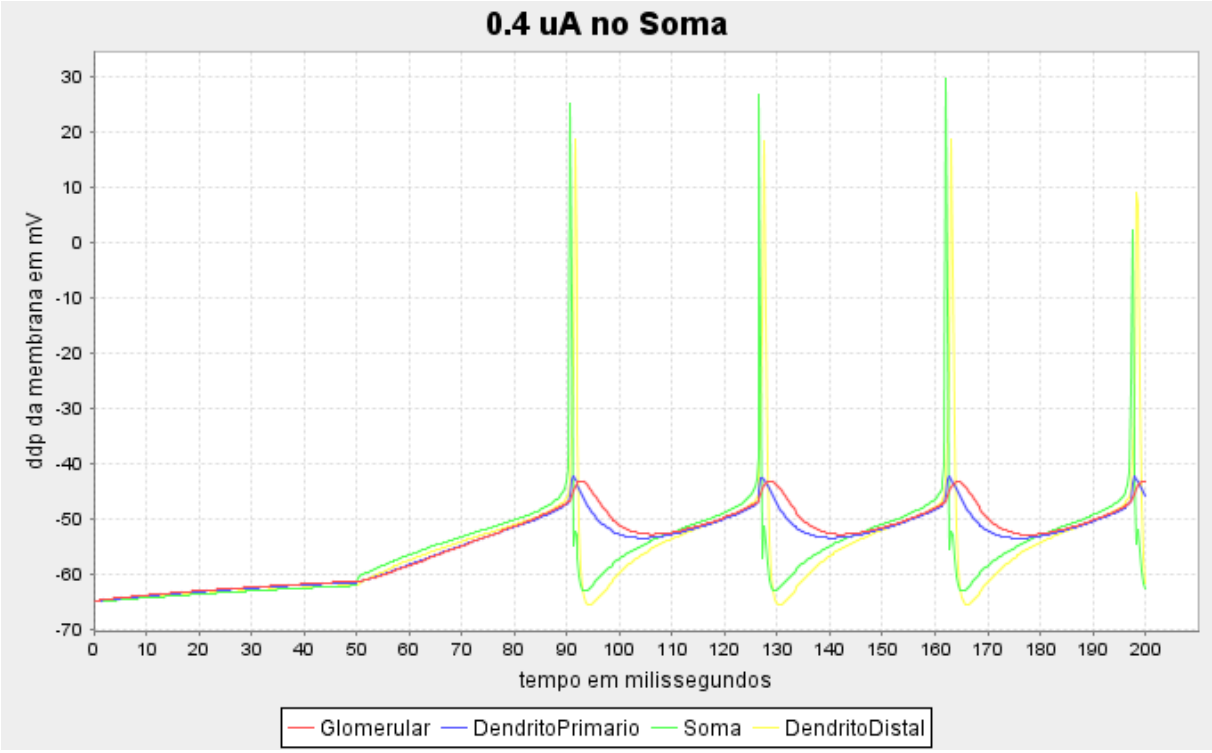

Figura 3.3: Gráfico obtido ao se injetar a densidade de corrente de $0,4\mu A/cm^2$ no compartimento somático. A corrente começou a ser injetada no instante 50 ms e durou por toda a simulação.

Brown (2000, FIG. 3B), nota-se um comportamento semelhante entre os dois, tanto com relação ao intante do primeiro disparo, quanto com relação a freqüência.

A figura 3.4 apresenta o gráfico obtido ao se injetar a densidade de corrente de $0,8\mu A/cm^2$ no compartimento somático. A corrente começou a ser injetada no instante 50 ms e foi mantida constante por toda a simulação. Comparando esse gráfico com o produzido por Davison, Feng e Brown (2000, FIG. 4B), nota-se um comportamento semelhante entre os dois, tanto com relação ao intante do primeiro disparo, quanto com relação a freqüência.

A figura 3.5 apresenta o gráfico obtido ao se injetar a densidade de corrente de $1,6\mu A/cm^2$ no compartimento glomerular. A corrente começou a ser injetada no instante 50 ms e foi mantida constante por toda a simulação. Comparando esse gráfico com o produzido por Davison, Feng e Brown (2000, FIG. 6B), nota-se um comportamento semelhante entre os dois com relação a freqüência, porém no presente trabalho o neurônio apresenta um disparo assim que a corrente é injetada o que não chega a ocorrer no modelo original, embora ele apresente uma leve ondulação.

A figura 3.6 apresenta o gráfico obtido ao se injetar a densidade de corrente de $1,6\mu A/cm^2$

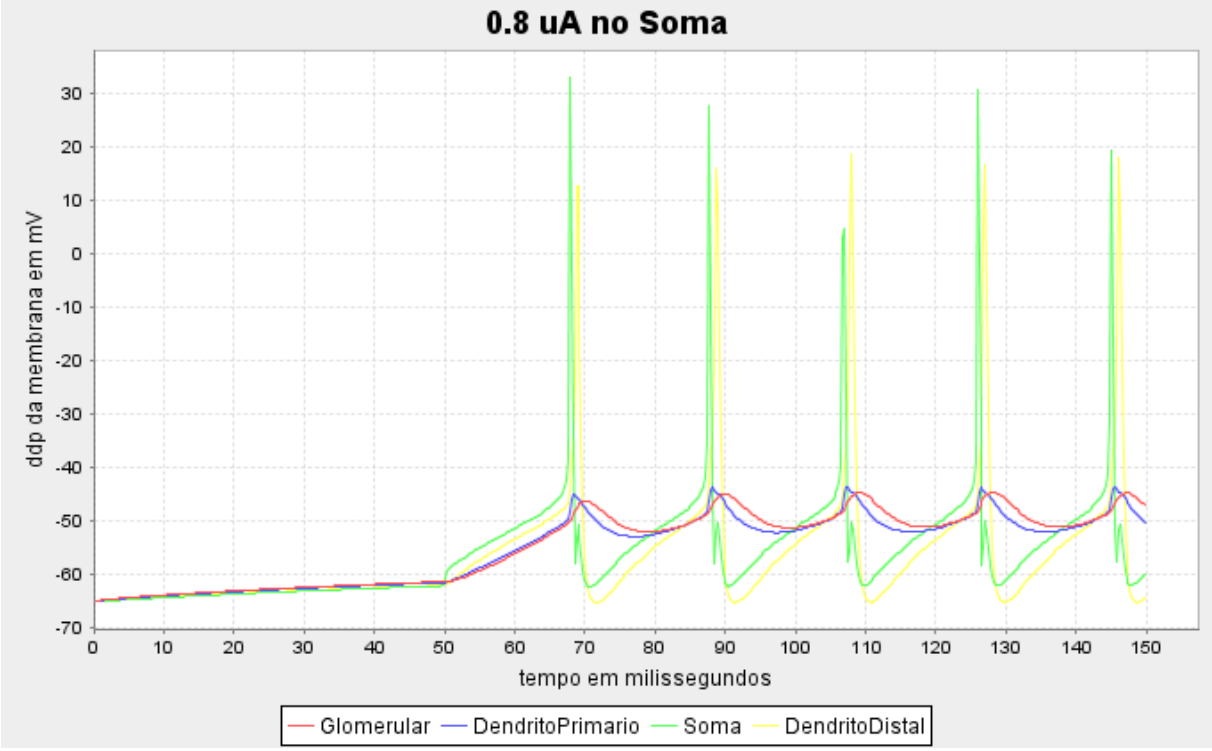

Figura 3.4: Gráfico obtido ao se injetar a densidade de corrente de $0,8\mu A/cm^2$ no compartimento somático. A corrente começou a ser injetada no instante 50 ms e foi mantida constante por toda a simulação.

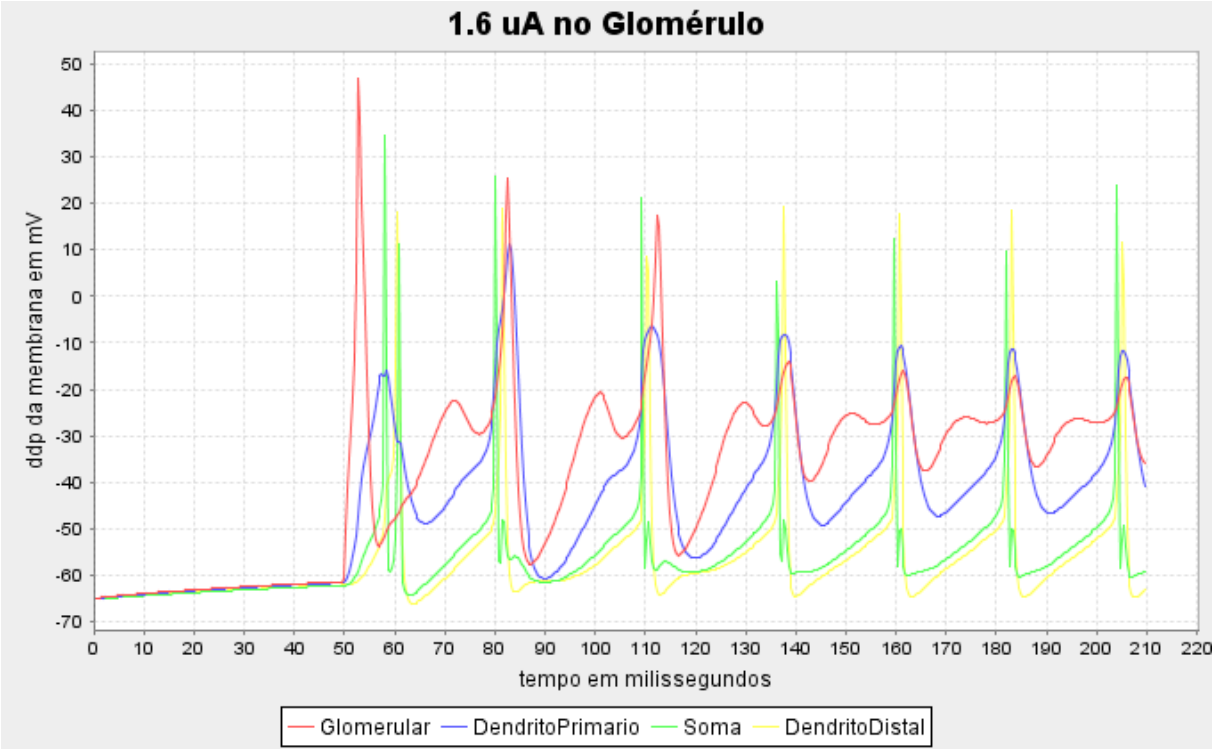

Figura 3.5: Gráfico obtido ao se injetar a densidade de corrente de $1,6\mu A/cm^2$ no compartimento glomerular. A corrente começou a ser injetada no instante 50 ms e durou por toda a simulação.

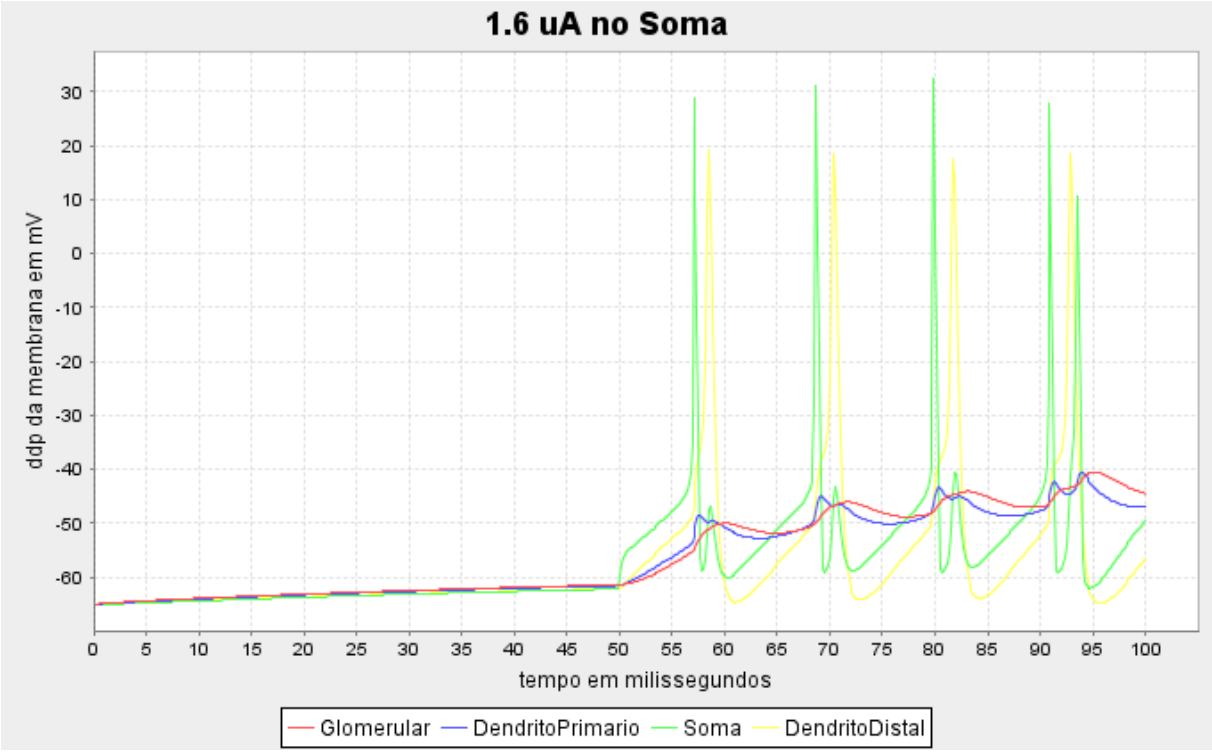

Figura 3.6: Gráfico obtido ao se injetar a densidade de corrente de $1,6\mu A/cm^2$ no compartimento somático. A corrente começou a ser injetada no instante 50 ms e durou por toda a simulação.

no compartimento somático. A corrente começou a ser injetada no instante 50 ms e foi mantida constante por toda a simulação. Comparando esse gráfico com o produzido por Davison, Feng e Brown (2000, FIG. 6A), nota-se um comportamento semelhante entre os dois, tanto com relação ao intante do primeiro disparo, quanto com relação a freqüência. Repare ainda, na leve subida do potencial logo após cada disparo, no presente trabalho o último disparo teve esse efeito tão forte que causou mais um disparo que não deveria ter ocorrido. Este efeito provavelmente se deve a característica de *stiffness* .

Outro teste realizado consistiu em aplicar valores diferentes de corrente, tanto no glomérulo como no soma, e medir a freqüência de disparos da célula bem como a latência do primeiro disparo, do inicio da simulação até o instante do primeiro disparo. Os resultados destes testes são apresentados nos gráficos a seguir.

A figura 3.7 mostra a freqüência com que o soma da célula dispara quando se aplicam diferentes valores de corrente no glomérulo e a figura 3.8 quando a corrente é aplicada no seu corpo celular (soma). Uma vez que todos os potenciais de ação apresentam aproximadamente as mesmas formas e valores, acredita-se que as informações são codificadas pelos neurônios em

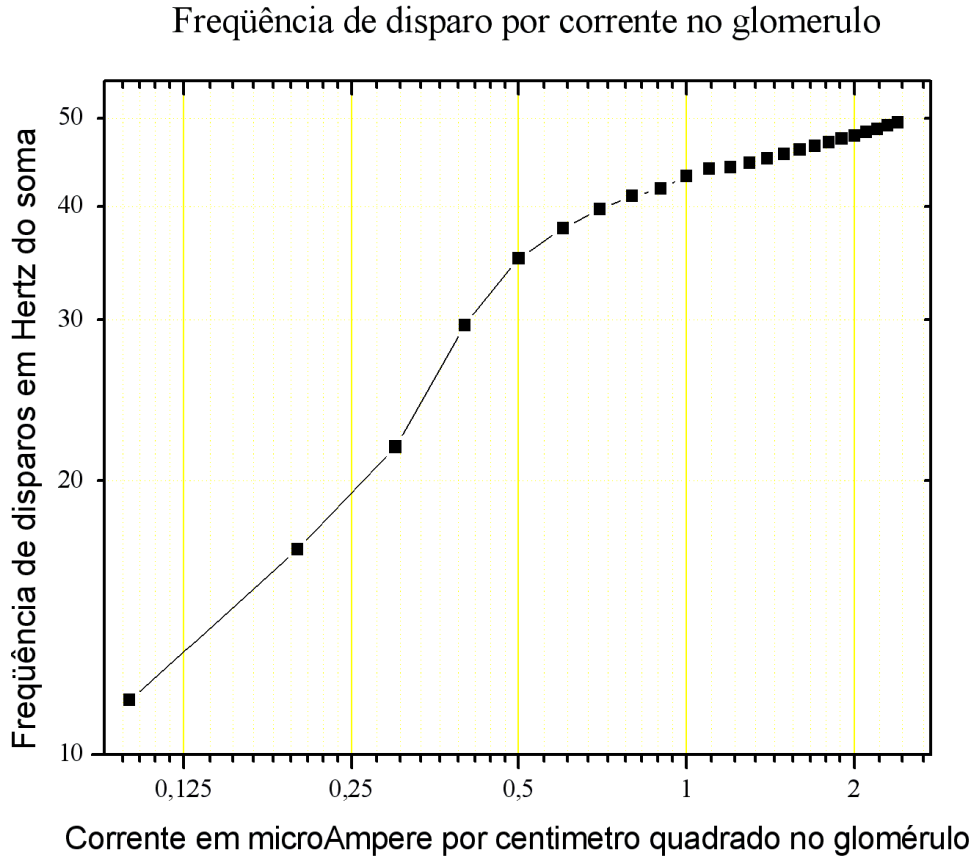

Figura 3.7: Freqüência de disparos da célula quando se aplicam diferentes valores de corrente no seu glomérulo

freqüências de disparo distintas para informações diferentes, daí a importância de tal função. Comparando-se este gráfico com os apresentados por Davison, Feng e Brown (2000, FIG. 7B e FIG. 7A, respectivamente) percebe-se uma grande semelhança entre eles. Repare que o gráfico apresentado aqui mostra o comportamento da frequência de disparos para valores de corrente acima dos considerados no modelo original, isto foi feito apenas para verificar como o modelo se comporta em tais situações.

A figura 3.9 mostra a latência do primeiro disparo do soma da célula, para diferentes valores de corrente aplicadas no glomérulo. A latência também é uma grandeza importante para a codificação de odores pelo bulbo olfativo Davison, Feng e Brown (2003). Comparando-se este gráfico com o obtido por Davison, Feng e Brown (2000, FIG. 7D) percebe-se uma grande semelhança entre eles.

A figura 3.10 mostra a latência do primeiro disparo do soma da célula, para diferentes valores de corrente aplicadas no próprio soma. Comparando-se este gráfico com o obtido por Davison, Feng e Brown (2000, FIG. 7C) percebe-se uma grande semelhança.

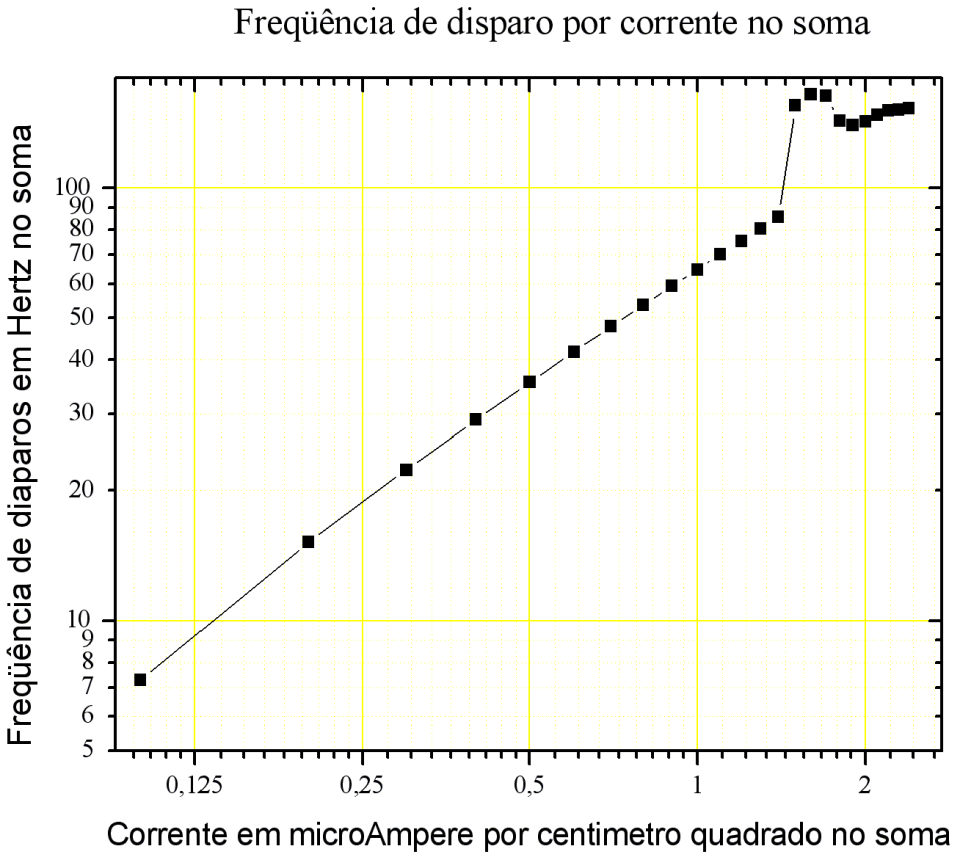

Figura 3.8: Freqüência de disparos da célula dispara quando se aplicam diferentes valores de corrente no seu corpo celular.

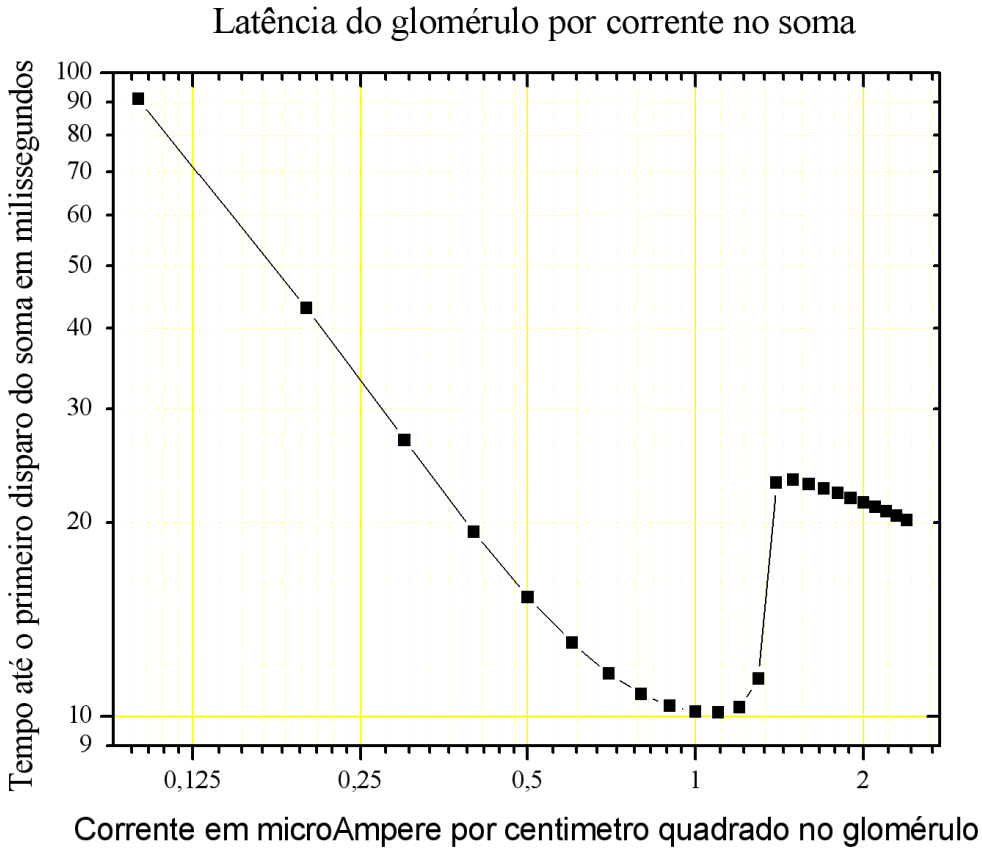

Figura 3.9: Latência do primeiro disparo da célula, para diferentes valores de corrente aplicadas no seu glomérulo.

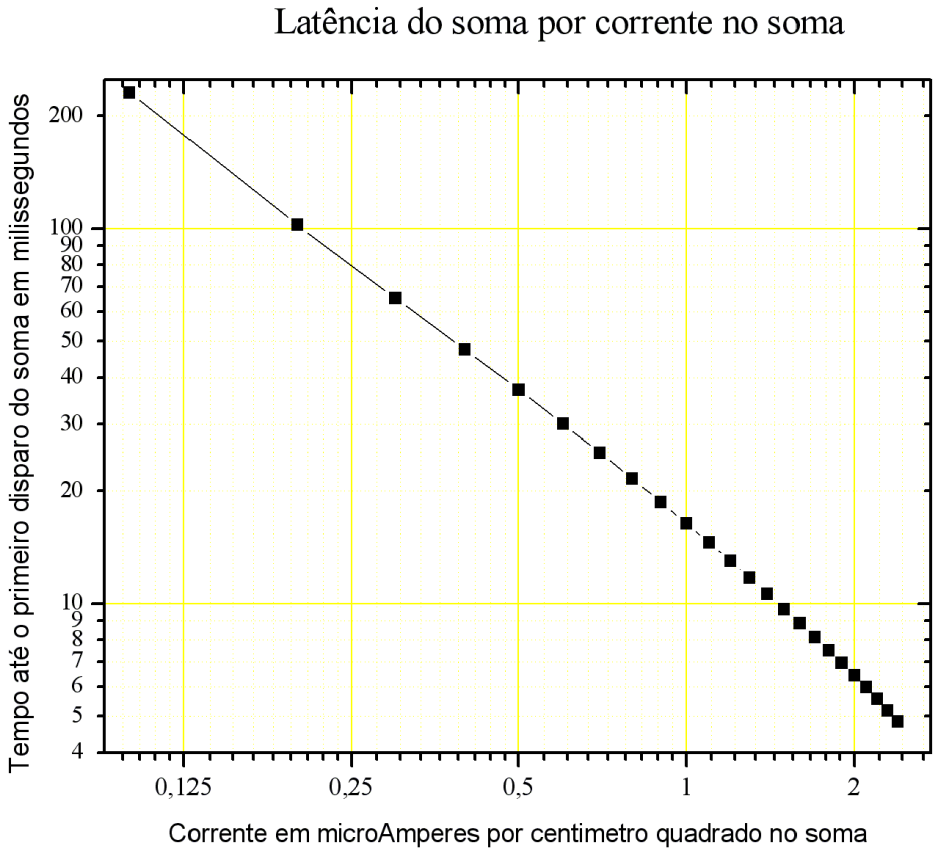

Figura 3.10: Latência do primeiro disparo do soma da célula, para diferentes valores de corrente aplicadas no próprio soma

4 *Discusões e Conclusões*

O principal resultado deste trabalho foi a implementação, em Java, do modelo de célula mitral do bulbo olfativo de (DAVISON; FENG; BROWN, 2000). O fato de se ter o modelo implementado em Java permite que ele seja simulado e estudado em diferentes plataformas, inclusive com o uso de diferentes métodos numéricos para a resolução das equações diferenciais.

Pelo menos no que tange ao modelo de (DAVISON; FENG; BROWN, 2000), ele apresenta resultados semelhantes para os experimentos feitos quando executado em sua versão original em NEURON e quando executado em sua versão em Java implementada neste trabalho. Isso significa que o modelo é robusto com relação a implementação em diferentes plataformas. No entanto, isto não permite dizer que outros modelos o sejam também e novos estudos teriam que ser feitos para cada caso particular.

Os cálculos da simulação do modelo de neurônio feitos neste trabalho se mostraram rápidos, com a maior parte do tempo gasto sendo dedicada à construção dos gráficos. Isto permite concluir que o modelo de neurônio construído pode ser utilizado em modelos de rede para o bulbo olfativo, compostos por muitas células, implementados em Java.

Referências Bibliográficas

BHALLA, U. S.; BOWER, J. M. Exploring parameter space in detailed singuel neuron models: simulations of the mitral and granule cells of the olfactory bulb. *Journal of Neurophysiology*, v. 69, n. 6, p. 1948–1965, 6 1993.

BOWER, J. M.; BEEMAN, D. *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System*. 2. ed. [S.l.]: Springer-Verlag, 1998.

BURDEN, R. L.; FAIRES, J. D. *Análise Numérica*. 7. ed. [S.l.]: Pioneira Thomson Learning, 2003.

BUTCHER, J. C. *Numerical Methods for Ordinary Differencial Equations*. Chichester, West Sussex PO19 8SQ, England: John Wiley & Sons, Ltd, 2003. ISBN 0-471-96758-0.

CARNEVALE, N. T.; HINES, M. L. *The NEURON Book*. [S.l.]: Cambridge University Press, 2004.

DAVISON, A. P.; FENG, J.; BROWN, D. A reduced compartmental model of the mitral cell for use in network models of the olfactory bulb. *Brain Research Bulletin*, v. 51, n. 5, p. 393–399, 2000.

DAVISON, A. P.; FENG, J.; BROWN, D. Dendrodendritic inhibition and simulated odor response in a detailed olfactory bulb network model. *Journal of Neurophysiology*, v. 90, p. 1921–1935, 4 2003.

FARLOW, S. J. *An Introduction to Diferencial Equations and their Applications*. [S.l.]: Dover, 1994.

GOSLING, J. et al. *The Java Language Specification*. 3. ed. [S.l.]: Addison Wesley Professional, 2005.

HINES, M. L. et al. Modeldb: A database to support computational neuroscience. *J Comput Neurosci*, v. 17, n. 1, p. 7–11, 2004.

HODGKIN, A. L.; HUXLEY, A. F. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, v. 117, p. 500–544, 1952.

KUTTA, W. Beitrag zur näherungsweisen integration totaler differentialgleichungen. *Z. Math. Phys.*, v. 46, p. 435–453, 1901.

LAMBERT, J. D. *Numerical Methods for Ordinary Differencial Systems*. [S.l.]: John Wiley & Sons, Ltd, 1991.

PRESS, W. H. et al. *Numerical Recipes: The Art of Scientific Computing*. 2nd. ed. [S.l.]: Cambridge University Press, 1992.

RALL, W. Branching dendritic trees and motoneuron membrane resistivity. *Exp. Neurol.*, v. 1, p. 491–527, 1959.

RUGGIERO, M. A. G.; LOPES, V. L. R. *Cálculo Numérico: aspectos teóricos e computacionais*. 2. ed. [S.l.]: Makron Books, 2004.

RUNGE, C. Über die numerische auflösung von differentialgleichungen. *Math. Ann.*, v. 44, p. 167–178, 1895.