PGF5210 - Simulação de Processos Físicos com Método de Monte Carlo

Método de Monte Carlo com Cadeia de Markov Aplicado ao Estudo das Interações entre Neurônios a partir de Dados Experimentais

Pedro Brandimarte Mendonça

Orientador: Prof. Dr. Jefferson Antonio Galves

Instituto de Física da Universidade de São Paulo

18 de junho de 2012

Resumo

O objetivo deste trabalho é estimar, via método de Monte Carlo com Cadeia de Markov, o grafo que melhor representa a interação entre neurônios a partir de dados de registros individuais de neurônios obtidos experimentalmente. Estes dados provêm de uma experiência conduzida pelo Prof. Dr. Sidarta T. G. Ribeiro no laboratório do Prof. Dr. Miguel A. L. Nicolelis na *Neurobiology School of Medicine - Duke University*, onde foram implantados microeletrodos em regiões específicas do cérebro de ratos [1]. Considerando um modelo probabilístico simples, foi implementado, em linguagem C, um algoritmo eficiente para o Monte Carlo com cadeia de Markov no espaço dos grafos.

Agradecimentos

Gostaria de agradecer, primeiramente, ao Prof. Dr. Jefferson Antonio Galves por me receber em seu grupo de pesquisa e me apresentar o problema estudado neste trabalho. Aos alunos de doutoramento Aline Duarte de Oliveira e Guilherme Ost de Aguiar, que estão se dedicando ao estudo deste problema e que, pacientemente, me explicaram o problema em si, assim como seus conceitos teóricos. Por fim, gostaria de agradecer também o Prof. Dr. Yoshiharu Kohayakawa pelas idéias e dicas fundamentais para a implementação do método.

Índice

Lista de Figuras

Capítulo 1

Introdução

O objetivo deste trabalho é estimar, via método de Monte Carlo com Cadeia de Markov, o grafo que melhor representa a interação entre neurônios observados por meio da implantação de microeletrodos em regiões específicas do cérebro de um rato.

Neste capítulo serão descritos, em um nível muito básico, como ocorrem as interações entre neurônios[1] e o experimento que deu origem aos dados utilizados [1].

1.1 Transmissão Sináptica

O neurônio é uma célula nervosa que tipicamente possui quatro regiões morfologicamente definidas: corpo celular, dendritos, axônio e terminais pré-sinápticos (`Figura 1.1`). O corpo celular é o centro metabólico da célula, contendo o núcleo (com os genes da célula) e o retículo endoplasmático (extensão do núcleo onde as proteínas da célula são sintetizadas).

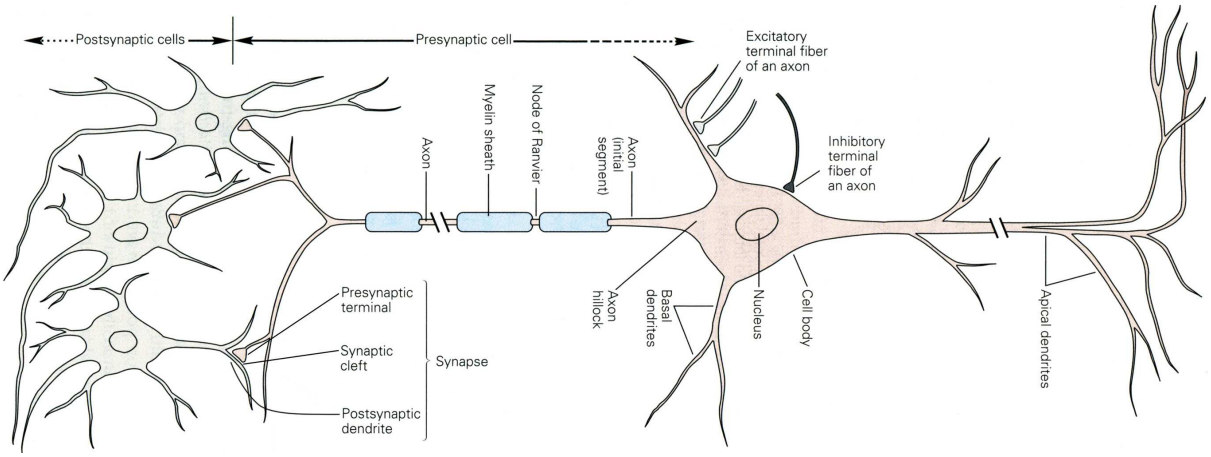


Figura 1.1: *Representação esquemática da estrutura de um neurônio.* (adaptado de [2])

[1] As informações sobre as interações entre neurônios foram extraídas principalmente da referência [2]

Geralmente, a partir do corpo celular existem dois tipos de prolongamentos: vários dendritos curtos e um axônio longo e tubular. Os dendritos constituem o principal instrumento de recepção de sinais provenientes de outras células nervosas. Já o axônio estende-se para longe do corpo celular e é a unidade principal de condução de sinais elétricos para outras células.

Estes sinais elétricos, chamados de potencial de ação, são rápidos e momentâneos, e são impulsos nervosos com a característica tudo-ou-nada (isto é, uma vez que o sinal começa a se propagar ele é regenerado em intervalos regulares ao longo do axônio, fazendo com que a amplitude do sinal permaneça constante). Os potenciais de ação possuem tipicamente uma amplitude de 100 mV e duração de 1 ms.

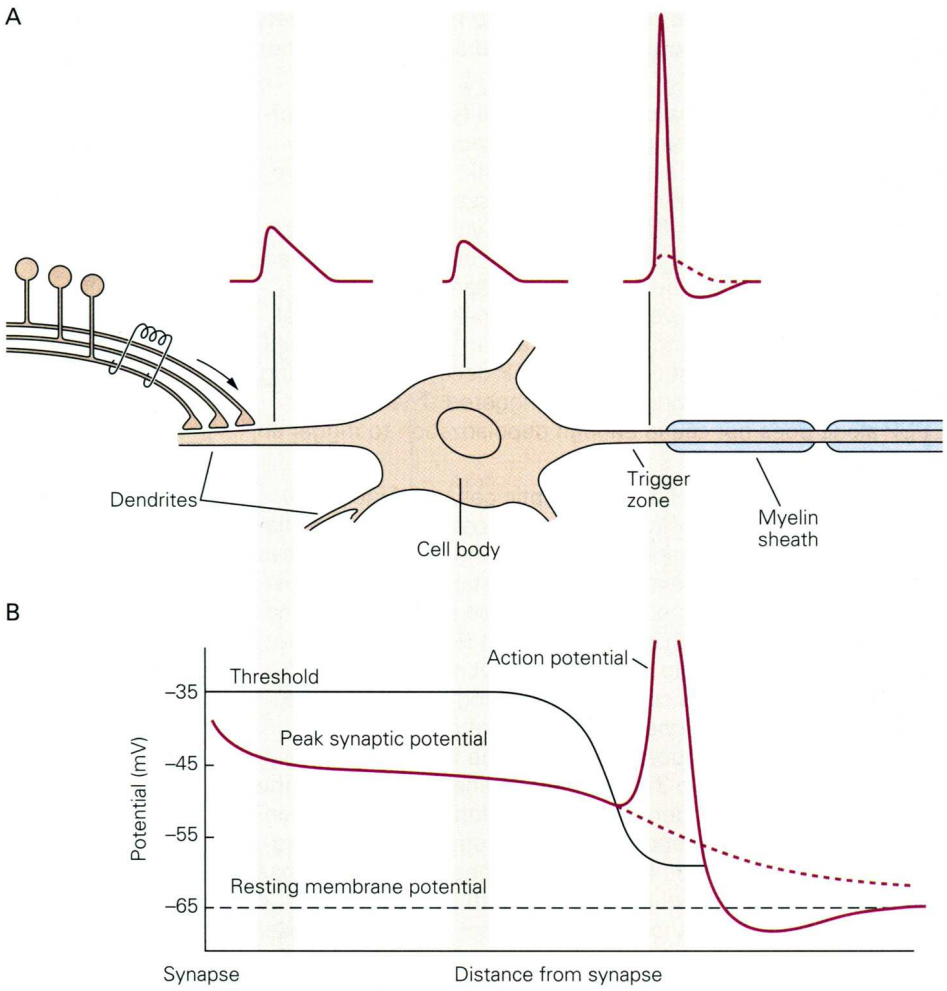


Figura 1.2: A. *Um potencial sináptico proveniente de um dendrito decresce conforme se propaga ao longo da célula, porém um potencial de ação pode ser originado no cone de implantação do axônio (*trigger zone*) devido ao baixo limiar de excitação nesta região.* B. *Comparação entre o potencial sináptico e o limiar para iniciar um potencial de ação nas partes do neurônio correspondentes ao desenho* A, *onde se nota que o potencial de ação é gerado quando a amplitude do potencial sináptico ultrapassa o limiar.* (extraído de [2])

Os potenciais de ação constituem os sinais pelo qual o cérebro recebe, analisa e transmite informações. Apesar de serem gerados por uma vasta gama de eventos em que o corpo humano é exposto (de luz até contato mecânico, de odores até ondas de pressão etc.), estes sinais são altamente padronizados. De fato, a informação transmitida por um potencial de ação é determinada não pelo sinal, mas, sim, pelo caminho que ele percorre no cérebro.

Quase no fim de sua extensão, o axônio se divide em ramos finos que formam locais de comunicação com outros neurônios. O ponto onde dois neurônios se comunicam é chamado sinapse e os ramos do axônio (da célula que transmite informação) são chamados de terminais pré-sinápticos. Na `Figura 1.2` pode se observar esquematicamente como um potencial sináptico de excitação proveniente de um dendrito pode gerar um potencial de ação que se propaga ao longo do axônio.

1.2 Experimento

Os dados experimentais utilizados provêm de uma experiência conduzida pelo Prof. Dr. Sidarta T. G. Ribeiro no laboratório do Prof. Dr. Miguel A. L. Nicolelis na *Neurobiology School of Medicine - Duke University* [1], onde se pretendia testar a teoria de que a ação combinada das fases do sono SWS (*slow-wave sleep*) e REM (*rapid eye movement*) é responsável por propagar mudanças sinápticas recentes do hipocampo ao córtex.

Grosso modo, o cérebro é constituído por uma camada externa altamente enrugada, chamada córtex (`Figura 1.3`), e por outras três estruturas internas: os gânglios da base, o núcleo amigdalóide e o hipocampo. Acredita-se que o armazenamento de memórias envolve a região do hipocampo a curto prazo e que, com o tempo, essas memórias migram para o córtex.

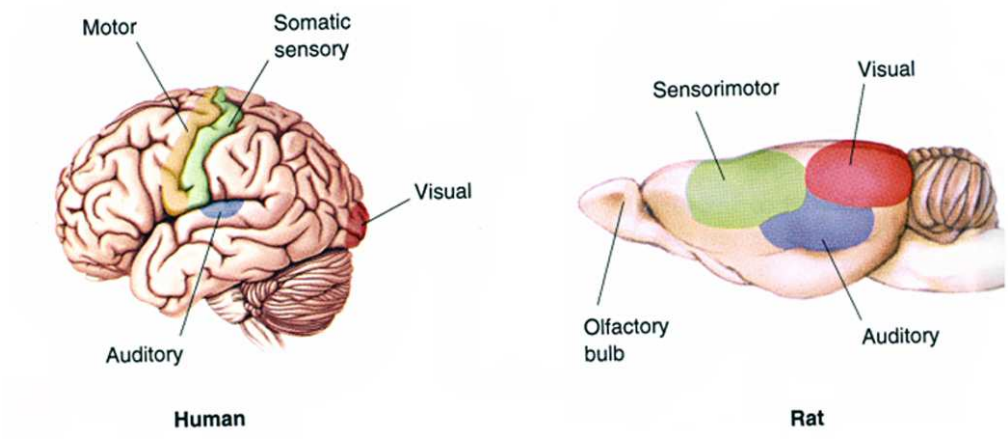


Figura 1.3: *Vista lateral do córtex cerebral de um humano (esquerda) e de um rato (direita) com as áreas sensoriais primárias identificadas.* (adaptado de [3])

Para avaliar o papel da ação combinada das fases do sono SWS e REM na consolidação da memória, observou-se a atividade neuronal de ratos expostos a estímulos espaciais e táteis. A experiência foi composta por três partes (`Figura 1.4`). Inicialmente, o rato é colocado em uma caixa vazia onde permanece durante aproximadamente 2 horas, de forma a que o animal fique bem habituado ao ambiente. Em seguida, a fim de produzir os estímulos sensoriais, quatro objetos são inseridos na caixa: comida dentro de um segmento de tubo de PVC, uma escova de fios de textura similar a cabelo, uma bola de golfe presa a uma mola e uma haste de madeira com pinos metálicos pontiagudos. O rato permanece em contato com os objetos por 20 minutos e, por fim, permanece novamente na caixa vazia por aproximadamente 3 horas. Toda a experiência ocorre sob iluminação no infravermelho para evitar estímulos no córtex visual primário (V1).

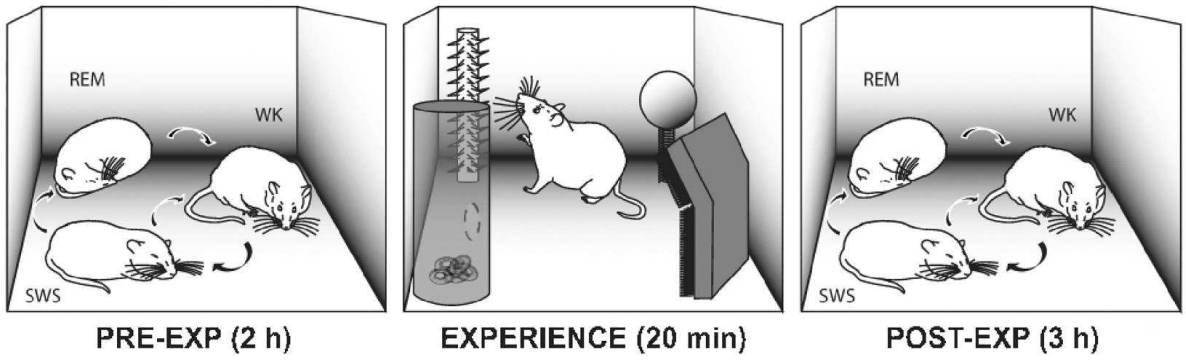


Figura 1.4: *Representação do experimento.* (extraído de [1])

Para registrar os disparos individuais de neurônios, foram implantados cirurgicamente três malhas de multieletrodos, constituídos por 16-32 microfios de tungstênio revestidos por Teflon.

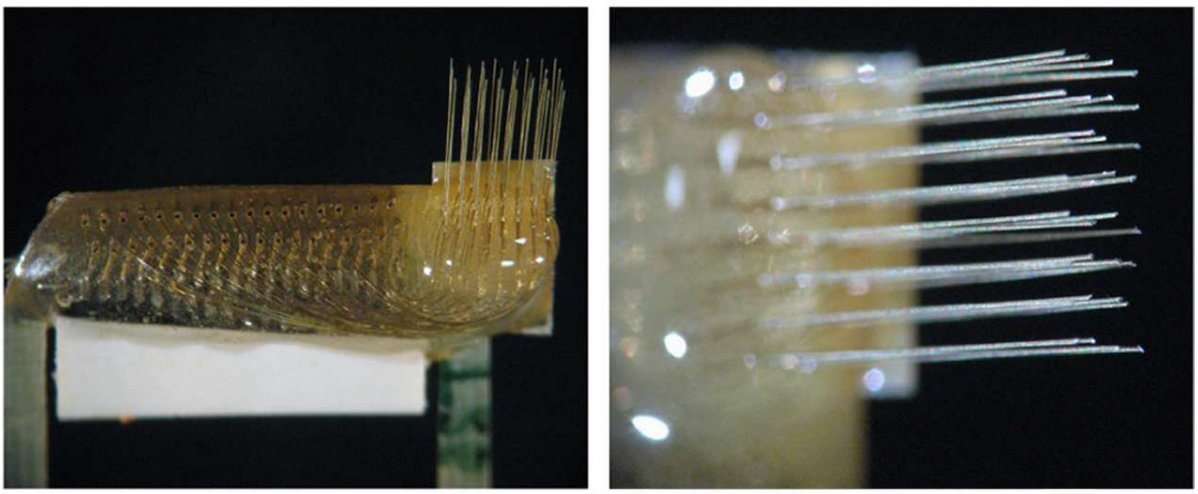


Figura 1.5: *Microeletrodo com 32 canais.* (adaptado de [4])

As malhas foram implantadas visando três regiões específicas do cérebro: o hipocampo (HP), o córtex somatossensorial primário (S1) e o córtex visual primário (V1). Em alguns casos, foi ainda possível distinguir eletrodos inseridos na região CA1 e na região dentada (DG) do hipocampo (`Figura 1.6`).

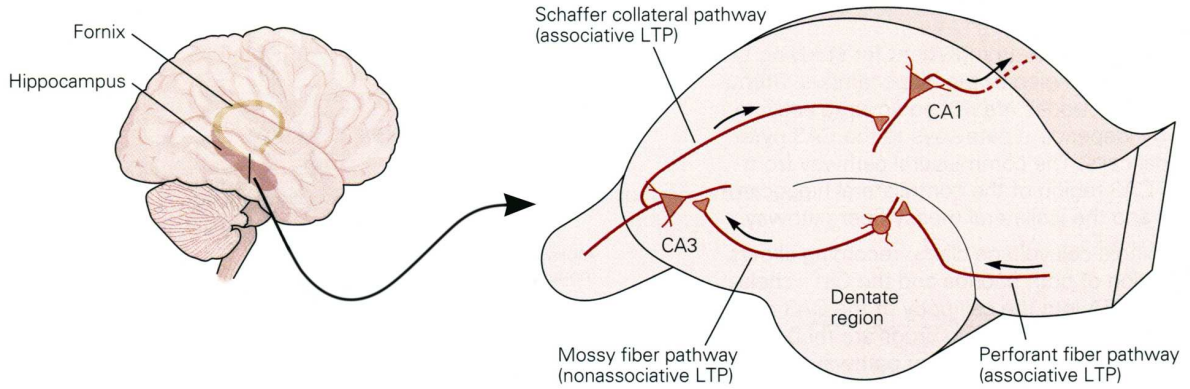


Figura 1.6: *Representação do caminho percorrido por impulsos nervosos no hipocampo.* (adaptado de [2])

O sistema de aquisição de dados para o registro de disparos de neurônios utilizado é capaz de distinguir e isolar sinais de neurônios distintos provenientes de um mesmo eletrodo em tempo real. Na `Figura 1.7` observa-se uma seção de aquisição de dados e na `Figura 1.8` observa-se em maior detalhe o programa de aquisição de dados, onde se pode notar sinais de três neurônios claramente isolados.

Figura 1.7: *Detalhe de um rato com eletrodos implantados (esquerda) e de uma seção de aquisição de dados (direita).* (adaptado de [4])

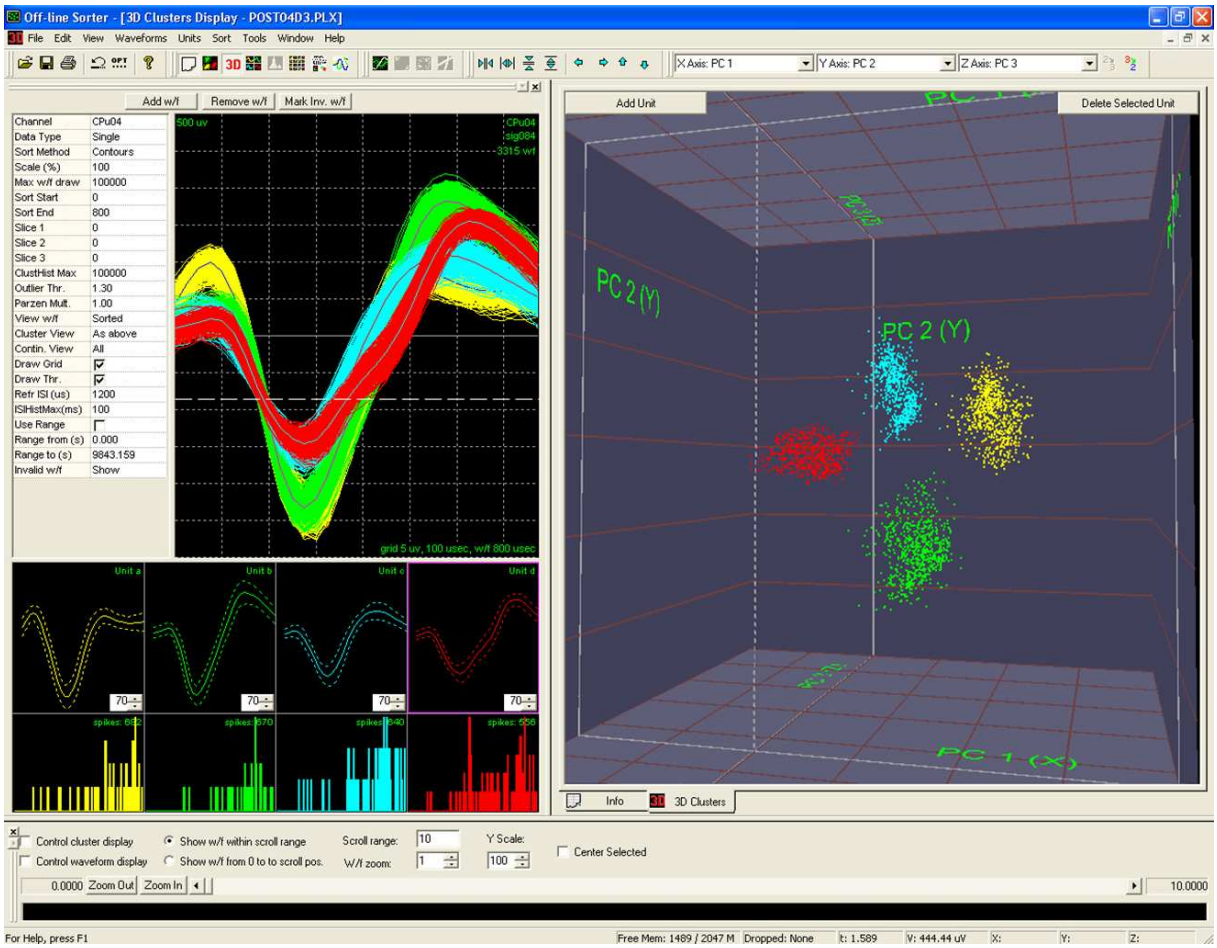


Figura 1.8: *Foto ilustrativa do programa de aquisição de dados utilizado para armazenar e para distinguir em tempo real disparos de neurônios isolados.* (adaptado de [4])

Capítulo 2

Modelo Teórico

Com os dados coletados dos disparos individuais de neurônios, a pergunta que se pretende responder no presente trabalho é: como estimar o grafo que melhor representa a interação entre neurônios de uma região do cérebro observada?

Neste capítulo serão apresentados um modelo probabilístico simples, baseado no modelo de Ising, para descrever tais interações e o método de Monte Carlo com Cadeia de Markov utilizado para obter o estimador de máxima verossimilhança do modelo. Para tanto, serão apresentados inicialmente os conceitos de grafo e de probabilidade a posteriori.

2.1 Grafos

Antes de definir o modelo adotado, considere a definição e algumas propriedades fundamentais sobre grafos.

> **Definição:** *Um* **grafo** *é um conjunto de* **vértices** $\mathcal{V}$ *adicionado a um conjunto de* **arestas** $\mathcal{A}$ *que conectam pares de vértices distintos.* □

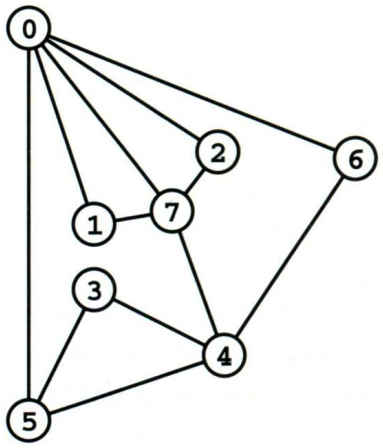
No problema da interação entre neurônios, considera-se cada neurônio como um vértice e a interação como uma aresta. Em particular, serão considerados grafos não-direcionados[1], isto é, a aresta $A(i, j)$ é idêntica à aresta $A(j, i)$. Grafos deste tipo possuem a seguinte propriedade:

> **Propriedade:** *Um grafo com* V *vértices tem no máximo* $V(V-1)/2$ *arestas, isto é,* $|\mathcal{A}| = V(V-1)/2$. □

[1]Serão considerados grafos *simples*, isto é, grafos que não contêm arestas paralelas (*multigraphs*) e arestas que conectam um vértice a si mesmo (*self-loops*).

PROVA: O total de V^2 possíveis pares de vértices inclui V arestas que conectam um vértice a si mesmo e conta duas vezes cada aresta entre vértices distintos. Logo, o número de arestas é no máximo $(V^2 - V)/2 = V(V-1)/2$. ■

Uma maneira simples de representar um grafo é por meio de uma matriz de 0's e 1's, conhecida com matriz de adjacências, onde o elemento $[i, j] = 0$ se não houver uma aresta conectando-os ou $[i, j] = 1$, caso contrário (`Figura 2.1`).



	0	1	2	3	4	5	6	7
0	0	1	1	0	0	1	1	1
1	1	0	0	0	0	0	0	1
2	1	0	0	0	0	0	0	1
3	0	0	1	0	1	0	0	0
4	0	0	0	1	0	1	1	0
5	1	0	0	1	1	0	0	0
6	1	0	0	0	1	0	0	0
7	1	1	1	0	1	0	0	0

Figura 2.1: *Exemplo de um grafo não-direcionado (esquerda) e respectiva representação como matriz de adjacências (direita).* (adaptado de [5])

2.2 Probabilidade a Posteriori

Neste trabalho, pretende-se calcular a probabilidade de um grafo dado os registros de disparos. Logo, antes de definir o modelo, é útil definir também o conceito de probabilidade a posteriori.

Um modelo estatístico qualquer pode ser representado por $p(X|\theta)$, onde X é uma variável aleatória e θ é um vetor de parâmetros (e.g. $X \sim \mathcal{N}(0,1)$ e $\theta = [\mu, \sigma^2]$). Então, para um parâmetro θ fixo, $p(X|\theta)$ é a distribuição amostral, logo:

$$\int_X p(X|\theta)\, dX = 1 \tag{2.1}$$

No entanto, dado que $X = x_1$ foi observado, pode-se calcular a probabilidade de θ dado $X = x_1$:

$$L(\theta|X = x_1) = p(X = x_1|\theta) \tag{2.2}$$

Esta medida de crença em θ é chamada de função de verossimilhança. No caso em que foi observado $X = x_1$, a distribuição conjunta $p(X, \theta)$ (com X fixo e θ variando), pode ser escrita como:

$$p(X, \theta) = p(X = x_1|\theta)\, p_o(\theta) = p(\theta|X = x_1)\, p(X = x_1) \tag{2.3}$$

onde $p_o(\theta)$ é a informação inicial que se conhece a respeito de θ (*a priori*) e $p(X = x_1)$ é a distribuição preditiva, isto é:

$$p(X = x_1) = \int_\theta p(X = x_1|\theta)\, p_o(\theta)\, d\theta = C_1 \tag{2.4}$$

Portanto, após uma observação $X = x_1$, a informação *a posteriori* a respeito de θ será:

$$\begin{aligned} p_1(\theta) &= p(\theta|X = x_1) = \frac{p(X = x_1|\theta)\, p_o(\theta)}{p(X = x_1)} = \\ &= \frac{p(X = x_1|\theta)\, p_o(\theta)}{\int_\theta p(X = x_1|\theta)\, p_o(\theta)\, d\theta} = \frac{p(X = x_1|\theta)\, p_o(\theta)}{C_1} \end{aligned} \tag{2.5}$$

Ou seja, $p_1(\theta) \propto p(X = x_1|\theta)\, p_o(\theta)$, o que vem a ser uma forma do Teorema de Bayes. Se fizermos n observações x_1, x_2, ..., x_n, então:

$$p_n(\theta) = \frac{p(X = x_n|\theta)\, p_{n-1}(\theta)}{C_{n-1}} = \left[\prod_{i=1}^{n} \frac{p(X = x_i|\theta)}{C_i}\right] p_o(\theta) \tag{2.6}$$

Em termos da função de verossimilhança:

$$p_n(\theta) = \left[\prod_{i=1}^{n} \frac{L(\theta|X = x_i)}{C_i}\right] p_o(\theta) \tag{2.7}$$

Quando não se conhece nada a respeito do parâmetro θ escolhe-se a priori como $p_o(\theta) = 1$, o que representa o desconhecimento, e depois renormaliza-se a posteriori.

2.3 Modelo Teórico

O modelo escolhido é baseado no modelo de Ising bidimensional de interação entre spins de sítios vizinhos em uma rede e onde a distribuição de probabilidade do espaço de configurações $\mathcal{Z}$ é dada pela distribuição de Gibbs:

$$\mathbb{P}(\gamma) = \frac{e^{-\beta\mathcal{H}(\gamma)}}{\sum_{\gamma' \in \mathcal{Z}} e^{-\beta\mathcal{H}(\gamma')}} \tag{2.8}$$

onde γ é uma configuração possível da rede, $\beta = (k_B T)^{-1}$ e $\mathcal{H} = K + V$ é a Hamiltoniana. Classicamente, a parte cinética K depende do momento $\vec{p}$ e possui distribuição normal (sendo, portanto, fácil calculá-la). Já a energia potencial devido à interação entre sítios vizinhos e a um campo magnético aplicado é dada por:

$$V(\gamma) = -\sum_{(i,j)} J_{ij}\sigma_i\sigma_j - H\mu\sum_i \sigma_i, \tag{2.9}$$

onde $\sum_{(i,j)}$ representa a soma sobre todos os vizinhos, $\sigma = \pm 1$ é a projeção do momento de dipolo (spin), H é o campo magnético e J_{ij} é a energia de interação entre dois sítios i e j.

No caso da interação entre neurônios, considere o conjunto $\mathcal{G}$ de todos os grafos possíveis com vértices em $\mathcal{V}$ e arestas em $\mathcal{A}$. Considere, também, uma amostra $\mathbf{X} = (\mathbf{X}^{(1)}, \mathbf{X}^{(2)}, ..., \mathbf{X}^{(N)})$ dos registros de disparos de N neurônios, onde $\mathbf{X}^{(i)} = (x_0^{(i)}, x_1^{(i)}, ..., x_n^{(i)})$ são os registros de disparos do i-ésimo neurônio nos instantes $\{0, 1, 2, \ldots, n\}$, isto é, $x_t^{(i)} = 1$ se ocorreu disparo do i-ésimo neurônio no instante t ou $x_t^{(i)} = 0$, caso contrário.

Em alusão ao modelo de Ising, define-se a probabilidade a posteriori de um grafo $g \in \mathcal{G}$, dada uma amostra $\mathbf{X}$, como:

$$\mathbb{P}(g|\mathbf{X}) = \frac{\exp\left[\sum_{(i,j)\in g} J_{ij}\left(\mathbf{X}^{(i)} \otimes \mathbf{X}^{(j)}\right)\right]}{\sum_{g' \in \mathcal{G}} \exp\left[\sum_{(i,j)\in g'} J_{ij}\left(\mathbf{X}^{(i)} \otimes \mathbf{X}^{(j)}\right)\right]} = \frac{\prod_{(i,j)\in g} e^{J_{ij}\left(\mathbf{X}^{(i)} \otimes \mathbf{X}^{(j)}\right)}}{\sum_{g' \in \mathcal{G}} \prod_{(i,j)\in g'} e^{J_{ij}\left(\mathbf{X}^{(i)} \otimes \mathbf{X}^{(j)}\right)}} \tag{2.10}$$

onde $J_{ij} \in \mathbb{R}$ representa o grau de interação entre os neurônios i e j. A operação produto $\mathbf{X}^{(i)} \otimes \mathbf{X}^{(j)} = \sum_{t=0}^{n} x_t^{(i)} \cdot x_t^{(j)}$ define o método utilizado para o cálculo da probabilidade a posteriori:

$$\textbf{Método 1:} \qquad x_t^{(i)} \cdot x_t^{(j)} = \begin{cases} 1 & \text{se } x_t^{(i)} = 1 \text{ e } x_t^{(j)} = 1 \\ 0 & \text{caso contrário} \end{cases} \tag{2.11}$$

$$\textbf{Método 2:} \qquad x_t^{(i)} \cdot x_t^{(j)} = \begin{cases} 1 & \text{se } x_t^{(i)} = 1 \text{ e } x_t^{(j)} = 1 \text{ ou } x_t^{(i)} = 0 \text{ e } x_t^{(j)} = 0 \\ 0 & \text{caso contrário} \end{cases} \tag{2.12}$$

$$\textbf{Método 3:} \qquad x_t^{(i)} \cdot x_t^{(j)} = \begin{cases} 1 & \text{se } x_t^{(i)} = 1 \text{ e } x_t^{(j)} = 1 \text{ ou } x_t^{(i)} = 0 \text{ e } x_t^{(j)} = 0 \\ -1 & \text{caso contrário} \end{cases} \tag{2.13}$$

Pode-se mostrar que o método 2 é de certa forma equivalente ao método 3, pois:

$$\left(\mathbf{X}^{(i)} \otimes \mathbf{X}^{(j)}\right)_{met2} = \sum_{t=0}^{n} \mathbb{1}_{\left\{x_t^{(i)} = x_t^{(j)}\right\}} = m \in [0, n] \tag{2.14}$$

$$\left(\mathbf{X}^{(i)} \otimes \mathbf{X}^{(j)}\right)_{met3} = \sum_{t=0}^{n} \mathbb{1}_{\left\{x_t^{(i)} = x_t^{(j)}\right\}} - \sum_{t=0}^{n} \mathbb{1}_{\left\{x_t^{(i)} \neq x_t^{(j)}\right\}} = 2m - (n+1) \tag{2.15}$$

Nota-se que, como a probabilidade a posteriori é dada pelo produto de uma exponencial para cada aresta $(i, j) \in g$, este modelo sempre dará preferência a grafos com maior quantidade de arestas, isto é, o grafo com todas as arestas é o mais provável. Logo, introduz-se no modelo uma medida de penalização relacionada à quantidade de arestas consideradas:

$$\pi(g) = \frac{\exp\left[-cn \sum_{a \in \mathcal{A}} \mathbb{1}_{\{a \in g\}}\right]}{\sum_{g' \in \mathcal{G}} \exp\left[-cn \sum_{a \in \mathcal{A}} \mathbb{1}_{\{a \in g'\}}\right]} \tag{2.16}$$

onde $c \in [\,0, 1\,]$ é uma constante que indica o custo de se considerar uma aresta a mais no modelo. Desta forma, a probabilidade a posteriori de um grafo $g \in \mathcal{G}$, dada uma amostra $\mathbf{X}$, fica:

$$\mathbb{P}(g|\mathbf{X}) = \pi(g) \frac{\prod_{(i,j) \in g} e^{J_{ij}\left(\mathbf{X}^{(i)} \otimes \mathbf{X}^{(j)}\right)}}{\sum_{g' \in \mathcal{G}} \prod_{(i,j) \in g'} e^{J_{ij}\left(\mathbf{X}^{(i)} \otimes \mathbf{X}^{(j)}\right)}} = \frac{\prod_{(i,j) \in g} e^{J_{ij}\left(\mathbf{X}^{(i)} \otimes \mathbf{X}^{(j)}\right) - cn}}{\sum_{g' \in \mathcal{G}} \prod_{(i,j) \in g'} e^{J_{ij}\left(\mathbf{X}^{(i)} \otimes \mathbf{X}^{(j)}\right) - cn}} \tag{2.17}$$

Apesar de parecer artificial num primeiro momento, o método de penalização estabelece um critério para a quantidade de parâmetros considerados e é utilizado em diversos modelos estatísticos envolvendo otimização de estimadores. Em muitos casos é possível encontrar empiricamente a constante ideal e em alguns consegue-se provar o motivo de sua existência (veja um exemplo para o caso de cadeias de Markov de alcance variável em [6]).

2.4 Método de Monte Carlo com Cadeia de Markov - MCMC

Para estimar, com o modelo apresentado, o grafo que melhor representa a interação entre os neurônios observados bastaria, em princípio, calcular a probabilidade a posteriori para todo grafo $g \in \mathcal{G}$. No entanto, a quantidade de grafos possíveis inviabiliza este procedimento. Por exemplo, se considerarmos um conjunto com apenas 20 neurônios, a quantidade de grafos possíveis é dada por:

$$2^{\binom{20}{2}} = 2^{190} \sim O\left(10^{57}\right) \tag{2.18}$$

Outra maneira de estimar o grafo mais representativo é construir, sob o conjunto dos grafos $\mathcal{G}$, uma cadeia de Markov $(Y_m)_{m\geq 1}$ irredutível e aperiódica, cuja distribuição limite seja dada por $\mathbb{P}(g|\mathbf{X})$. Ou seja, a cadeia deve ser construída de forma a que, para um número suficientemente grande de interações, a cadeia convirja para esta distribuição (Teorema Ergódico):

$$\lim_{m\to\infty} \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}_{\{Y_i=g\}} = \mathbb{P}(g|\mathbf{X}) \tag{2.19}$$

Desta forma, para encontrar o grafo g que maximiza a probabilidade a posteriori $\mathbb{P}(g|\mathbf{X})$, basta escolher, para m suficiente grande:

$$g^* = \arg\max_g \left\{ \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}_{\{Y_i=g\}} \right\} \tag{2.20}$$

A construção de uma cadeia de Markov com as características descritas acima pode ser realizada por meio do método de Monte Carlo com algoritmo de Metropolis-Hastings. Começando de um estado inicial g_i, o algoritmo procede da seguinte forma:

① Um candidato a próximo estado g_j é proposto com probabilidade de transição Q_i^j.

② O estado g_j é aceito com probabilidade de aceitação $\alpha\left(g_i, g_j\right)$.

③ Caso contrário, o estado g_j é rejeitado e a cadeia permanece no estado g_i.

④ Retorne ao passo 1.

Para obter-se uma cadeia de Markov com distribuição limite $\mathbb{P}(g|\mathbf{X})$, a probabilidade de aceitação $\alpha\left(g_i, g_j\right)$ deve ser escolhida de forma a obedecer a equação de balanço detalhado:

$$\mathbb{P}(g_i|\mathbf{X})Q_i^j\alpha\left(g_i, g_j\right) = \mathbb{P}(g_j|\mathbf{X})Q_j^i\alpha\left(g_j, g_i\right) \tag{2.21}$$

Uma probabilidade de aceitação que obedece a equação acima é a proposta por Metropolis-Hastings:

$$\alpha\left(g_i, g_j\right) = \min\left(1, \frac{\mathbb{P}(g_j|\mathbf{X})Q_j^i}{\mathbb{P}(g_i|\mathbf{X})Q_i^j}\right) \tag{2.22}$$

Se o núcleo de amostragem utilizado for simétrico ($Q_i^j = Q_j^i$), a probabilidade de aceitação simplifica a:

$$\alpha\left(g_i, g_j\right) = \min\left(1, \frac{\mathbb{P}(g_j|\mathbf{X})}{\mathbb{P}(g_i|\mathbf{X})}\right) \tag{2.23}$$

Isto indica que, caso o novo estado g_j seja mais favorável ($\mathbb{P}(g_j|\mathbf{X}) \geq \mathbb{P}(g_i|\mathbf{X})$), ele é aceito com probabilidade $\alpha\left(g_i, g_j\right) = 1$, caso contrário, ele pode ser aceito com probabilidade dada pela razão $\mathbb{P}(g_j|\mathbf{X})/\mathbb{P}(g_i|\mathbf{X})$.

Neste trabalho, foi utilizado como núcleo de amostragem simétrico dado pela distribuição uniforme em $\Big[0, |\mathcal{A}|\Big]$, onde $|\mathcal{A}|$ é o cardinal do conjunto de todas as arestas. Considerou-se que o candidato a próximo estado g_j pode possuir somente uma aresta diferente quando comparado ao estado atual a g_i, isto é:

$$d\left(g_i, g_j\right) = \sum_{a \in \mathcal{A}} \left| \mathbb{1}_{\{a \in g_i\}} - \mathbb{1}_{\{a \in g_j\}} \right| = 1 \tag{2.24}$$

Portanto, para gerar um novo candidato g_j sorteia-se uma aresta $b \in \mathcal{A}$ com probabilidade $1/|\mathcal{A}|$. Caso o grafo atual g_i não contenha a aresta b, o candidato a novo estado g_j é gerado a partir do estado atual g_i incluindo-se a aresta b. Caso o grafo atual g_i contenha a aresta b, o candidato a novo estado g_j é gerado a partir do estado atual g_i retirando-se a aresta b. Desta forma, a probabilidade de transição de um grafo g_i para um grafo g_j será dada por:

$$p(g_j|g_i) = \frac{1}{|\mathcal{A}|} \min\left(1, \frac{\mathbb{P}(g_j|\mathbf{X})}{\mathbb{P}(g_i|\mathbf{X})}\right) + \mathbb{1}_{\{g_j = g_i\}} \left(1 - \frac{1}{|\mathcal{A}|} \sum_{g_k \in \mathcal{G}} \min\left(1, \frac{\mathbb{P}(g_k|\mathbf{X})}{\mathbb{P}(g_i|\mathbf{X})}\right)\right) \tag{2.25}$$

Capítulo 3

Resultados

O algoritmo de Monte Carlo com cadeia de Markov (MCMC) para o modelo discutido no Capítulo 2 foi implementado em linguagem C (Apêndice A). Neste capítulo serão apresentados detalhes da implementação, assim como os resultados obtidos.

3.1 Implementação

3.1.1 Representação do Grafo

Conforme discutido na Seção 2.1, a interação entre neurônios de um dado conjunto observado pode ser representada por um grafo não-direcionado. Em particular, o grafo pode ser representado por uma matriz de adjacência (veja Figura 2.1) que, no caso de grafos simples, é simétrica.

Portanto, durante o algoritmo MCMC, ao invés de armazenar uma matriz de adjacências para cada novo grafo gerado, basta armazenar um vetor contendo os elementos abaixo (ou acima) da diagonal principal. O tamanho desse vetor é igual à quantidade de arestas possíveis ($|\mathcal{A}|$). Por exemplo, para representar um grafo de um conjunto com 20 neurônios, ao invés de utilizarmos uma matriz com 400 elementos basta, um vetor com 190 elementos.

Contudo, uma vez adotada uma regra para a maneira como os elementos são indexados no vetor, ela deve ser coerente em todo o programa. Nesta implementação, escolheu-se indexar por linhas (*row-major order*) os elementos abaixo da diagonal principal da matriz de adjacências. Desta forma, o elemento $[\,i, j\,]$ da matriz de adjacências pode ser recuperado da seguinte forma[1]:

[1]Em linguagem C o primeiro elemento de um vetor é indexado por 0. Em outras linguagens onde o primeiro elemento de um vetor possui índice 1, como Fortran, a Equação 3.1 tem que ser adaptada.

$$[\,i,j\,] \longrightarrow \begin{cases} \left[i+1+(j+1)\frac{j-2}{2}\right] & \text{se } i < j \\ \\ \left[j+1+(i+1)\frac{i-2}{2}\right] & \text{caso contrário} \end{cases} \tag{3.1}$$

3.1.2 *Skip List*

Um vez que o grafo é representado por um vetor de 0's e 1's, ele pode ser considerado como uma *string* (um vetor de caracteres). Desta forma é possível estabelecer uma relação de ordem entre os grafos gerados, no caso a ordem lexicográfica. Este fato viabiliza a criação de uma tabela de símbolos para o armazenamento dos grafos gerados.

> **Definição:** *Uma* **tabela de símbolos**, *ou dicionário, é uma estrutura de dados de itens com chaves que suporta duas operações básicas: inserir um novo item e retornar um item com uma dada chave.* □

Por exemplo, no dicionário da língua portuguesa, as "chaves" são as palavras e os "itens" são as entradas associadas às palavras que contêm a definição, a pronúncia etc.

Neste trabalho foi implementada uma estrutura abstrata de dados para uma tabela de símbolos conhecida como *skip list*. Essa estrutura, criada por William Pugh, foi apresentada pela primeira vez em 1990 no trabalho *Skip Lists: A Probabilistic Alternative to Balanced Trees* [7]. Basicamente, ela utiliza apontadores adicionais nas células de uma lista-ligada a fim de viabilizar "saltos" por grandes porções da lista.

> **Definição:** *Uma* ***skip list*** *é uma lista-ligada ordenada, onde cada célula contém um número variado de apontadores, com os i-ésimos apontadores nas células formando uma única lista-ligada, que salta sobre sobre as células que possuem menos do que i apontadores.* □

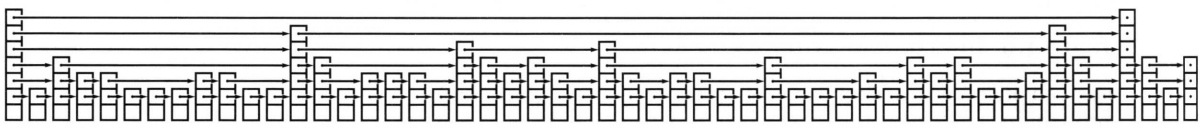


Figura 3.1: *Representação de uma* skip list *após a inserção de 50 elementos.* (extraído de [5])

Na inserção de um novo elemento na *skip list*, sorteia-se a quantidade j de apontadores da nova célula com probabilidade proporcional a $1/t^j$, onde t é um parâmetro fixado (em geral $t = 2$). A idéia é que, para uma *skip list* com N elementos, os apontadores do maior nível

executem saltos proporcionais a $N/2$. Esta característica faz com que os algoritmos de busca e de inserção exijam $O(\log N)$ comparações.

Na `Figura 3.2`, observa-se o "caminho" percorrido na *skip list* ao realizar uma busca pela chave "L". A busca inicia-se em uma célula "cabeça", que possui a quantidade máxima de apontadores na lista (neste caso 4), a partir do nível mais elevado, descendo de nível toda vez que for encontrada uma chave maior do que a que está sendo buscada.

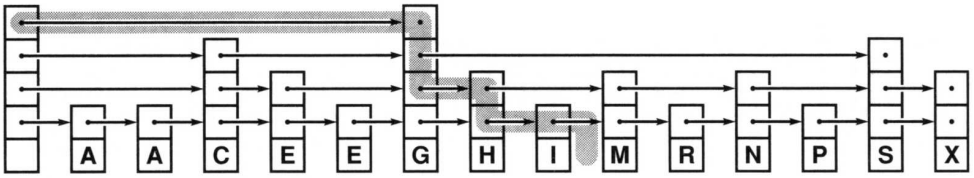


Figura 3.2: *Representação da execução do algoritmo de busca para a chave "L" em uma* skip list. (extraído de [5])

O algoritmo de inserção procede de forma semelhante à busca, mas inicia-se a partir do nível mais alto da nova célula, corrigindo os apontadores da lista a cada nível. A `Figura 3.2` representa a inserção em uma *skip list* de um item de chave "L" com 3 apontadores.

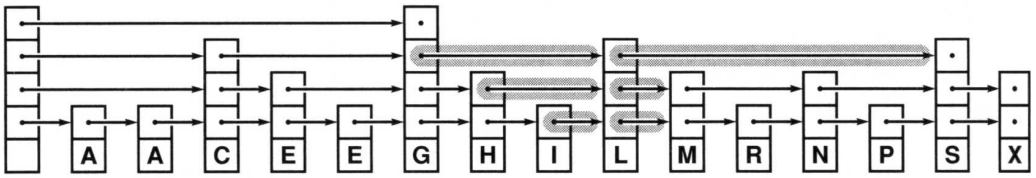


Figura 3.3: *Representação da execução do algoritmo de inserção para um item de chave "L" com 3 apontadores em uma* skip list. (extraído de [5])

A implementação da *skip list* para armazenar os grafos gerados pelo algoritmo MCMC foi baseada na referência [5], com algumas adaptações convenientes ao problema. Cada célula possui uma *string* representando um grafo (chave) e um contador da frequência. Desta forma, a cada grafo gerado no MCMC, realiza-se uma busca na tabela de símbolos e, caso o grafo seja encontrado, incrementa-se o contador de frequência, caso contrário, insere-se o novo grafo na tabela.

Portanto, com esta adaptação da *skip list*, além de obter algoritmos eficientes de busca e inserção, não é necessário armazenar todos os grafos gerados no MCMC, o que gera uma economia drástica de memória.

Além disso, a estrutura possui um apontador para o item de maior frequência. Desta forma,

a resposta de qual é o grafo mais representativo no MCMC é imediata.

3.1.3 Algoritmo de Metropolis-Hastings

Conforme discutido na `Seção 2.4`, no algoritmo MCMC considerou-se que o candidato a próximo estado g_j pode possuir somente uma aresta diferente quando comparado ao estado atual a g_i e, portanto, para gerar o novo candidato g_j sorteia-se uma aresta $b = (u, v) \in \mathcal{A}$ com probabilidade $1/|\mathcal{A}|$. Desta forma, na probabilidade de aceitação, a razão das probabilidades a posteriori simplifica-se a:

$$\alpha\left(g_i, g_j\right) = \begin{cases} \min\left(1, e^{J_{uv}\left(\mathbf{X}^{(u)} \otimes \mathbf{X}^{(v)}\right) - cn}\right) & \text{caso a aresta } b = (u, v) \text{ seja inserida} \\ \\ \min\left(1, e^{-J_{uv}\left(\mathbf{X}^{(u)} \otimes \mathbf{X}^{(v)}\right) + cn}\right) & \text{caso a aresta } b = (u, v) \text{ seja removida} \end{cases} \tag{3.2}$$

Assim, gera-se o novo candidato g_j (isto é, aloca-se uma nova *string*) somente se a mudança da aresta $b = (u, v) \in \mathcal{A}$ for aceita pelo algoritmo de Metropolis-Hastings.

3.1.4 Seleção dos Dados Observados

Os dados experimentais contêm registros dos instantes de disparos de cada neurônio observado. Com uma análise inicial, foi verificado que janelas de tempo de 10 ms correspondem a um limiar para o qual raramente ocorrem mais do que 1 disparo dentro da janela.

Desta forma, o vetor $\mathbf{X}^{(i)} = (x_0^{(i)}, x_1^{(i)}, ..., x_n^{(i)})$ com os registros de disparos do i-ésimo neurônio é obtido considerando janelas de tempo de 10 ms, ou seja, $x_j^{(i)} = 1$ se ocorreu 1 ou mais disparos na j-ésima janela de tempo, caso contrário $x_j^{(i)} = 0$.

Além disso, para garantir a independência entre os dados de um mesmo neurônio, considerou-se somente uma janela de 10 ms a cada 100 janelas. Biologicamente, esta é uma assunção forte, pois, em neurônios da região do hipocampo, podem ocorrer mudanças de longo prazo no limiar de excitação, que podem perdurar por até algumas horas. Porém, este tipo de efeito foi desconsiderado neste trabalho.

3.2 Resultados

O programa implementado executa o MCMC com os registros de disparos do conjunto de neurônios de uma região do cérebro para um dado rato na primeira e terceira parte do experimento (isto é, antes e depois do rato entrar em contato com os objetos geométricos, respectivamente). Ele é divido em duas funções principais. A primeira ("graphPenalty.c") calcula

os grafos mais representativos para diferentes valores da constante de penalização e por meio dos 3 métodos apresentados na `Seção 2.3`. A segunda ("bestGraph.c") calcula o grafo mais representativo para uma constante de penalização e um método especificados (considerando que já foi realizada uma análise de qual deve ser a constante de penalização considerada com os resultados da execução da função "graphPenalty.c").

Serão apresentados aqui alguns resultados obtidos para os neurônios da região do hipocampo do rato 6 na primeira parte do experimento (isto compreende 22 neurônios que foram observados durante $\sim 58\ min$). Na `Figura 3.4` observa-se as probabilidades a posteriori (não normalizadas e sem considerar a penalização) dos grafos mais representativos obtidos para diferentes valores da constante de penalização e com os métodos 1 (esquerda) e 2 (direita), após 5 milhões de passos de Monte Carlo.

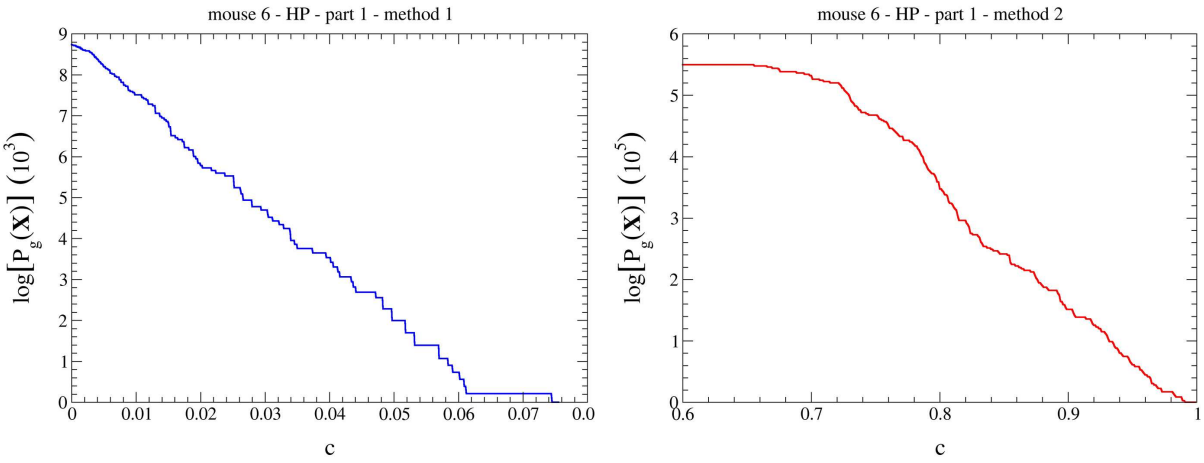


Figura 3.4: *Probabilidades a posteriori (não normalizadas e sem considerar a penalização) dos grafos mais representativos obtidos para diferentes valores da constante de penalização, com os métodos 1 (esquerda) e 2 (direita), após 5 milhões de passos de Monte Carlo.*

Como esperado, em ambos os métodos, nota-se que, para uma penalização suficientemente pequena, a probabilidade a posteriori do grafo mais representativo é máxima e diminui com o aumento da penalização.

Na `Figura 3.5` observa-se as probabilidades empíricas (isto é, a frequência do grafo mais representativo dividido pela quantidade de passos de Monte Carlo realizados) obtidas para diferentes valores da constante de penalização e com os métodos 1 (esquerda) e 2 (direita), após 5 milhões de passos de Monte Carlo.

No método 1, nota-se que quanto menor é a constante de penalização, menor é a probabilidade empírica. Isto indica que muitos grafos foram gerados e que 5 milhões de passos de Monte Carlo são insuficientes para obter uma possível convergência. Já no método 2, observa-se que,

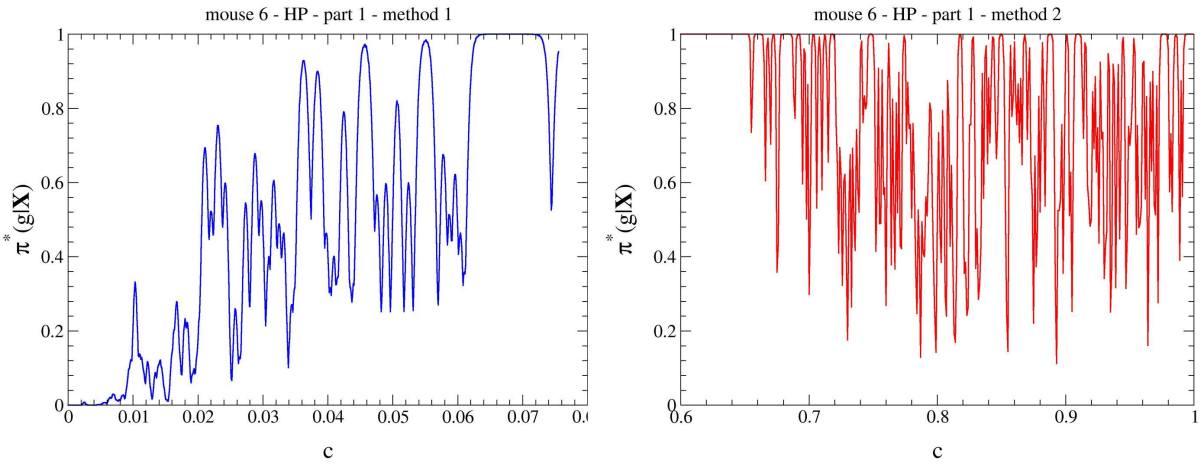


Figura 3.5: *Probabilidades empíricas obtidas para diferentes valores da constante de penalização, com os métodos 1 (esquerda) e 2 (direita), após 5 milhões de passos de Monte Carlo.*

com a mesma quantidade de passos de Monte Carlo, as probabilidades empíricas são relativamente maiores. Isto era esperado, já que o método 2 considera mais informações dos dados experimentais.

Um fato interessante é que cada vale nos gráficos da `Figura 3.5` representam a transição de um grafo g_{c_1} para outro g_{c_2}, que possui exatamente uma aresta a menos. Logo, iniciando com uma penalização suficientemente pequena, o grafo mais representativo contém todas as possíveis arestas e, com o aumento da penalização, as arestas vão sendo retiradas. A ordem com que as arestas são retiradas depende estritamente dos dados experimentais.

Na `Figura 3.6` observa-se os grafos obtidos pelo método 1, após 20 milhões de passos de Monte Carlo, considerando-se 4 penalizações distintas. Novamente, nota-se que, para uma penalização suficientemente pequena, o grafo mais representativo contém todas as possíveis arestas e, com o aumento da penalização, as arestas vão sendo retiradas.

Estes grafos (`Figura 3.6`) indicam, ainda, que os neurônios localizados à esquerda na representação gráfica ($3a$, $3b$, $3c$, $9a$, $10a$ etc.) apresentam maior interação.

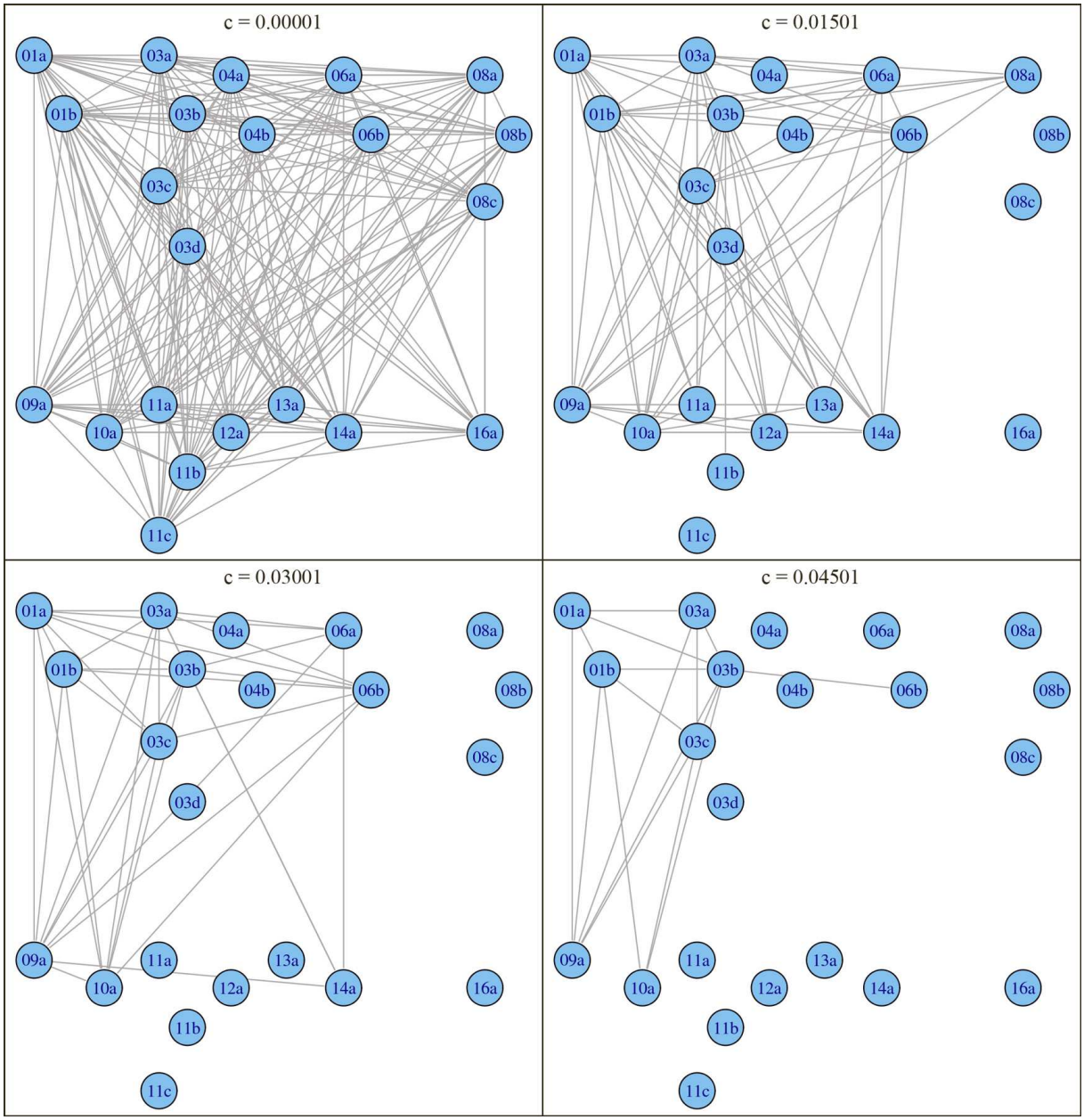


Figura 3.6: *Representação dos grafos obtidos com método 1, após 20 milhões de passos de Monte Carlo, considerando-se 4 penalizações diferentes. (Obs.: esta não é a disposição real dos neurônios observados)*

Na `Figura 3.7` observa-se os grafos obtidos pelo método 2, após 20 milhões de passos de Monte Carlo, considerando-se 4 penalizações distintas. Como esperado, a quantidade de arestas diminui com o aumento da penalização, porém os grafos obtidos indicam que os neurônios localizados à direita na representação gráfica ($16a$, $8b$, $8c$, $9a$, $11c$ etc.) apresentam maior interação. Ou seja, os métodos 1 e 2 levam a conclusões antagônicas.

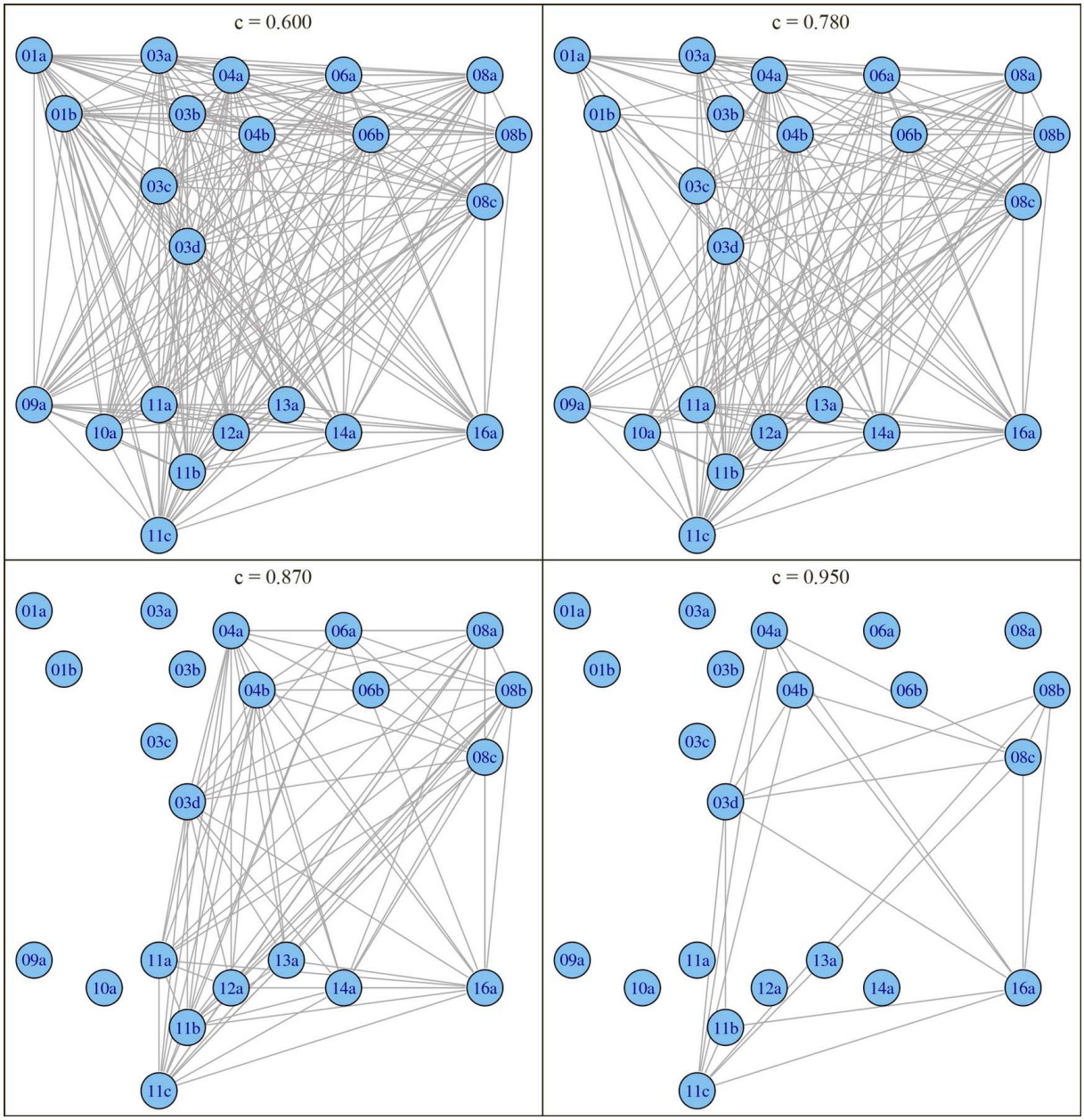


Figura 3.7: *Representação dos grafos obtidos com método 2, após 20 milhões de passos de Monte Carlo, considerando-se 4 penalizações diferentes. (Obs.: esta não é a disposição real dos neurônios observados)*

Na verdade, estes resultados evidenciam um defeito do método 2 (e, consequentemente, do método 3). O fato do método 2 considerar $x_t^{(i)} \cdot x_t^{(j)} = 1$ quando não ocorre disparos no instante t em ambos os neurônios i e j ($x_t^{(i)} = 0$ e $x_t^{(j)} = 0$), faz com que neurônios com baixa atividade sejam considerados conectados.

Contudo, estes resultados indicam que, de forma geral, os neurônios estão mais inativos do que ativos.

3.3 Perspectivas

Os resultados indicaram que o método 1, apesar de considerar menos informações quando comparados aos métodos 2 e 3, é o mais coerente. No entanto, pela `Figura 3.5`, verificou-se que o método 1 necessita uma maior quantidade de passos de Monte Carlo para obter uma possível convergência. Logo, para aumentar a quantidade de passos de Monte Carlo sem ultrapassar o limite físico de memória no computador, seria interessante considerar a implementação de B-árvores[2], com armazenamento em disco.

No caso dos métodos 2 e 3, o peso dado ao caso em que $x_t^{(i)} = x_t^{(j)} = 0$ poderia ser considerado de maneira diferente (inferior) quando comparado ao peso dado no caso em que $x_t^{(i)} = x_t^{(j)} = 1$. Ou ainda, pode ser interessante realizar a análise para cada caso $x_t^{(i)} = x_t^{(j)} = 0$ e $x_t^{(i)} = x_t^{(j)} = 1$ separadamente.

Outra análise interessante seria analisar os instantes de disparos fornecidos, que tem precisão de μs (supondo que esta é a precisão verdadeira das medidas). Considerando que o tempo típico de um potencial de ação é da ordem de 1 ms, poderia-se fazer uma análise da seguinte forma: dado que ocorreu um disparo do neurônio i no instante t, verificar quais neurônios dispararam no instante $t + \delta$ (ou ainda, verificar se ocorrem casos onde, toda vez que um neurônio i dispara, o neurônio j não dispara, indicando um processo inibitório). Isto indicaria uma possível trajetória do sinal. Pode ser que, com esta análise, seja possível definir valores para o grau de interação J_{ij} entre os neurônios.

[2]B-árvore é uma estrutura de dados normalmente utilizada em banco de dados.

Apêndice A

Programa Implementado

A.1 leiaMe.txt

Descrição breve do programa e de seu desenvolvimento.

```
/**  ************************************************  **/
/**     ** Estimativa do Grafo que Melhor Representa **     **/
/**     **        a Interação entre Neurônios        **     **/
/**     **   via Monte Carlo com Cadeias de Markov   **     **/
/**                                                          **/
/**                     **  Versao 7  **                     **/
/**                                                          **/
/**                          IME/USP                         **/
/**                                                          **/
/**   Orientador: Prof. Dr. Antonio Galves                   **/
/**                                                          **/
/**   Co-orientadores: Aline Duarte de Oliveira              **/
/**                    Guilherme Ost de Aguiar               **/
/**                                                          **/
/**   Ajudas e dicas fundamentais dadas pelo Prof. Dr.       **/
/**                                  Yoshiharu Kohayakawa    **/
/**                                                          **/
/**   Autor: Pedro Brandimarte Mendonça                      **/
/**                                                          **/
/**  ************************************************  **/
/**  Breve descrição: (para uma descrição mais detalhada    **/
/**  refira-se ao 'NeuronGraphMCMC.pdf').                   **/
/**                                                          **/
/**  I. Executáveis:                                        **/
/**                                                          **/
/**   I.1. graphPenalty - executa o método de Monte Carlo   **/
/**     com Cadeias de Markov no espaço dos grafos,          **/
/**     representando as conexões entre neurônios, a fim     **/
/**     de estimar, para cada rato, o grafo que melhor       **/
/**     representa os dados observados na primeira e         **/
/**     terceira partes do experimento (isto é, antes e      **/
/**     depois dos ratos entrarem em contato com os          **/
/**     objetos geométricos, respectivamente) considerando   **/
/**     diferentes valores para a constante de penalização   **/
/**     da probabilidade a posteriori e considerando 3       **/
/**     métodos diferentes para o cálculo da probabilidade   **/
/**     a posteriori:                                        **/
/**       1. <Xi|Xj>=1 se Xi=1 e Xj=1                        **/
```

```
/**       2. <Xi|Xj>=1 se Xi=Xj ou <Xi|Xj>=0 (se Xi!=Xj)    **/
/**       3. <Xi|Xj>=1 se Xi=Xj ou <Xi|Xj>=-1 (se Xi!=Xj)   **/
/**     Os arquivos de saída são enviados para a pasta      **/
/**     'out/outPenalty'. Por exemplo:                      **/
/**       - o arquivo 'outputM6HPp1Met1.dat' contém         **/
/**       informações gerais a respeito da execução do      **/
/**       programa para o rato 6 (M6), na região do         **/
/**       hipocampo (HP), na primeira parte do experimento  **/
/**       (p1) e onde a probabilidade a posteriori foi      **/
/**       calculada com o método 1 (Met1).                  **/
/**       - o arquivo 'penal1M6HPp1Met1.dat' contém os      **/
/**       valores da constante de penalização versus o      **/
/**       logaritmo da probabilidade a posteriori não       **/
/**       normalizada (incluindo-se a penalização).         **/
/**       - o arquivo 'penal2M6HPp1Met1.dat' contém os      **/
/**       valores da constante de penalização versus o      **/
/**       logaritmo da probabilidade a posteriori não       **/
/**       normalizada (sem a penalização).                  **/
/**       - o arquivo 'penal3M6HPp1Met1.dat' contém os      **/
/**       valores da constante de penalização versus a      **/
/**       probabilidade empírica obtida pelo método de      **/
/**       Monte Carlo.                                      **/
/**   I.2. bestGraph - para uma dada constante de           **/
/**     penalização, escolhida com base na análise dos      **/
/**     resultados do executável anterior, este programa    **/
/**     executa o método de Monte Carlo com Cadeia de       **/
/**     Markov no espaço dos grafos, representando as       **/
/**     conexões entre neurônios, para estimar o grafo      **/
/**     mais representativo de cada conjunto de neurônios   **/
/**     observados.                                         **/
/**     Os arquivos de saída são enviados para a pasta      **/
/**     'out/outBestGraph'. Por exemplo:                    **/
/**       - o arquivo 'outputHPMet1.dat' contém             **/
/**       informações gerais a respeito da execução do      **/
/**       programa para todos os ratos na região do         **/
/**       hipocampo (HP) onde a probabilidade a posteriori  **/
/**       foi calculada com o método 1 (Met1).              **/
/**       - o arquivo 'adjM6HPp1Met1.dat' contém a matriz   **/
/**       de adjacência do grafo que melhor representa os   **/
/**       neurônios do hipocampo (HP) do rato 6 (M6) na     **/
/**       primeira parte do experimento (p1) e onde a       **/
/**       probabilidade a posteriori foi calculada com o    **/
/**       método 1 (Met1).v                                 **/
/**                                                         **/
/**  II. Scripts:                                           **/
/**                                                         **/
/**   II.1. linux.sh - script para usuários linux para      **/
/**     facilitar a compilação e execução do programa.      **/
/**   II.2. mac.sh - script para usuários mac para          **/
/**     facilitar a compilação e execução do programa.      **/
/**   II.3. jobPen.x - exemplo de script Portable Batch     **/
/**     System (PBS) para executar o programa               **/
/**     'graphPenalty' em um cluster (neste caso, no        **/
/**     cluster PUMA do LCCA USP - Laboratório de           **/
/**     Computação Científica Avançada da USP).             **/
/**   II.4. jobBGr.x - exemplo de script Portable Batch     **/
/**     System (PBS) para executar o programa 'bestGraph'   **/
/**     em um cluster (neste caso, no cluster PUMA do LCCA  **/
/**     USP - Laboratório de Computação Científica          **/
/**     Avançada da USP).                                   **/
/**   II.5. rotula.sh - script que verifica e organiza os   **/
/**     dados experimentais disponíveis, criando arquivos   **/
/**     na pasta 'path' com a descrição dos dados.          **/
```

```
/**     Por exemplo, o arquivo 'HPmousesP1.dat' contém      **/
/**     informações a respeito dos dados observados no      **/
/**     hipocampo (HP) na primeira parte do experimento     **/
/**     (P1 - antes dos ratos entrarem em contatos com os   **/
/**     objetos geométricos). A linha '5 13 1.380850 3610'  **/
/**     neste arquivo indica que no rato 5 foram            **/
/**     observados 13 neurônios no hipocampo onde o tempo   **/
/**     (máximo) inicial de observação aconteceu aos        **/
/**     1.380850 segundos e onde o menor tempo em que este  **/
/**     rato entrou em contato com os objetos geométricos   **/
/**     ocorreu aos 3610 segundos. Esta linha é seguida de  **/
/**     outras 13 linhas contendo o nome e o caminho para o **/
/**     arquivo contendo os disparos de cada neurônio.      **/
/**   II.6. graphs.r - script em R para gerar figuras dos   **/
/**     grafos a partir do arquivo contendo a matriz de     **/
/**     adjacência. O script não é automatizado, sendo      **/
/**     necessário editá-lo para cada caso específico.      **/
/**                                                         **/
/**  III. Código fonte:                                     **/
/**                                                         **/
/**   III.1. graphPenalty.c - programa cliente para         **/
/**     estimar, para cada rato, o grafo que melhor         **/
/**     representa os dados observados, na primeira e       **/
/**     terceira partes do experimento, considerando        **/
/**     diferentes valores para a constante de penalização  **/
/**     da probabilidade a posteriori e considerando os 3   **/
/**     métodos diferentes para o cálculo da probabilidade  **/
/**     a posteriori (veja o executável 'graphPenalty'      **/
/**     acima).                                             **/
/**   III.2. bestGraph.c - programa cliente que executa o   **/
/**     método de Monte Carlo com Cadeia de Markov no       **/
/**     espaço dos grafos, representando as conexões entre  **/
/**     neurônios, para estimar o grafo mais                **/
/**     representativo de cada conjunto de neurônios        **/
/**     observados, dado a constante de penalização         **/
/**     (escolhida com base na análise dos resultados do    **/
/**     executável 'graphPenalty') e dado o método para o   **/
/**     cálculo da probabilidade a posteriori (veja o       **/
/**     executável 'bestGraph' acima).                      **/
/**   III.3. Neuro.h - interface para método de Monte       **/
/**     Carlo com Cadeia de Markov no espaço dos grafos,    **/
/**     representando as conexões entre neurônios.          **/
/**   III.4. Neuro.c - Implementação do método de Monte     **/
/**     Carlo com Cadeia de Markov no espaço dos grafos,    **/
/**     representando as conexões entre neurônios. A        **/
/**     partir dos dados experimentais, gera-se a Cadeia    **/
/**     de Markov, em espaço de grafos não direcionados,    **/
/**     cuja distribuição limite é dada pela probabilidade  **/
/**     a posteriori 'P(g|X)'. Isto é feito por meio do     **/
/**     método de Monte Carlo com o algoritmo de            **/
/**     Metropolis.                                         **/
/**   III.5. ST.c - implementação de tipo abstrato de       **/
/**     dados para tabelas de símbolos "skip list", cujos   **/
/**     itens possuem um contador que é incrementado cada   **/
/**     vez que a busca por um item já existente é          **/
/**     realizada (para evitar a necessidade de armazenar   **/
/**     todo novo item gerado, i.e., somente itens          **/
/**     distintos são armazenados).                         **/
/**   III.6. ST.h - interface de tipo abstrato de dados     **/
/**     para tabelas de símbolos, cujos itens possuem um    **/
/**     contador que é incrementado cada vez que a busca    **/
/**     por um item já existente é realizada.               **/
/**   III.7. Item.c - implementação de tipo abstrato de     **/
```

```
/**     dados para grafos (i.e. palavras contendo '0's e    **/
/**     '1's).                                              **/
/**   III.8. Item.h - interface de tipo abstrato de         **/
/**     dados.                                              **/
/**   III.9. Utils.c - implementações alternativas de       **/
/**     funções bem conhecidas de 'stdio.h' e 'stdlib.h'    **/
/**     para evitar repetição de código.                    **/
/**   III.10. Utils.h - interface para versões              **/
/**     alternativas de funções bem conhecidas de           **/
/**     'stdio.h' e 'stdlib.h' para evitar repetição de     **/
/**     código.                                             **/
/**   III.11. Makefile - este arquivo é lido pelo programa  **/
/**     'make' para compilar todos os arquivos do programa  **/
/**     (ou somente os que sofreram mudanças).              **/
/**                                                         **/
/**  IV. Execução:                                          **/
/**                                                         **/
/**   IV.1. linux: abra um terminal e vá para a pasta       **/
/**     principal do programa ('cd Versao6'). Em seguida    **/
/**     execute o script 'linux.sh' ('./linux.sh') e siga   **/
/**     as instruções.                                      **/
/**   IV.2. mac: abra um terminal e vá para a pasta         **/
/**     principal do programa ('cd Versao6'). Em seguida    **/
/**     execute o script 'mac.sh' ('./mac.sh') e siga as    **/
/**     instruções.                                         **/
/**   IV.3. clusters com PBS: edite os scripts 'jobPen.x '  **/
/**     e 'jobBGr.x' de acordo com as especificações da     **/
/**     máquina a ser utilizada e submeta o script          **/
/**     ('qsub jobPen.x').                                  **/
/**                                                         **/
/**  V. Dependências para execução:                         **/
/**                                                         **/
/**   V.1. gcc (testado com versões de 4.1 a 4.6).          **/
/**   V.2. GNU Make (testado com versão 3.81).              **/
/**   V.3. para usar o script 'graph.r' é necessário ter    **/
/**     instalada a biblioteca 'gRbase'.                    **/
/**                                                         **/
/**  *****************************************************  **/
/**                                                         **/
/**  Versões:                                               **/
/**                                                         **/
/**   1 (18.05.2012): Origem das idéias principais          **/
/**     - construção da estrutura de skip list com          **/
/**     adaptações convenientes ao problema.                **/
/**     - construção do Monte Carlo com Cadeia de Markov.   **/
/**     - construção do script 'rotula.sh' para organizar   **/
/**     os dados.                                           **/
/**   2 (25.05.2012):                                       **/
/**     - mudança no script 'rotula.sh' para considerar     **/
/**     somente os intervalos de tempo antes (parte 1) e    **/
/**     após (parte 3) os ratos entrarem em contato com os  **/
/**     objetos geométricos.                                **/
/**     - extensão do programa para considerar as demais    **/
/**     regiões do cérebro observadas.                      **/
/**   3 (27.05.2012):                                       **/
/**     - tentativa sem sucesso de definir um critério de   **/
/**     parada do Monte Carlo calculando-se o número        **/
/**     decimal correspondente a cada grafo (número         **/
/**     binário). Se isto fosse viável, seria possível      **/
/**     calcular a variância dos grafos em tempo real com   **/
/**     um truque esperto. No entanto, a representação      **/
/**     decimal só é possível para grafos com poucos        **/
/**     vértices devido à limitação de precisão do          **/
```

```
/**     computador na representação de números.             **/
/**   4 (30.05.2012):                                       **/
/**     - reorganização do programa em diretórios: 'bin'    **/
/**     para executáveis binários, 'data' para os dados     **/
/**     experimentais, 'out' para os arquivos de saída,     **/
/**     'scripts' para scripts como 'rotula.sh' (para       **/
/**     organizar os dados disponíveis) e 'graph.r' (para   **/
/**     gerar imagens dos grafos com R), 'path' os          **/
/**     arquivos produzidos pelo script 'rotula.sh', e      **/
/**     'src' para o código fonte.                          **/
/**     - correção de erro no contador do Monte Carlo.      **/
/**     - construção de scripts, para linux e mac, para     **/
/**     facilitar a execução do programa.                   **/
/**   5 (01.06.2012):                                       **/
/**     - correção de erro na liberação de memória.         **/
/**     - inclusão nos scripts de execução de cálculo da    **/
/**     memória disponível no sistema para maximizar o      **/
/**     número de passos de Monte Carlo.                    **/
/**   6 (04.06.2012):                                       **/
/**     - programa dividido em duas funções principais. A   **/
/**     primeira é a 'graphPenalty.c' que calcula, para     **/
/**     diferentes valores da constante de penalização, a   **/
/**     probabilidade a posteriori por meio de 3 métodos    **/
/**     diferentes:                                         **/
/**       1. <Xi|Xj>=1 se Xi=1 e Xj=1                       **/
/**       2. <Xi|Xj>=1 se Xi=Xj ou <Xi|Xj>=0 (se Xi!=Xj)    **/
/**       3. <Xi|Xj>=1 se Xi=Xj ou <Xi|Xj>=-1 (se Xi!=Xj)   **/
/**     A segunda é a 'bestGraph.c' que, dada a constante   **/
/**     de penalização e um método específico, calcula o    **/
/**     grafo mais representativo via método de Monte       **/
/**     Carlo com Cadeia de Markov.                         **/
/**     - correção de erro na função 'mcSteps' de           **/
/**     'Neuro.c'.                                          **/
/**     - correção de erro na alocação de memória dos       **/
/**     grafos.                                             **/
/**   7 (09.06.2012):                                       **/
/**     - código totalmente comentado.                      **/
/**     - inclusão, nos scripts de execução, de opção da    **/
/**     região do cérebro e de escolha do rato a ser        **/
/**     estudado.                                           **/
/**                                                         **/
/**  *****************************************************  **/
```

A.2 linux.sh

Script para usuários linux[1] para compilação e execução do programa.

```
#!/bin/bash

clear

# Prints the header.
echo -e "/**  *****************************************************  **/"
echo -e "/**       ** Finding the Most Representative Graph **       **/"
echo -e "/**       **    Model for Neuronal Interactions    **       **/"
echo -e "/**       **     via Markov Chain Monte Carlo      **       **/"
echo -e "/**                                                         **/"
echo -e "/**                    **  Version 7  **                    **/"
```

[1] O script para mac, "mac.sh", é muito parecido e será omitido neste relatório.

```
echo -e "/**                                                          **/"
echo -e "/**                          IME/USP                         **/"
echo -e "/**                                                          **/"
echo -e "/**   Advisor: Prof. Dr. Antonio Galves                      **/"
echo -e "/**                                                          **/"
echo -e "/**   Co-advisors: Aline Duarte de Oliveira                  **/"
echo -e "/**                Guilherme Ost de Aguiar                   **/"
echo -e "/**                                                          **/"
echo -e "/**   Fundamental aids and hints from Prof. Dr. Yoshiharu    **/"
echo -e "/**                                            Kohayakawa    **/"
echo -e "/**                                                          **/"
echo -e "/**   Author: Pedro Brandimarte Mendonca                     **/"
echo -e "/**                                                          **/"
echo -e "/**  ******************************************************  **/"
echo -e "/**   For info about the code check the file 'readMe.txt'    **/"
echo -e "/**  ******************************************************  **/"

# Checks if data path directory exists and is readable.
check=path
echo -e "\nChecking data path.\n"
if [ -r ${check} ]
then
   check=`ls path/ | wc -l`

   # Checks if the directory contains 12 files.
   if [ "${check}" != "12" ]
   then
   echo -n "Organizing the data... "
      cd scripts
      ./rotula.sh
      cd ../
      echo -e "done\n"
   fi
else
   echo -n "Organizing the data... "
   mkdir path
   cd scripts
   ./rotula.sh
   cd ../
   echo -e "done\n"
fi

# Clears swap memory if necessary.
SWAPmem=`free -b | grep "Swap:" | awk '{print $3}'`
if [ "${SWAPmem}" != "0" ]
then
   echo -e "Cleaning swap memory (only for super-user)..."
   sudo swapoff -a
   sudo swapon -a
   echo -e "...done\n"
fi

# Computes the total memory available in bits.
FREEmem=`free -b | grep "Mem:" | awk '{print $4}'`
MCmem=`echo "(8 * ${FREEmem})" | bc`
echo -e "Available memory: ${MCmem} bits\n"

verif=0
echo -e "Choose the kind of computation:"
while [ ${verif} -eq 0 ]
do
   echo -n "(type 0 for penalty analysis or 1 for best graph) "
   read option
```

```
if [ "${option}" == "0" ]
then
   # Compiles the program.
   cd src
   echo -e ""
   while [ ${verif} -eq 0 ]
   do
      echo -n "Do you want to clean old compilations? (y or n) "
      read option
      if [ "${option}" == "y" ]
      then
         verif=1
         echo -e "\nCleaning old compilations:\n"
         make clean
         make graphPenalty
         check=‘echo $?‘
         if [ ${check} == 2 ]
         then
            exit -1
         fi
         echo -e "...done\n"
      elif [ "${option}" == "n" ]
      then
         verif=1
         echo -e "\nCompiling changes:\n"
         make graphPenalty
         check=‘echo $?‘
         if [ ${check} == 2 ]
         then
            exit -1
         fi
         echo -e "...done\n"
      else
         echo -e "\nWrong option!\n"
      fi
   done
   cd ../

   # Choosing the mouse
   verif=0
   while [ ${verif} -eq 0 ]
   do
      echo -n "Choose the mouse you want to study (4, 5, 6, 9, 12, 13 or all): "
      read opt
      if [ "${opt}" == "4" ]
      then
         mouse="4"
         stepsMax=‘echo "3 * ${MCmem} / 756" | bc‘
         verif=1
      elif [ "${opt}" == "5" ]
      then
         mouse="5"
         stepsMax=‘echo "3 * ${MCmem} / 520" | bc‘
         verif=1
      elif [ "${opt}" == "6" ]
      then
         mouse="6"
         stepsMax=‘echo "3 * ${MCmem} / 756" | bc‘
         verif=1
      elif [ "${opt}" == "9" ]
      then
         mouse="9"
         stepsMax=‘echo "3 * ${MCmem} / 1722" | bc‘
```

```
      verif=1
   elif [ "${opt}" == "12" ]
   then
      mouse="12"
      stepsMax=‘echo "3 * ${MCmem} / 812" | bc‘
      verif=1
   elif [ "${opt}" == "13" ]
   then
      mouse="13"
      stepsMax=‘echo "3 * ${MCmem} / 930" | bc‘
      verif=1
   elif [ "${opt}" == "all" ]
   then
      mouse="all"
      stepsMax=‘echo "3 * ${MCmem} / 1722" | bc‘
      verif=1
   else
      echo -e "\nWrong option!\n"
   fi
done

# Choosing the brain region
verif=0
echo -e ""
while [ ${verif} -eq 0 ]
do
   echo -e "Choose the brain region you want to study"
   echo -e "1. Hippocampus (HP)"
   echo -e "2. Primary Somatic Sensory Cortex (S1)"
   echo -e "3. Primary Visual Cortex (V1)"
   echo -e "4. Hippocampus Dentate Gyrus (HPDG)"
   echo -e "5. Hippocampus Cornu Ammonis 1 (HPCA1)"
   echo -e "6. all regions\n"
   echo -n "Type the option (1, 2, 3, 4, 5 or 6): "
   read opt
   if [ "${opt}" == "1" ]
   then
      region="HP"
      verif=1
   elif [ "${opt}" == "2" ]
   then
      region="S1"
      verif=1
   elif [ "${opt}" == "3" ]
   then
      region="V1"
      verif=1
   elif [ "${opt}" == "4" ]
   then
      region="HPDG"
      verif=1
   elif [ "${opt}" == "5" ]
   then
      region="HPCA1"
      verif=1
   elif [ "${opt}" == "6" ]
   then
      region="0"
      verif=1
   else
      echo -e "\nWrong option!\n"
   fi
done
```

```
# Ask if the user wants an arbitrary number of MC steps.
verif=0
echo -e ""
while [ ${verif} -eq 0 ]
do
   echo -n "Do you want to choose a fixed number of MC steps? (y or n) "
   read option
   if [ "${option}" == "y" ]
   then
      while [ ${verif} -eq 0 ]
      do
         echo -e ""
         echo -n "Type the number of MC steps (less than ${stepsMax}): "
         read mcSteps
         if [ ${mcSteps} -lt ${stepsMax} ]
         then
            MCmem=${mcSteps}
            stepsMax=1
            verif=1
         else
            echo -e "\nWrong option!"
         fi
      done
   elif [ "${option}" == "n" ]
   then
      stepsMax=0
      verif=1
   else
      echo -e "\nWrong option!\n"
   fi
done

# Checks if output directory exists and is readable.
check=out/outPenalty
if [ -r ${check} ]
then
   echo -e ""
   echo -n "Removing old results "
   if [ "${region}" == "0" ] && [ "${mouse}" == "all" ]
   then
      rm out/outPenalty/*
      echo -e "...done\n"
   elif [ "${region}" == "0" ]
   then
      rm out/outPenalty/*${mouse}*
      echo -e "...done\n"
   elif [ "${mouse}" == "all" ]
   then
      rm out/outPenalty/*${region}*
      echo -e "...done\n"
   else
      rm out/outPenalty/*${mouse}${region}*
      echo -e "...done\n"
   fi
else
   echo -e ""
   echo -n "Creating directory for output results... "
   mkdir out/outPenalty
   echo -e "done\n"
fi

# Runs the program.
```

```
    cd bin
    echo -e "\nRunning the program...\n"
    echo -n "Start of run: "
    date
    ini=$(date +%s%N) # initial time with nanoseconds accuracy
    if [ "${region}" == "0" ] && [ "${mouse}" == "all" ]
    then
       for region in "HP" "S1" "V1" "HPDG" "HPCA1"
       do
          for mouse in "4" "5" "6" "9" "12" "13"
          do
            ./graphPenalty ../path/ ../out/outPenalty/ ${MCmem} ${stepsMax} ${region} ${mouse}
          done
       done
    elif [ "${region}" == "0" ]
    then
       for region in "HP" "S1" "V1" "HPDG" "HPCA1"
       do
          ./graphPenalty ../path/ ../out/outPenalty/ ${MCmem} ${stepsMax} ${region} ${mouse}
       done
    elif [ "${mouse}" == "all" ]
    then
       for mouse in "4" "5" "6" "9" "12" "13"
       do
          ./graphPenalty ../path/ ../out/outPenalty/ ${MCmem} ${stepsMax} ${region} ${mouse}
       done
    else
       ./graphPenalty ../path/ ../out/outPenalty/ ${MCmem} ${stepsMax} ${region} ${mouse}
    fi
    fim=$(date +%s%N) # final time with nanoseconds accuracy
    echo -e ""
    echo -n "End of run: "
    date
    tempo=`echo "scale = 10; (${fim} - ${ini}) / 60000000000" | bc`
    echo -e "\nRun time: ${tempo} min\n"
    cd ../

elif [ "${option}" == "1" ]
then
    # Compiles the program.
    cd src
    echo -e ""
    while [ ${verif} -eq 0 ]
    do
       echo -n "Do you want to clean old compilations? (y or n) "
       read option
       if [ "${option}" == "y" ]
       then
          verif=1
          echo -e "\nCleaning old compilations:\n"
          make clean
          make bestGraph
          check=`echo $?`
          if [ ${check} == 2 ]
          then
             exit -1
          fi
          echo -e "...done\n"
       elif [ "${option}" == "n" ]
       then
          verif=1
          echo -e "\nCompiling changes:\n"
          make bestGraph
```

```
      check=‘echo $?‘
      if [ ${check} == 2 ]
      then
         exit -1
      fi
      echo -e "...done\n"
   else
      echo -e "\nWrong option!\n"
   fi
done
cd ../

# Choosing the mouse
verif=0
while [ ${verif} -eq 0 ]
do
   echo -n "Choose the mouse you want to study (4, 5, 6, 9, 12 or 13): "
   read opt
   if [ "${opt}" == "4" ]
   then
      mouse="4"
      stepsMax=‘echo "3 * ${MCmem} / 756" | bc‘
      verif=1
   elif [ "${opt}" == "5" ]
   then
      mouse="5"
      stepsMax=‘echo "3 * ${MCmem} / 520" | bc‘
      verif=1
   elif [ "${opt}" == "6" ]
   then
      mouse="6"
      stepsMax=‘echo "3 * ${MCmem} / 756" | bc‘
      verif=1
   elif [ "${opt}" == "9" ]
   then
      mouse="9"
      stepsMax=‘echo "3 * ${MCmem} / 1722" | bc‘
      verif=1
   elif [ "${opt}" == "12" ]
   then
      mouse="12"
      stepsMax=‘echo "3 * ${MCmem} / 812" | bc‘
      verif=1
   elif [ "${opt}" == "13" ]
   then
      mouse="13"
      stepsMax=‘echo "3 * ${MCmem} / 930" | bc‘
      verif=1
   else
      echo -e "\nWrong option!\n"
   fi
done

# Choosing the brain region
verif=0
echo -e ""
while [ ${verif} -eq 0 ]
do
   echo -e "Choose the brain region you want to study"
   echo -e "1. Hippocampus (HP)"
   echo -e "2. Primary Somatic Sensory Cortex (S1)"
   echo -e "3. Primary Visual Cortex (V1)"
   echo -e "4. Hippocampus Dentate Gyrus (HPDG)"
```

```
   echo -e "5. Hippocampus Cornu Ammonis 1 (HPCA1)\n"
   echo -n "Type the option (1, 2, 3, 4 or 5): "
   read opt
   if [ "${opt}" == "1" ]
   then
      region="HP"
      verif=1
   elif [ "${opt}" == "2" ]
   then
      region="S1"
      verif=1
   elif [ "${opt}" == "3" ]
   then
      region="V1"
      verif=1
   elif [ "${opt}" == "4" ]
   then
      region="HPDG"
      verif=1
   elif [ "${opt}" == "5" ]
   then
      region="HPCA1"
      verif=1
   else
      echo -e "\nWrong option!\n"
   fi
done

# Ask if the user wants an arbitrary number of MC steps.
verif=0
echo -e ""
while [ ${verif} -eq 0 ]
do
   echo -n "Do you want to choose a fixed number of MC steps? (y or n) "
   read option
   if [ "${option}" == "y" ]
   then
      while [ ${verif} -eq 0 ]
      do
         echo -e ""
         echo -n "Type the number of MC steps (less than ${stepsMax}): "
         read mcSteps
         if [ ${mcSteps} -lt ${stepsMax} ]
         then
            MCmem=${mcSteps}
            stepsMax=1
            verif=1
         else
            echo -e "\nWrong option!"
         fi
      done
   elif [ "${option}" == "n" ]
   then
      stepsMax=0
      verif=1
   else
      echo -e "\nWrong option!\n"
   fi
done

# Checks if output directory exists and is readable.
check=out/outBestGraph
if [ -r ${check} ]
```

```
      then
         echo -e ""
         echo -n "Removing old results "
         rm out/outBestGraph/*${mouse}${region}*
         echo -e "...done\n"
      else
         echo -e ""
         echo -n "Creating directory for output results... "
         mkdir out/outBestGraph
         echo -e "done\n"
      fi

      # Choosing the method for probability computation
      verif=0
      while [ ${verif} -eq 0 ]
      do
         echo -n "Choose the method for probability computation (1, 2 or 3): "
         read method
         if [ "${method}" == "1" ]
         then
            verif=1
         elif [ "${method}" == "2" ]
         then
            verif=1
         elif [ "${method}" == "3" ]
         then
            verif=1
         else
            echo -e "\nWrong option!\n"
         fi
      done

      # Choosing the penalty constant
      echo -e ""
      echo -n "Type the positive penalty constant: "
      read penal

      # Runs the program.
      cd bin
      echo -e "\nRunning the program...\n"
      echo -n "Start of run: "
      date
      ini=$(date +%s%N) # initial time with nanoseconds accuracy
      ./bestGraph ../path/ ../out/outBestGraph/ ${MCmem} ${stepsMax} ${region} ${mouse} \
       ${method} ${penal}
      fim=$(date +%s%N) # final time with nanoseconds accuracy
      echo -e "\n"
      echo -n "End of run: "
      date
      tempo=`echo "scale = 10; (${fim} - ${ini}) / 60000000000" | bc`
      echo -e "\nRun time: ${tempo} min\n"
      cd ../
   else
      echo -e "\nWrong option!\n"
   fi
done
```

A.3 rotula.sh

Script que verifica e organiza os dados experimentais disponíveis, criando arquivos na pasta "path" com a descrição dos dados.

```
# ************************************************************* #
#                                                               #
# This script verifies the experimental data at 'data' folder and #
# creates files in 'path' folder describing all the observed      #
# data.                                                         #
#                                                               #
# ************************************************************

#!/bin/bash

DAT=../data/ # directory with experimental data
SPK=/spikes/01/ # path inside the data folder
DPATH=../path/ # output folder

# Gets the mouses.
MOUSE=`ls ${DAT}`
MOUSE=`echo "${MOUSE}" | grep "ge" | tr -d 'ge' | sort -n`

# Creates the files at output folder.
echo "${MOUSE}" > ${DPATH}mouses.dat
> ${DPATH}HPmousesP1.dat
> ${DPATH}HPmousesP3.dat
> ${DPATH}HPCA1mousesP1.dat
> ${DPATH}HPCA1mousesP3.dat
> ${DPATH}HPDGmousesP1.dat
> ${DPATH}HPDGmousesP3.dat
> ${DPATH}S1mousesP1.dat
> ${DPATH}S1mousesP3.dat
> ${DPATH}V1mousesP1.dat
> ${DPATH}V1mousesP3.dat
> ${DPATH}dataPath.dat

for RAT in ${MOUSE}
do
   echo "${DAT}ge${RAT}/" >> ${DPATH}dataPath.dat
   all=`ls ${DAT}ge${RAT}${SPK}`
   check=`ls ${DAT}ge${RAT} | grep "contacts"`
   if [ ${check} ] # gets min and max time where the mouse
                   # was in touch with the geometric objects
   then
      min=`awk '{printf "%d\n", $1}' ${DAT}ge${RAT}/ge${RAT}_contacts.txt |sort -n |head -1`
      max=`awk -F, '{printf "%d\n", $2}' ${DAT}ge${RAT}/ge${RAT}_contacts.txt |sort -n |tail -1`
   fi

   # Hippocampus.
   hp=`echo "${all}" | grep "HP" | sed "s/HP_//" | tr -d '.spk' | sed '/i/d'`

   ca1=`echo "${hp}" | grep "CA1" | head -1 | cut -c1-3`
   # Hippocampus Cornu Ammonis 1.
   if [ "$ca1" == "CA1" ]
   then
      # Gets the names of the observed neurons.
      ca1=`echo "${hp}" | grep "CA1" | sed "s/CA1_//"`
      echo "${ca1}" > ${DAT}ge${RAT}/rotulosHPCA1.dat
      count=`echo "${ca1}" | wc -l` # count the number of observed neurons
      inf=0.0
```

```
sup=100000.0
for ca1rot in ${ca1}
do
   # Finds the (maximum) start time.
   aux=‘head -1 ${DAT}ge${RAT}${SPK}HP_CA1_${ca1rot}.spk‘
   if [ ‘echo "${aux} > ${inf}" | bc‘ == 1 ]
   then
      inf=‘echo ${aux}‘
   fi
   # Finds the (minimum) end time.
   aux=‘tail -1 ${DAT}ge${RAT}${SPK}HP_CA1_${ca1rot}.spk‘
   if [ ‘echo "${aux} < ${sup}" | bc‘ == 1 ]
   then
      sup=‘echo ${aux}‘
   fi
done
if [ ${check} ]
then
   echo "${RAT}" "${count}" "${inf}" "${min}" >> ${DPATH}HPCA1mousesP1.dat
   echo "${RAT}" "${count}" "${max}" "${sup}" >> ${DPATH}HPCA1mousesP3.dat
   for ca1rot in ${ca1}
   do
      echo "${ca1rot} ${DAT}ge${RAT}${SPK}HP_CA1_${ca1rot}.spk" \
       >> ${DPATH}HPCA1mousesP1.dat
      echo "${ca1rot} ${DAT}ge${RAT}${SPK}HP_CA1_${ca1rot}.spk" \
       >> ${DPATH}HPCA1mousesP3.dat
   done
else # there isn’t information about when the mouse got in touch with geometric objects
   echo "${RAT}" "${count}" "${inf}" "${sup}" >> ${DPATH}HPCA1mousesP1.dat
   for ca1rot in ${ca1}
   do
      echo "${ca1rot} ${DAT}ge${RAT}${SPK}HP_CA1_${ca1rot}.spk" \
       >> ${DPATH}HPCA1mousesP1.dat
   done
fi

dg=‘echo "${hp}" | grep "DG" | head -1 | cut -c1-2‘
# Hippocampus Dentate Gyrus.
if [ "$dg" == "DG" ]
then
   # Gets the names of the observed neurons.
   dg=‘echo "${hp}" | grep "DG" | sed "s/DG_//"‘
   echo "${dg}" > ${DAT}ge${RAT}/rotulosHPDG.dat
   count=‘echo "${dg}" | wc -l‘ # count the number of observed neurons
   inf=0.0
   sup=100000.0
   for dgrot in ${dg}
   do
      # Finds the (maximum) start time.
      aux=‘head -1 ${DAT}ge${RAT}${SPK}HP_DG_${dgrot}.spk‘
      if [ ‘echo "${aux} > ${inf}" | bc‘ == 1 ]
      then
         inf=‘echo ${aux}‘
      fi
      # Finds the (minimum) end time.
      aux=‘tail -1 ${DAT}ge${RAT}${SPK}HP_DG_${dgrot}.spk‘
      if [ ‘echo "${aux} < ${sup}" | bc‘ == 1 ]
      then
         sup=‘echo ${aux}‘
      fi
   done
   if [ ${check} ]
   then
```

```
         echo "${RAT}" "${count}" "${inf}" "${min}" >> ${DPATH}HPDGmousesP1.dat
         echo "${RAT}" "${count}" "${max}" "${sup}" >> ${DPATH}HPDGmousesP3.dat
         for dgrot in ${dg}
         do
            echo "${dgrot} ${DAT}ge${RAT}${SPK}HP_DG_${dgrot}.spk" \
             >> ${DPATH}HPDGmousesP1.dat
            echo "${dgrot} ${DAT}ge${RAT}${SPK}HP_DG_${dgrot}.spk" \
             >> ${DPATH}HPDGmousesP3.dat
         done
      else # there isn't information about when the mouse got in touch with geometric objects
         echo "${RAT}" "${count}" "${inf}" "${sup}" >> ${DPATH}HPDGmousesP1.dat
         for dgrot in ${dg}
         do
            echo "${dgrot} ${DAT}ge${RAT}${SPK}HP_DG_${dgrot}.spk" \
             >> ${DPATH}HPDGmousesP1.dat
         done
      fi
   fi
else
   # Gets the names of the observed neurons.
   echo "${hp}" > ${DAT}ge${RAT}/rotulosHP.dat
   count=`echo "${hp}" | wc -l` # count the number of observed neurons
   inf=0.0
   sup=100000.0
   for hprot in ${hp}
   do
      # Finds the (maximum) start time.
      aux=`head -1 ${DAT}ge${RAT}${SPK}HP_${hprot}.spk`
      if [ `echo "${aux} > ${inf}" | bc` == 1 ]
      then
         inf=`echo ${aux}`
      fi
      # Finds the (minimum) end time.
      aux=`tail -1 ${DAT}ge${RAT}${SPK}HP_${hprot}.spk`
      if [ `echo "${aux} < ${sup}" | bc` == 1 ]
      then
         sup=`echo ${aux}`
      fi
   done
   if [ ${check} ]
   then
      echo "${RAT}" "${count}" "${inf}" "${min}" >> ${DPATH}HPmousesP1.dat
      echo "${RAT}" "${count}" "${max}" "${sup}" >> ${DPATH}HPmousesP3.dat
      for hprot in ${hp}
      do
         echo "${hprot} ${DAT}ge${RAT}${SPK}HP_${hprot}.spk" >> ${DPATH}HPmousesP1.dat
         echo "${hprot} ${DAT}ge${RAT}${SPK}HP_${hprot}.spk" >> ${DPATH}HPmousesP3.dat
      done
   else # there isn't information about when the mouse got in touch with geometric objects
      echo "${RAT}" "${count}" "${inf}" "${sup}" >> ${DPATH}HPmousesP1.dat
      for hprot in ${hp}
      do
         echo "${hprot} ${DAT}ge${RAT}${SPK}HP_${hprot}.spk" >> ${DPATH}HPmousesP1.dat
      done
   fi
fi

# Primary Somatic Sensory Cortex.
# Gets the names of the observed neurons.
s1=`echo "${all}" | grep "S1" | sed "s/S1_//" | tr -d '.spk' | sed '/i/d'`
echo "${s1}" > ${DAT}ge${RAT}/rotulosS1.dat
count=`echo "${s1}" | wc -l` # count the number of observed neurons
inf=0.0
```

```
sup=100000.0
for s1rot in ${s1}
do
   # Finds the (maximum) start time.
   aux=‘head -1 ${DAT}ge${RAT}${SPK}S1_${s1rot}.spk‘
   if [ ‘echo "${aux} > ${inf}" | bc‘ == 1 ]
   then
      inf=‘echo ${aux}‘
   fi
   # Finds the (minimum) end time.
   aux=‘tail -1 ${DAT}ge${RAT}${SPK}S1_${s1rot}.spk‘
   if [ ‘echo "${aux} < ${sup}" | bc‘ == 1 ]
   then
      sup=‘echo ${aux}‘
   fi
done
if [ ${check} ]
then
   echo "${RAT}" "${count}" "${inf}" "${min}" >> ${DPATH}S1mousesP1.dat
   echo "${RAT}" "${count}" "${max}" "${sup}" >> ${DPATH}S1mousesP3.dat
   for s1rot in ${s1}
   do
      echo "${s1rot} ${DAT}ge${RAT}${SPK}S1_${s1rot}.spk" >> ${DPATH}S1mousesP1.dat
      echo "${s1rot} ${DAT}ge${RAT}${SPK}S1_${s1rot}.spk" >> ${DPATH}S1mousesP3.dat
   done
else # there isn’t information about when the mouse got in touch with geometric objects
   echo "${RAT}" "${count}" "${inf}" "${sup}" >> ${DPATH}S1mousesP1.dat
   for s1rot in ${s1}
   do
      echo "${s1rot} ${DAT}ge${RAT}${SPK}S1_${s1rot}.spk" >> ${DPATH}S1mousesP1.dat
   done
fi

# Primary Visual Cortex.
# Gets the names of the observed neurons.
v1=‘echo "${all}" | grep "V1" | sed "s/V1_//" | tr -d ’.spk’ | sed ’/i/d’‘
echo "${v1}" > ${DAT}ge${RAT}/rotulosV1.dat
count=‘echo "${v1}" | wc -l‘ # count the number of observed neurons
inf=0.0
sup=100000.0
for v1rot in ${v1}
do
   # Finds the (maximum) start time.
   aux=‘head -1 ${DAT}ge${RAT}${SPK}V1_${v1rot}.spk‘
   if [ ‘echo "${aux} > ${inf}" | bc‘ == 1 ]
   then
      inf=‘echo ${aux}‘
   fi
   # Finds the (minimum) end time.
   aux=‘tail -1 ${DAT}ge${RAT}${SPK}V1_${v1rot}.spk‘
   if [ ‘echo "${aux} < ${sup}" | bc‘ == 1 ]
   then
      sup=‘echo ${aux}‘
   fi
done
if [ ${check} ]
then
   echo "${RAT}" "${count}" "${inf}" "${min}" >> ${DPATH}V1mousesP1.dat
   echo "${RAT}" "${count}" "${max}" "${sup}" >> ${DPATH}V1mousesP3.dat
   for v1rot in ${v1}
   do
      echo "${v1rot} ${DAT}ge${RAT}${SPK}V1_${v1rot}.spk" >> ${DPATH}V1mousesP1.dat
      echo "${v1rot} ${DAT}ge${RAT}${SPK}V1_${v1rot}.spk" >> ${DPATH}V1mousesP3.dat
```

```
      done
   else # there isn't information about when the mouse got in touch with geometric objects
      echo "${RAT}" "${count}" "${inf}" "${sup}" >> ${DPATH}V1mousesP1.dat
      for v1rot in ${v1}
      do
         echo "${v1rot} ${DAT}ge${RAT}${SPK}V1_${v1rot}.spk" >> ${DPATH}V1mousesP1.dat
      done
   fi

done
```

A.4 graphPenalty.c

Programa "cliente" para avaliar a constante de penalização.

```
/**  ************************************************  **/
/**  This is a (client) program that estimates for each     **/
/**  mouse the graph that best represents the observed      **/
/**  data in the first and third parts of the experiment    **/
/**  (before and after the mouse got in touch with          **/
/**  geometric objects) considering different values for    **/
/**  the penalty constant and considering 3 different       **/
/**  methods for computing the posterior probability:       **/
/**    1. <Xi|Xj>=1 if Xi=1 and Xj=1                        **/
/**    2. <Xi|Xj>=1 if Xi=Xj or <Xi|Xj>=0 (if Xi!=Xj)       **/
/**    3. <Xi|Xj>=1 if Xi=Xj or <Xi|Xj>=-1 (if Xi!=Xj)      **/
/**  This is done via  Markov Chain Monte Carlo method.     **/
/**  The output files are sent to the chosen output folder  **/
/**  (default is 'out/outPenalty'). For example:            **/
/**    - the file 'outputM6HPp1Met1.dat' contains general   **/
/**    information about the run for mouse 6 (M6) in the    **/
/**    hippocampus region (HP) in the first part of the     **/
/**    experiment (p1) and where the posterior probability  **/
/**    were computed with method 1 (Met1).                  **/
/**    - the file 'penal1M6HPp1Met1.dat' contains the       **/
/**    penalty constant versus the logarithm of the         **/
/**    non-normalized posterior probability (with penalty   **/
/**    included).                                           **/
/**    - the file 'penal2M6HPp1Met1.dat' contains the       **/
/**    penalty constant versus the logarithm of the         **/
/**    non-normalized posterior probability (without        **/
/**    penalty).                                            **/
/**    - the file 'penal3M6HPp1Met1.dat' contains the       **/
/**    penalty constant versus the empirical probability    **/
/**    (obtained from the Monte Carlo).                     **/
/**  ************************************************  **/

#include <stdio.h>
#include <stdlib.h>
#include "Neuro.h"

int main (int nargs, char *arg[])
{

   /* Checks if the input were typed correctly. */
   if (nargs != 7) {
      fprintf (stderr, "\n\n Wrong number of arguments!\n");
      fprintf (stderr, "\n Use: ./graphPenalty"); /* arg[0] */
      fprintf (stderr, " [directory with data paths]"); /* arg[1] */
      fprintf (stderr, " [directory for output files]"); /* arg[2] */
```

```
      fprintf (stderr, " [available memory]"); /* arg[3] */
      fprintf (stderr, " [fixed # of MC steps option: 0 or 1]"); /* arg[4] */
      fprintf (stderr, " [brain region option]"); /* arg[5] */
      fprintf (stderr, " [chosen mouse]\n\n"); /* arg[6] */
      exit (EXIT_FAILURE);
   }

   /* Sets available memory for graphs storage. */
   NEUROsetMem (arg[3]);

   /* Sets the option for a fixed number of MC steps. */
   NEUROsetMCsteps (arg[4]);

   /* Sets the brain region chosen for study. */
   NEUROsetRegion (arg[5]);

   /* Sets the mouse chosen for study. */
   NEUROsetMouse (arg[6]);

   /* Brain region specified by the user. */
   NEUROpenalAnalysis (arg[1], arg[2]);

   return 0;

} /* main */
```

A.5 bestGraph.c

Programa "cliente" para estimar o grafo mais representativo.

```
/**  ************************************************  **/
/**  This is a (client) program that computes a Markov      **/
/**  Chain Monte Carlo on graphs representing neuronal      **/
/**  connectivity to estimate the most representative       **/
/**  graph for each set of observed neurons. Given a        **/
/**  penalty constant (chosen form a previous analysis      **/
/**  with 'graphPenalty' program) and a specific method     **/
/**  for computing the posterior probability                **/
/**    1. <Xi|Xj>=1 if Xi=1 and Xj=1                        **/
/**    2. <Xi|Xj>=1 if Xi=Xj or <Xi|Xj>=0 (if Xi!=Xj)       **/
/**    3. <Xi|Xj>=1 if Xi=Xj or <Xi|Xj>=-1 (if Xi!=Xj)      **/
/**  it estimates the graph that best represents each       **/
/**  observed brain region of each observed mouse.          **/
/**  The output files are sent to the chosen output folder  **/
/**  (default is 'out/outBestGraph'). For example:          **/
/**    - the file 'outputHPMet1.dat' contains general       **/
/**    information about the run for all mouses where the   **/
/**    hippocampus region (HP) where observed and where     **/
/**    the posterior probability were computed with method  **/
/**    1 (Met1).                                            **/
/**    - the file 'adjM6HPp1Met1.dat' is the adjacent       **/
/**    matrix of the most probable graph of mouse 6 (M6),   **/
/**    in the hippocampus region (HP), in the first part    **/
/**    of the experiment (p1) and where the posterior       **/
/**    probability was computed with method 1 (Met1).       **/
/**  ************************************************  **/

#include <stdio.h>
#include <stdlib.h>
#include "Neuro.h"
```

```
int main (int nargs, char *arg[])
{

   /* Checks if the input were typed correctly. */
   if (nargs != 9) {
      fprintf (stderr, "\n\n Wrong number of arguments!\n");
      fprintf (stderr, "\n Use: ./bestGraph"); /* arg[0] */
      fprintf (stderr, " [directory with data paths]"); /* arg[1] */
      fprintf (stderr, " [directory for output files]"); /* arg[2] */
      fprintf (stderr, " [available memory]"); /* arg[3] */
      fprintf (stderr, " [fixed # of MC steps option: 0 or 1]"); /* arg[4] */
      fprintf (stderr, " [brain region option]"); /* arg[5] */
      fprintf (stderr, " [chosen mouse]"); /* arg[6] */
      fprintf (stderr,
               " [method for probability computation (0,1,2)]"); /* arg[7] */
      fprintf (stderr, " [penalty value]\n\n"); /* arg[8] */
      exit (EXIT_FAILURE);
   }

   /* Sets the available memory for graphs storage. */
   NEUROsetMem (arg[3]);

   /* Sets the option for a fixed number of MC steps. */
   NEUROsetMCsteps (arg[4]);

   /* Sets the brain region chosen for study. */
   NEUROsetRegion (arg[5]);

   /* Sets the mouse chosen for study. */
   NEUROsetMouse (arg[6]);

   /* Sets the method for posterior probability computation. */
   NEUROsetMethod (arg[7]);

   /* Sets the penalty constant. */
   NEUROsetPenal (arg[8]);

   /* Brain region specified by the user. */
   NEURObestGraph (arg[1], arg[2]);

   return 0;

} /* main */
```

A.6 Neuro.h

Interface para método de Monte Carlo com Cadeia de Markov no espaço dos grafos, representando as conexões entre neurônios.

```
/**  ************************************************  **/
/**  Interface of Markov Chain Monte Carlo on graphs    **/
/**  representing neuronal connectivity.                **/
/**  ************************************************  **/

/* Matrix index in row-major order. */
#define idx2d(i, j, n) ((i) * (n) + j)

/* Sets the available memory for graphs storage. */
```

```
void NEUROsetMem (char *freeMem);

/* Sets the option for a fixed number of Monte Carlo steps. */
void NEUROsetMCsteps (char *steps);

/* Sets the brain region chosen for study. */
void NEUROsetRegion (char *rg);

/* Sets the mouse chosen for study. */
void NEUROsetMouse (char *mouse);

/* Sets the method for posterior probability computation. */
void NEUROsetMethod (char *method);

/* Sets the penalty constant chosen by the user. */
void NEUROsetPenal (char *penalty);

/* Estimates for each mouse the graph that best represents the  */
/* observed data in the first and third parts of the experiment */
/* for a fixed penalty value and method (1, 2 and 3) of         */
/* computing the posterior probability.                         */
void NEURObestGraph (char *dataPath, char *outPath);

/* Estimates for each mouse the graph that best represents the  */
/* observed data in the first and third parts of the experiment */
/* considering different penalty values and the different       */
/* methods (1, 2 and 3) of computing the posterior probability. */
void NEUROpenalAnalysis (char *dataPath, char *outPath);
```

A.7 Neuro.c

Implementação do método de Monte Carlo com Cadeia de Markov no espaço dos grafos, representando as conexões entre neurônios.

```
/**  ************************************************  **/
/**  Implementation of Markov Chain Monte Carlo on graphs   **/
/**  representing neuronal connectivity. Given the          **/
/**  experimental data, it generates a Markov Chain, on an  **/
/**  undirected graph space, whose limit distribution is    **/
/**  given by the posterior probability 'P(g|X)'. This is   **/
/**  done via Monte Carlo method with Metropolis            **/
/**  algorithm.                                             **/
/**  The generated graphs are stored in a skip list         **/
/**  symbol-table whose nodes have a counter that is        **/
/**  incremented each time an item of an existing node is   **/
/**  searched (to avoid the need to store each generated    **/
/**  graph, i.e., only distinct graphs are stored). The     **/
/**  use of skip list structure has also the advantage      **/
/**  that search, insertion and removal are O(log N).       **/
/**  ************************************************  **/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "Utils.h"
#include "Item.h"
#include "ST.h"
#include "Neuro.h"
```

```
#define Trange 0.01 /* time range = 0.01 s = 10 ms */
#define Tstep 100 /* consider 1 Trange window after Tstep=100*Trange */
#define Jij 1.0 /* interaction "energy" */
#define Nsteps 100000 /* # Monte Carlo steps */

/* Structure to store the considered spikes read from the    */
/* data file, where 'label' is the neuron label and 'spikes' */
/* is a vector of spikes in the time interval considered.    */
typedef struct NEUROspk spkInfo;
struct  NEUROspk { char label[5]; int *spikes; };

static long MEM; /* available memory */
static int fixSteps; /* fixed number of MC steps option (0 or 1) */
static double penal; /* penalty constant */
static int met; /* label of the method for probability computation */
static int spkRange; /* time range of considered spikes */
static int rat; /* mouse id */
static int Nneuron; /* number of neurons */
static int Nedges; /* maximum number of edges */
static int part; /* part of the experiment (1 or 3) */
static char region[6]; /* brain region */


/* ******************************************************** */
/* Sets the available memory for graphs storage.            */
void NEUROsetMem (char *freeMem)
{
   MEM = atol (freeMem);

} /* NEUROsetMem */


/* ******************************************************** */
/* Sets the option for a fixed number of Monte Carlo steps  */
/* ('1' means that a fixed number was chosen by the user and */
/* '0' represents the opposite).                            */
void NEUROsetMCsteps (char *steps)
{
   fixSteps = atoi (steps);

} /* NEUROsetMCsteps */


/* ******************************************************** */
/* Sets the brain region chosen for study.                  */
void NEUROsetRegion (char *rg)
{
   region[0] = '\0';
   copy (region, rg);

} /* NEUROsetRegion */


/* ******************************************************** */
/* Sets the mouse chosen for study.                         */
void NEUROsetMouse (char *mouse)
{
   rat = atoi (mouse);

} /* NEUROsetMouse */
```

```
/* ******************************************************** */
/* Sets the method for posterior probability computation.    */
/* If first method was chosen (method = '1'), than it is     */
/* considered: <Xi|Xj>=1 if Xi=1 and Xj=1.                    */
/* If second method was chosen (method = '2'), than it is    */
/* considered: <Xi|Xj>=1 if Xi=Xj or <Xi|Xj>=0 (if Xi!=Xj).  */
/* If third method was chosen (method = '3'), than it is     */
/* considered: <Xi|Xj>=1 if Xi=Xj or <Xi|Xj>=-1 (if Xi!=Xj). */
void NEUROsetMethod (char *method)
{
   met = atoi (method);

} /* NEUROsetMethod */


/* ******************************************************** */
/* Sets the penalty constant chosen by the user.             */
void NEUROsetPenal (char *penalty)
{
   penal = atof (penalty);

} /* NEUROsetPenal */


/* ******************************************************** */
/* Prints the adjacency matrix representation of the key     */
/* 'max' at 'std' file. The key is a graph in vector         */
/* representation and, in order to maintain the coherence    */
/* with other parts of the program, the rule is fill the     */
/* elements below the main diagonal in row-major order.      */
static void printAdjMatrix (FILE *std, Key max, spkInfo *tkt)
{
   int i, j;
   Key adjMatrix;

   /* Creates the matrix. */
   adjMatrix = UTILmalloc (Nneuron * Nneuron * sizeof (adjMatrix));

   /* Initializes the matrix with "0s". */
   for (i = 0; i < Nneuron; i++)
      for (j = 0; j < Nneuron; j++)
         adjMatrix[idx2d(i,j,Nneuron)] = '0';

   /*  *** This is an important "trick" of the algorithm!! ***  */
   /* Copies the values of vector 'max' to the symmetric matrix */
   /* following the rule of filling the elements below the      */
   /* main diagonal in row-major order. Note that, 'i > j' and  */
   /* 'idxAdj(i,j)' will give the correct index (see Item.h).   */
   for (i = 1; i < Nneuron; i++)
      for (j = 0; j < i; j++) {
         adjMatrix[idx2d(i,j,Nneuron)] = max[idxAdj(i,j)];
         adjMatrix[idx2d(j,i,Nneuron)] = max[idxAdj(i,j)];
      }

   /* Prints the matrix in 'std' file with the   */
   /* electrodes labels at each column and line. */
   fprintf (std, "    ");
   for (i = 0; i < Nneuron; i++)
      fprintf (std, " %s ", tkt[i].label);
   fprintf (std, "\n");
   for (i = 0; i < Nneuron; i++) {
      fprintf (std, "%s ", tkt[i].label);
      for (j = 0; j < Nneuron; j++)
```

```
         fprintf (std, "  %c  ", adjMatrix[idx2d(i,j,Nneuron)]);
      fprintf (std, "\n");
   }

} /* printAdjMatrix */


/* ******************************************************* */
/* Receives an output file name 'outName' and prints a lot   */
/* of relevant information about the run (the code is quite  */
/* self explanatory).                                        */
static void output (char *outName, spkInfo *tkt, unsigned long accept,
                    unsigned long steps, unsigned long maxMCsteps)
{
   int i;
   FILE *out; /* file for general output */

   out = UTILfopen (outName, "a"); /* opens the general output file */

   fprintf (out, "** Mouse %d - %s - part %d **\n", rat, region, part);
   fprintf (out, "\nNumber of neurons: %d", Nneuron);
   fprintf (out, "\nNeurons labels: ");
   for (i = 0; i < Nneuron; i++)
      fprintf (out, " %s ", tkt[i].label);
   fprintf (out, "\nMC steps: %lu", steps);
   fprintf (out, "\nMaximum allowed MC steps: %lu", maxMCsteps);
   fprintf (out, "\nTotal graphs counted: %lu", STtotalCount());
   fprintf (out, "\nPenalty constant: %.5f", penal);
   fprintf (out, "\nDistinct graphs: %d", STcount());

   /* *** This might interest you!!! *** */
   /* For those who do not believe that this program really */
   /* works please uncomment the following line code. It    */
   /* will print all distinct graphs at the output file,    */
   /* but be aware that it can be a lot of graphs.          */
   /* STsort (stdout, ITEMshow); */

   fprintf (out, "\nAccepted graphs: %lu", accept);
   fprintf (out, "\nMost representative graph counter = %lu", STmaxCont());
   fprintf (out, "\nMost representative graph probability = %.5f",
            1.0*STmaxCont()/(steps + 1));
   fprintf (out, "\nMost representative graph (vectorial form):\n");
   STshowMaxItem (out); /* show in vectorial form */
   fprintf (out, "\nMost representative graph (adjacency matrix):\n");
   printAdjMatrix (out, STmaxItem(), tkt); /* show adjacency matrix */
   fprintf (out, "\n\n");

   fclose (out); /* closes the general output file */

} /* output */


/* ******************************************************* */
/* Receives an output file name 'outName' and prints the     */
/* adjacency matrix of the highest score graph in this file. */
static void outputAdjM (char *outName, spkInfo *tkt)
{

   FILE *out; /* file for adjacency matrix output */

   out = UTILfopen (outName, "w"); /* opens the file */
   printAdjMatrix (out, STmaxItem(), tkt); /* prints the adjacency matrix */
   fclose (out); /* closes the file */
```

```
} /* outputAdj */


/* ******************************************************** */
/* Creates the Monte Carlo starting state randomly (the      */
/* probability of having each edge is 0.5).                 */
static Key MCinit ()
{
   int i;
   double u;
   Key ini;

   /* Allocates the graph. */
   ini = UTILmalloc ((Nedges + 1) * sizeof (ini));

   /* Decides (randomly) if each possible edge exists. */
   for (i = 0; i < Nedges; i++) {
      u =  1.0 * rand() / RAND_MAX; /* pseudo-random u ~ Unif[0,1] */
      if (u < 0.5) /* probability = 0.5 */
         ini[i] = '0'; /* no edge */
      else
         ini[i] = '1'; /* with edge */
   }
   ini[i] = '\0'; /* terminating null character */

   return ini;

} /* MCinit */


/* ******************************************************** */
/* Given two observed data 'obs1' and 'obs2', computes their */
/* their "interaction" according to the method chosen:       */
/*    1:  <Xi|Xj>=1 if Xi=1 and Xj=1.                        */
/*    2:  <Xi|Xj>=1 if Xi=Xj or <Xi|Xj>=0 (if Xi!=Xj).       */
/*    3:  <Xi|Xj>=1 if Xi=Xj or <Xi|Xj>=-1 (if Xi!=Xj).      */
static int gibbsEnMethod (int obs1, int obs2)
{
   if (met == 1) /* <Xi|Xj>=1 if Xi=1 and Xj=1. */
      return (obs1 * obs2);
   if (met == 2) { /* <Xi|Xj>=1 if Xi=Xj or <Xi|Xj>=0 (if Xi!=Xj). */
      if (obs1 == obs2)
         return 1;
      return 0;
   }
   /* <Xi|Xj>=1 if Xi=Xj or <Xi|Xj>=-1 (if Xi!=Xj). */
   if (obs1 == obs2)
      return 1;
   return -1;

} /* gibbsEnMethod */


/* ******************************************************** */
/* Computes the "interaction energies" between all possible  */
/* neighbors.                                               */
static double *gibbsEn (spkInfo *tkt)
{
   int i, j, k, g;
   double *Vij;

   /* Allocates the "interaction energy" vector. */
   Vij = UTILmalloc (Nedges * sizeof (double));
```

```
   /*  *** This is an important "trick" of the algorithm!! ***  */
   /* The index of the "interaction energy" vector must have    */
   /* the same rule of the graph index when writing the         */
   /* adjacency matrix. Here this rule is fill the elements     */
   /* below the main diagonal in row-major order.               */
   for (g = 0, i = 1; i < Nneuron; i++)
      for (j = 0; j < i; j++) {
          /* Scan the spikes. */
         for (Vij[g] = 0.0, k = 0; k < spkRange; k++)
            Vij[g] += gibbsEnMethod (tkt[i].spikes[k], tkt[j].spikes[k]);
         Vij[++g] *= Jij; /* "interaction energy" between neighbors */
      }

   return Vij;

} /* *gibbsEn */


/* ******************************************************** */
/* Receives a candidate for a new state (the changed 'edge') */
/* and returns '1' if it is accepted or '0' if not. Uses the */
/* acceptance probability proposed originally by Metropolis  */
/* et al (1954) with a symmetric sampling kernel ('Qij=Qji') */
/*           'a(i,j) = min {1, P(g_j|X)/P(g_i|X)}'           */
/* where 'g_i' is the current state, 'g_j' is the candidate  */
/* and 'P(g_i|X)' is the posterior probability. It is easy   */
/* to see that the ratio 'P(g_j|X)/P(g_i|X)}' is given by    */
/* one of the following options:                             */
/*   1. 'exp(penal*spkRange-Jij*<Xi|Xj>)' if 'g_j' has one   */
/*      less edge than 'g_i' (an edge has been removed).     */
/*   2. 'exp(Jij*<Xi|Xj>-penal*spkRange)' if 'g_j' has one   */
/*      more edge than 'g_i' (an edge has been inserted).    */
/* The information about which edge has been changed comes   */
/* from the integer 'edge': if 'edge<0' it means that an     */
/* edge has been removed, otherwise an edge has been         */
/* inserted.                                                 */
/* Obs.: the addition of '1' in the 'edge' value was         */
/* necessary to avoid mistake when the index is zero (see    */
/* 'ITEMrandIdx' function at Item.c).                        */
static int metropolis (double *gibbsVij, int edge)
{
   double u;

   /* Generates u ~ Unif[0,1]. */
   u = 1.0 * rand() / RAND_MAX;

   /* An edge has been removed. */
   if (edge < 0) {
      if (exp (penal * spkRange - gibbsVij[-edge-1]) > u)
         return 1; /* candidate accepted */
   }
   else { /* An edge has been inserted. */
      if (exp (gibbsVij[edge-1] - penal * spkRange) > u)
         return 1; /* candidate accepted */
   }
   return 0; /* candidate rejected */

} /* metropolis */
```

```
/* ******************************************************** */
/* Given an initial state 'gr', it computes 'Nsteps' Monte   */
/* Carlo steps without including the generated graphs into   */
/* the accepted graphs list. This "thermalization steps" are */
/* used reduce the importance of choosing the initial state. */
/* After all the steps, the resulting graph 'gr' is taken as */
/* the initial state of the Monte Carlo.                     */
static void mcThermSteps (double *gibbsVij, Key gr)
{
   int i;
   int edge;

   /* 'Nsteps' Monte Carlo steps. */
   for (i = 0; i < Nsteps; i++) {

      /* Chooses a random edge to change. */
      edge = ITEMrandIdx (gr);

      /* metropolis = 1 if the candidate is accepted. */
      if (metropolis (gibbsVij, edge) == 1) {

         /* Changes the edge. */
         if (edge >= 0)
            gr[edge-1] = '1'; /* inserts the edge */
         else
            gr[-edge-1] = '0'; /* removes the edge */

      }

   } /* for (i = 0; i ... */

} /* mcThermSteps */


/* ******************************************************** */
/* Given an initial state 'gr', it computes 'Nsteps' Monte   */
/* Carlo steps. Each accepted state is included into the     */
/* accepted graphs list (skip list symbol-table - see ST.c). */
/* Returns the number of accepted graphs.                    */
static int mcSteps (double *gibbsVij, Key *gr)
{
   int i;
   int accept; /* # of accepted graphs */
   int edge; /* index of the changed edge */
   Item item; /* skip list object */
   Key grProx; /* candidate graph */

   /* 'Nsteps' Monte Carlo steps. */
   for (accept = 0, i = 0; i < Nsteps; i++) {

      /* Chooses a random edge to change. */
      edge = ITEMrandIdx (*gr);

      /* metropolis = 1 if the candidate is accepted. */
      if (metropolis (gibbsVij, edge) == 1) {

         accept++; /* one more graph */

         /* Allocates memory for the new graph. */
         grProx = UTILmalloc ((Nedges + 1 ) * sizeof (grProx));
         copy (grProx, *gr); /* grProx = *gr*/
         ITEMgenerator (grProx, edge); /* changes the edge */
```

```
         /* Searches for 'grProx' at graphs list. Returns its   */
         /* pointer and increments its counter if it was found, */
         /* otherwise returns 'NULLitem' (see ST.c).            */
         *gr = STsearch(grProx);

         if (*gr == NULLitem) { /* there is no grProx at graphs list */
            key(item) = grProx;
            STinsert (item); /* adds to the list */
            *gr = grProx; /* grProx is the current state */
         }
         else /* grProx exists already at graphs list */
            free (grProx);

      }
      else /* candidate rejected */
         *gr = STsearch(*gr); /* increments 'gr' counter */

   } /* for (i = 0; i ... */

   return accept;

} /* mcSteps */


/* ******************************************************** */
/* Generates a Markov Chain, on an undirected graph space,   */
/* whose limit distribution is given by the posterior        */
/* probability 'P(g|X)'. This is done via Monte Carlo method */
/* with Metropolis algorithm.                                */
static void mcmc (int type, char *outPath, spkInfo *tkt, double *logPP)
{
   int i, length;
   unsigned long steps, accept; /* MC steps counter, # accepted graphs */
   Item item; /* skip list object */
   Key gr; /* current graph */
   double *Vij; /* "interaction energy": Vij = Jij * <Xi|Xj> */
   unsigned long maxMCsteps; /* maximum MC steps */
   char *outName; /* file name for general output */

   /* Initializes variables. */
   Nedges = Nneuron * (Nneuron - 1) / 2; /* # of edges */
   gr = MCinit (); /* first graph generated randomly */

   /* File name for general output. */
   length = size (outPath);
   outName = UTILmalloc ((length + 30) * sizeof (char));
   outName[0] = '\0';

   /* Maximum Monte Carlo steps. */
   if (fixSteps) /* chosen by the user */
      maxMCsteps = MEM;
   else /* depends on the memory available */
      maxMCsteps = 3 * (MEM / (2 * (Nedges + 35)));

   /* Computes all possible "interaction energies". */
   Vij = gibbsEn (tkt);

   /* "Thermalization" steps. */
   mcThermSteps (Vij, gr);

   /* Creates and initializes the skip list. */
   key(item) = gr;
   STinit (); /* creates the skip list */
```

```
   STinsert (item); /* insert 'gr' in the skip list */
   accept = 1; /* # of accepted graphs */

   /* Monte Carlo steps. */
   for (steps = 0; steps < maxMCsteps; steps += Nsteps)
      accept += mcSteps (Vij, &gr);

   /* Computes some results. */
   if (type == 0) { /* 'penalty analysis' run */
      logPP[0] = logPP[1] = logPP[2] = 0.0;
      gr = STmaxItem();
      for (i = 0; i < Nedges; i++) {
         /* Non-normalized log-posterior probability (with penalty). */
         logPP[0] += (gr[i] - '0') * (Vij[i] - penal * spkRange);
         /* Non-normalized log-posterior probability (without penalty). */
         logPP[1] += (gr[i] - '0') * Vij[i];
      }
      /* Empirical probability (obtained from the Monte Carlo). */
      logPP[2] = 1.0*STmaxCont()/(steps + 1);

      /* Writes output data. */
      sprintf (outName, "%soutputM%d%sp%dMet%d.dat",
               outPath, rat, region, part, met);
      output (outName, tkt, accept, steps, maxMCsteps);

      /* Frees memory. */
      free (Vij);
      STfree ();
      free (outName);

   }
   else { /* 'best graph' run */
      /* Writes output data. */
      sprintf (outName, "%soutputM%d%sp%dMet%d.dat",
               outPath, rat, region, part, met);
      output (outName, tkt, accept, steps, maxMCsteps);

      /* Output of the adjacency matrix. */
      outName[0] = '\0';
      sprintf (outName, "%sadjM%d%sp%dMet%d.dat",
               outPath, rat, region, part, met);
      outputAdjM (outName, tkt);

      /* Frees memory. */
      free (Vij);
      STfree ();
      free (outName);
   }

} /* mcmc */


/* ******************************************************* */
/* Receives the name of a file containing the observed     */
/* spikes of a given neuron and the minimum time to be     */
/* considered. Reads the file and generates a vector of    */
/* spikes where an element is equal to '1' if there was a  */
/* spike at a 'Trange' time window after each 'Tstep' time */
/* interval or '0' if not. This vector is then returned.   */
static int *spkRead (char *spkPath, double min)
{
   int j;
   int *tktSPK; /* vector of spikes */
```

```
   double time, aux;
   FILE *spk; /* file with spikes */

   /* Opens the file with spikes. */
   spk = UTILfopen (spkPath, "r");

   /* Allocates the vector of spikes. */
   tktSPK = UTILmalloc (spkRange * sizeof (int));

   /* Scans the file considering 1 'Trange' window after each 'Tstep'. */
   aux = 0.0;
   time = min;
   for (j = 0; j < spkRange; j++) {

      /* Reads the file until it reaches the next starting point. */
      while (aux < time)
         UTILcheckFscan (fscanf (spk, "%lf", &aux), spkPath);

      tktSPK[j] = 0;
      while (aux < time + Trange) { /* checks if there is a spike */
         UTILcheckFscan (fscanf (spk, "%lf", &aux), spkPath);
         tktSPK[j] = 1;
      }

      /* Jumps the 'Tstep' time interval. */
      time += Tstep * Trange;

   } /* for (j = 0; j < ... */

   fclose (spk); /* closes the file */

   return tktSPK;

} /* *spkRead */


/* ******************************************************** */
/* Frees memory of the stored spikes data (i.e. neuron label */
/* and its spikes in the time interval considered).          */
static void spkFree (spkInfo *tkt)
{
   int i;

   for (i = 0; i < Nneuron; i++)
      free (tkt[i].spikes);
   free (tkt);

} /* spkFree */


/* ******************************************************** */
/* Function for penalty analysis. It calls the 'mcmc'       */
/* function (Markov Chain Monte Carlo) with different       */
/* penalty values and with the different methods (1, 2 and  */
/* 3) for computing the posterior probability. It receives a */
/* path 'outPath' for writing the output files, the penalty  */
/* interval ('ini' to 'end') to be considered and the rate   */
/* of change of the penalty. Creates 3 files containing the  */
/* penalty value versus:                                    */
/*    penal1: Non-normalized log-posterior probability      */
/*            with penalty.                                 */
/*    penal2: Non-normalized log-posterior probability      */
/*            without penalty.                              */
```

```
/*    penal3: Empirical probability obtained from the        */
/*            Monte Carlo.                                   */
static void penalMetMCMC (char *outPath, spkInfo *tkt,
                          double ini, double end, double delta)
{
   int length, cont;
   double *logPP; /* log-posterior probability */
   char *outName1, *outName2, *outName3; /* names of the output files */
   FILE *out1, *out2, *out3; /* files for output */

   /* Initializes variables. */
   logPP = UTILmalloc (3 * sizeof (double)); /* log-posterior probability */
   length = size (outPath);
   outName1 = UTILmalloc ((length + 26) * sizeof (char));
   outName2 = UTILmalloc ((length + 26) * sizeof (char));
   outName3 = UTILmalloc ((length + 26) * sizeof (char));
   outName1[0] = '\0';
   outName2[0] = '\0';
   outName3[0] = '\0';
   sprintf (outName1, "%spenal1M%d%sp%dMet%d.dat",
            outPath, rat, region, part, met);
   sprintf (outName2, "%spenal2M%d%sp%dMet%d.dat",
            outPath, rat, region, part, met);
   sprintf (outName3, "%spenal3M%d%sp%dMet%d.dat",
            outPath, rat, region, part, met);

   /* Some user interaction. */
   printf ("\n  method %d", met);
   setvbuf (stdout, NULL, _IONBF, 0); /* print now! */

   /* Opens the output files. */
   out1 = UTILfopen (outName1, "w");
   out2 = UTILfopen (outName2, "w");
   out3 = UTILfopen (outName3, "w");

   /* Looping over penalty values. */
   for (cont = 0, penal = ini; penal < end; penal += delta) {
      /* Markov Chain Monte Carlo. */
      mcmc (0, outPath, tkt, logPP);

      /* Writes results. */
      fprintf (out1, "%.7f  %.10f\n", penal, logPP[0]);
      fflush (out1); /* print now! */
      fprintf (out2, "%.7f  %.10f\n", penal, logPP[1]);
      fflush (out2); /* print now! */
      fprintf (out3, "%.7f  %.10f\n", penal, logPP[2]);
      fflush (out3); /* print now! */

      /* Just to avoid unnecessary computation. */
      if (logPP[0] < 0.0000001) {
         cont++;
         if (cont > 10)
            break;
      }

   } /* for (cont = 0, penal = ... */

   /* Closes the output files. */
   fclose (out1);
   fclose (out2);
   fclose (out3);

   /* Frees memory. */
```

```
   free (outName1);
   free (outName2);
   free (outName3);
   free (logPP);

} /* penalMetMCMC */


/* ******************************************************** */
/* Function for penalty analysis. It receives the name of a  */
/* file 'dataFile' containing summarized information about   */
/* the experimental data from the chosen brain region (like  */
/* (the mouse ID, number of neurons, start and end times of  */
/* observation, label and path to the observed spikes of     */
/* each neuron. Receives also a path 'outPath' for writing   */
/* the output files. Then for each mouse in 'dataFile' it    */
/* gets its spikes information and computes the Markov Chain */
/* Monte Carlo with different methods for computing the      */
/* posterior probability and with different penalty values   */
/* by calling the 'penalMetMCMC' function.                   */
static void neuroPenal (char *dataFile, char *outPath)
{
   int i, mouse;
   double min, max; /* 'min' and 'max' spike times in a set */
   spkInfo *tkt; /* vector with electrode label and spikes */
   char spkPath[150]; /* path to the spikes data */
   char aux1[5], aux2[150];
   FILE *summary; /* file containing a summary of data */

   /* Opens the input file with data paths. */
   summary = UTILfopen (dataFile, "r");

   /* Looping over the mouses. */
   while (fscanf (summary, "%d%d", &mouse, &Nneuron) == 2) {

      /* Reads the 'min' and 'max' values of spike time in the set. */
      UTILcheckFscan (fscanf (summary, "%lf%lf", &min, &max), dataFile);

      if (mouse == rat) { /* mouse chosen for study */

         /* Time range considered (discounting the first and the last ~5min). */
         min += 300.0;
         max -= 300.0;
         spkRange = (int) (max - min);

         /* Computes the Monte Carlo only if time is greater than ~30 min. */
         if (spkRange > 1000) {

            /* Allocates a vector for neuron's label and spikes. */
            tkt = UTILmalloc (Nneuron * sizeof (spkInfo));

            /* Looping over the neurons. */
            for (i = 0; i < Nneuron; i++) {

               /* Reads the neuron's label and the path to the spikes data. */
               UTILcheckFscan (fscanf (summary, "%s%s",
                                       tkt[i].label, spkPath), dataFile);

               /* Gets the spikes. */
               tkt[i].spikes = spkRead (spkPath, min);
            }

            /* Computes the MCMC with different methods for computing the */
```

```
            /* posterior probability and with different penalty values.   */

            /* First method: <Xi|Xj>=1 if Xi=1 and Xj=1. */
            met = 1;
            penalMetMCMC (outPath, tkt, 0.00001, 0.1, 0.0001);

            /* Second method: <Xi|Xj>=1 if Xi=Xj or <Xi|Xj>=0 (if Xi!=Xj). */
            met = 2;
            penalMetMCMC (outPath, tkt, 0.6, 1.0, 0.001);

            /* Third method: <Xi|Xj>=1 if Xi=Xj or <Xi|Xj>=-1 (if Xi!=Xj). */
            met = 3;
            penalMetMCMC (outPath, tkt, 0.2, 1.0, 0.001);

            /* Frees memory. */
            spkFree (tkt);
         }
         else {
            /* Looping over the neurons. */
            for (i = 0; i < Nneuron; i++)
               /* Reads the neuron's label and the path to the spikes data. */
               UTILcheckFscan (fscanf (summary, "%s%s", aux1, aux2), dataFile);

            /* Some user interaction. */
            printf (" (insufficient data)");
            setvbuf (stdout, NULL, _IONBF, 0); /* print now! */
         }
      }
      else { /* this is not the mouse chosen for study */
         /* Looping over the neurons. */
         for (i = 0; i < Nneuron; i++)
            /* Reads the neuron's label and the path to the spikes data. */
            UTILcheckFscan (fscanf (summary, "%s%s", aux1, aux2), dataFile);
      }

   } /* while */

   /* Closes the input file. */
   fclose (summary);

} /* neuroPenal */


/* ******************************************************** */
/* Function for best graph computation. It receives the name */
/* of a file 'dataFile' containing summarized information    */
/* about the experimental data from the chosen brain region  */
/* (like the mouse ID, number of neurons, start and end      */
/* times of observation, label and path to the observed      */
/* spikes of each neuron. Receives also a path 'outPath' for */
/* writing the output files. Then for each mouse in          */
/* 'dataFile' it gets its spikes information and computes    */
/* the Markov Chain Monte Carlo.                             */
static void neuro (char *dataFile, char *outPath)
{
   int i, mouse;
   double min, max; /* 'min' and 'max' spike times in a set */
   double *aux = &min;
   spkInfo *tkt; /* vector with electrode label and spikes */
   char spkPath[150]; /* path to the spikes data */
   char aux1[5], aux2[150];
   FILE *summary; /* file containing a summary of data */
```

```
   /* Opens the input file with data paths. */
   summary = UTILfopen (dataFile, "r");

   /* Looping over the mouses. */
   while (fscanf (summary, "%d%d", &mouse, &Nneuron) == 2) {

      /* Reads the 'min' and 'max' values of spike time in the set. */
      UTILcheckFscan (fscanf (summary, "%lf%lf", &min, &max), dataFile);

      if (mouse == rat) { /* mouse chosen for study */

         /* Time range considered (discounting the first and the last ~5min). */
         min += 300.0;
         max -= 300.0;
         spkRange = (int) (max - min); /* time range of considered spikes */

         /* Computes the Monte Carlo only if time is greater than ~30 min. */
         if (spkRange > 1000) {

            /* Allocates a vector for neuron's label and spikes. */
            tkt = UTILmalloc (Nneuron * sizeof (spkInfo));

            /* Looping over the neurons. */
            for (i = 0; i < Nneuron; i++) {

               /* Reads the neuron's label and the path to the spikes data. */
               UTILcheckFscan (fscanf (summary, "%s%s",
                                       tkt[i].label, spkPath), dataFile);

               /* Gets the spikes. */
               tkt[i].spikes = spkRead (spkPath, min);
            }

            /* Markov Chain Monte Carlo. */
            mcmc (1, outPath, tkt, aux);

            /* Frees memory. */
            spkFree (tkt);
         }
         else {
            /* Loop over the neurons. */
            for (i = 0; i < Nneuron; i++)
               /* Reads the neuron's label and the path to the spikes data. */
               UTILcheckFscan (fscanf (summary, "%s%s", aux1, aux2), dataFile);

            /* Some user interaction. */
            printf (" (insufficient data)");
            setvbuf (stdout, NULL, _IONBF, 0); /* print now! */
         }
      }
      else { /* this is not the mouse chosen for study */
         /* Looping over the neurons. */
         for (i = 0; i < Nneuron; i++)
            /* Reads the neuron's label and the path to the spikes data. */
            UTILcheckFscan (fscanf (summary, "%s%s", aux1, aux2), dataFile);
      }

   } /* while */

   /* Closes the input file. */
   fclose (summary);

} /* neuro */
```

```
/* ******************************************************** */
/* Given the path to the summarized data 'dataPath' and the  */
/* path 'outPath' for writing the output files, it           */
/* estimates, for the chosen mouse at the chosen region, the */
/* graph that best represents the observed data in the first */
/* and third parts of the experiment (before and after the   */
/* mouse got in touch with geometric objects) for a fixed    */
/* penalty value and method (1, 2 and 3) of computing the    */
/* posterior probability.                                    */
void NEURObestGraph (char *dataPath, char *outPath)
{
   int length;
   char *file;

   length = size (dataPath);
   file = UTILmalloc ((length + 20) * sizeof (char));

   /* First part of the experiment (before contacts). */
   printf ("\n Mouse %d - %s region - part 1", rat, region);
   setvbuf (stdout, NULL, _IONBF, 0); /* print now! */
   file[0] = '\0';
   sprintf (file, "%s%smousesP1.dat", dataPath, region);
   part = 1; /* first part of the experiment */
   neuro (file, outPath);

   /* Third part of the experiment (after contacts). */
   printf ("\n\n Mouse %d - %s region - part 3", rat, region);
   setvbuf (stdout, NULL, _IONBF, 0); /* print now! */
   file[0] = '\0';
   sprintf (file, "%s%smousesP3.dat", dataPath, region);
   part = 3; /* third part of the experiment */
   neuro (file, outPath);

   /* Frees memory. */
   free (file);

} /* NEURObestGraph */


/* ******************************************************** */
/* Given the path to the summarized data 'dataPath' and the  */
/* path 'outPath' for writing the output files, it           */
/* estimates, for the chosen mouse at the chosen region, the */
/* graph that best represents the observed data in the first */
/* and third parts of the experiment (before and after the   */
/* mouse got in touch with geometric objects) considering    */
/* different penalty values and the different methods (1, 2  */
/* and 3) of computing the posterior probability.            */
void NEUROpenalAnalysis (char *dataPath, char *outPath)
{
   int length;
   char *file;

   length = size (dataPath);
   file = UTILmalloc ((length + 20) * sizeof (char));

   /* First part of the experiment (before contacts). */
   printf ("\n Mouse %d - %s region - part 1", rat, region);
   setvbuf (stdout, NULL, _IONBF, 0); /* print now! */
   file[0] = '\0';
   sprintf (file, "%s%smousesP1.dat", dataPath, region);
   part = 1; /* first part of the experiment */
   neuroPenal (file, outPath);
```

```
   /* Third part of the experiment (after contacts). */
   printf ("\n\n Mouse %d - %s region - part 3", rat, region);
   setvbuf (stdout, NULL, _IONBF, 0); /* print now! */
   file[0] = '\0';
   sprintf (file, "%s%smousesP3.dat", dataPath, region);
   part = 3; /* third part of the experiment */
   neuroPenal (file, outPath);

   /* Frees memory. */
   free (file);

} /* NEUROpenalAnalysis */
```

A.8 ST.h

Interface de tipo abstrato de dados para tabelas de símbolos, cujos itens possuem um contador que é incrementado cada vez que a busca por um item já existente é realizada.

```
/**  ************************************************  **/
/**  This code is based on R. Sedgewick "Algorithms in C    **/
/**  Parts 1-4", 3rd Edition, Addison-Wesley (1998).        **/
/**  ************************************************  **/
/**  Abstract data type interface for symbol-table whose    **/
/**  items have a counter that is incremented each time     **/
/**  the item is searched.                                  **/
/**  ************************************************  **/

/* Initializes. */
void STinit ();

/* Adds a new item. */
void STinsert (Item);

/* Searches an item with a given key. */
Item STsearch (Key);

/* Removes an item. */
void STdelete (Item);

/* Returns the "int"-th smallest item. */
Item STselect (int);

/* Visit the items in the order of their keys (calling */
/* a procedure passed as an argument for each item).   */
void STsort (FILE *std, void (*visit)(FILE *std, Item));

/* Return the quantity of different items. */
int STcount ();

/* Returns the key of the highest score item. */
Key STmaxItem ();

/* Prints at 'std' the key of the highest score item. */
void STshowMaxItem (FILE *std);

/* Returns the score of the highest score item. */
unsigned long STmaxCont ();
```

```
/* Returns the sum of the scores of all items. */
unsigned long STtotalCount ();

/* Frees memory (destroys the symbol-table). */
void STfree ();
```

A.9 ST.c

Implementação de tipo abstrato de dados para tabelas de símbolos "skip list", cujos itens possuem um contador que é incrementado cada vez que a busca por um item já existente é realizada (para evitar a necessidade de armazenar todo novo item gerado, i.e., somente itens distintos são armazenados).

```
/**  ************************************************  **/
/**  This code is based on R. Sedgewick "Algorithms in C    **/
/**  Parts 1-4", 3rd Edition, Addison-Wesley (1998).        **/
/**  ************************************************  **/
/**  Abstract data type implementation for skip list        **/
/**  symbol-table whose nodes have a counter that is        **/
/**  incremented each time an item of an existing node is   **/
/**  searched.                                              **/
/**  The skip list data structure was developed by Pugh in  **/
/**  1990. It is an ordered linked list where each node     **/
/**  contains a variable number of links (set randomly).    **/
/**  There by, during a search it skips through large       **/
/**  portions of the list at a time due to the extra links  **/
/**  in the nodes. This characteristics provides a search,  **/
/**  insertion and removal of O(log N).                     **/
/**  ************************************************  **/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Item.h"
#include "Utils.h"
#include "ST.h"

#define lgNmax 30 /* maximum number of levels */

/* Each node has an item, an array of links with length 'sz' */
/* and a counter 'cont' that is incremented each time the    */
/* item is searched.                                         */
typedef struct STnode *link;
struct STnode{ Item item; link *next; int sz; unsigned long cont; };

static link head;
static int N; /* number of items in the list */
static int lgN; /* actual number of levels */
static link maxItem; /* element with higher frequency */


/* ************************************************** */
/* Creates a new 'STnode' with 'item' content and 'k' links, */
/* and returns its pointer.                                  */
static link NEW (Item item, int k)
{
   int i;
```

```
   link x;

   x = UTILmalloc (sizeof *x); /* allocates the 'STnode' */
   x->next = UTILmalloc (k * sizeof (link)); /* array of links */

   x->item = item; /* content */
   x->sz = k; /* number of links */
   for (i = 0; i < k; i++)
      x->next[i] = NULL; /* initializes the links with NULL pointer */
   x->cont = 1; /* initializes the counter */

   /* If 'maxItem' has not yet been initialized... */
   if (maxItem == NULLitem)
      maxItem = x; /* initializes the maxItem */

   return x;

} /* NEW */


/* ******************************************************** */
/* Initializes the skip list. Initializes the item's counter */
/* 'N' and the actual number of levels 'lgN' with 0, creates */
/* the head node with 'NULLitem' content and with 'lgNmax'   */
/* links and initializes the pointer to the element with     */
/* higher frequency with 'NULLitem'.                         */
void STinit ()
{
   N = 0; /* item's counter */
   lgN = 0; /* actual number of levels */
   head = NEW (NULLitem, lgNmax); /* creates the head node */
   maxItem = NULLitem; /* initializes the maxItem pointer */

} /* STinit */


/* ******************************************************** */
/* Generates a pseudo-random number 'i' (number of links of  */
/* a new node which will be inserted) with probability       */
/* '1/j^i' (where here 'j' is set equal 2).                  */
/* Obs.: for large skip lists, the memory space occupied by  */
/* the extra pointers becomes non-trivial and, in this case, */
/* the user can change the algorithm below by replacing      */
/* 'j = 2' by 'j = 3' and 'j = j * 2' by 'j = j * 3'.        */
static int randX ()
{
   int i, j, t;

   /* Generates a pseudo-random integer in [0,RAND_MAX]. */
   t = rand ();

   /* Probability: 1/2, 1/2^2, 1/2^3, ... */
   for (i = 1, j = 2; i < lgNmax; i++, j = j * 2)
      if (t > RAND_MAX / j)
         break;

   if (i > lgN)
      lgN = i; /* updates the actual number of levels */

   return i;

} /* randX */
```

```
/* ******************************************************** */
/* Recursive function that inserts 'x' after 't' at level    */
/* 'k', preserving the lexicographic order.                 */
static void insertR (link t, link x, int k)
{
   Key v;

   /* Gets 'x' key. */
   v = key(x->item);
   if ((t->next[k] == NULL) || less (v, key (t->next[k]->item))) {
      if (k < x->sz) { /* dancing links... */
         x->next[k] = t->next[k];
         t->next[k] = x;
      }
      if (k == 0) /* the insertion ended */
         return;
      insertR (t, x, k-1); /* tail recursion */
      return;
   }

   /* 'x' is after 't->next[k]' */
   insertR (t->next[k], x, k);

} /* insertR */


/* ******************************************************** */
/* Adds a new item (envelope function to be exported).      */
void STinsert (Item item)
{
   /* The insertion starts from the 'head'   */
   /* and at the actual highest level 'lgN'. */
   insertR (head, NEW (item, randX()), lgN);
   N++; /* one more item in the list */

} /* STinsert */


/* ******************************************************** */
/* Recursive function that searches an item with key 'v'    */
/* after 't' (taking into account that the list is in       */
/* lexicographic order) at level 'k'. The recursion proceeds */
/* as follows: it moves to the next node in the list on     */
/* level 'k' if its key is smaller than the search key 'v'  */
/* or down to level 'k-1' if its key is not smaller.        */
static Item searchR (link t, Key v, int k)
{
   if (t->next[k] == NULL) { /* end of the list of level 'k' */
      if (k == 0) /* not found */
         return NULLitem;
      return searchR (t, v, k-1); /* down to level 'k-1' */
   }
   if (eq (v, key (t->next[k]->item))) { /* it was found */
      t->next[k]->cont++; /* increments the item counter */
      /* Checks if the item counter exceeded the actual maxItem. */
      if (maxItem->cont < t->next[k]->cont)
               maxItem = t->next[k];
      return t->next[k]->item;
   }
   if (less (v, key (t->next[k]->item))) { /* 'v' is smaller */
      if (k == 0) /* not found */
         return NULLitem;
```

```
      return searchR (t, v, k-1); /* down to level 'k-1' */
   }
   return searchR (t->next[k], v, k); /* searches from the next item */

} /* searchR */


/* ******************************************************** */
/* Searches an item with a given key 'v' (envelope function  */
/* to be exported).                                          */
Item STsearch (Key v)
{
   /* The search begins from the 'head' and */
   /* at the actual highest level 'lgN'.    */
   return searchR (head, v, lgN);

} /* STsearch */


/* ******************************************************** */
/* Returns a pointer to the highest score element of the     */
/* list.                                                     */
/* Obs.: it is only to be used to correct the 'maxItem'      */
/* pointer (which may occur, for example, at 'deleteR'       */
/* function).                                                */
static link searchMaxItem ()
{
   link newMax, t;

   t = head;
   if (t->next[0] == NULL) /* empty list */
      return NULLitem;

   newMax = t->next[0];
   t = t->next[0];
   while (t->next[0] != NULL) { /* scans the list */
      t = t->next[0];
      if (t->cont > newMax->cont)
         newMax = t;
   }

   return newMax;

} /* searchMaxItem */

/* ******************************************************** */
/* Recursive function that removes the item with key 'v'. It */
/* proceeds quite similar to the 'searchR' function. If the  */
/* item found has a counter greater than 1 this function     */
/* only decrements its counter, otherwise it is necessary to */
/* unlink it at each level that we find a link to it and     */
/* free the entire node when it reaches the bottom level. If */
/* the item is the highest score item, the 'maxItem' pointer */
/* have to be fixed by calling 'searchMaxItem' function.     */
static void deleteR (link t, Key v, int k)
{
   link x = t->next[k];
   if (t->next[k] == NULL) { /* end of the list of level 'k' */
      if (k > 0)
         deleteR(t, v, k-1); /* down to level 'k-1' */
   }
   else if (eq (v, key (t->next[k]->item))) {
      if (t->next[k]->cont > 1) { /* only decrements the item's counter */
```

```
         t->next[k]->cont--;
         if (maxItem == t->next[k])
            maxItem = searchMaxItem (); /* fixes maxItem pointer */
         return;
      }
      t->next[k] = x->next[k]; /* unlink at level k */
      if (k == 0) { /* reached the bottom level */
         if (maxItem == x) {
            free (x->item); /* frees the node's item */
            free (x->next); /* frees the node's vector of links */
            free (x); /* frees the node */
            maxItem = searchMaxItem (); /* fixes maxItem pointer */
         }
         else {
            free (x->item); /* frees the node's item */
            free (x->next); /* frees the node's vector of links */
            free (x); /* frees the node */
         }
         return;
      }
      deleteR (t, v, k-1); /* down to level 'k-1' */
   }
   else if (less (v, key (t->next[k]->item))) { /* 'v' is smaller */
      if (k > 0)
         deleteR (t, v, k-1); /* down to level 'k-1' */
   }
   else deleteR (t->next[k], v, k); /* 'v' is greater */

} /* deleteR */


/* ******************************************************** */
/* Removes an item with key 'v' (envelope function to be     */
/* exported).                                                */
void STdelete (Key v)
{
   /* It starts looking for the item to be deleted from */
   /* the 'head' and at the actual highest level 'lgN'. */
   deleteR (head, v, lgN);
   N--; /* one less item in the list */

} /* STdelete */


/* ******************************************************** */
/* Returns the 'k'-th smallest item or returns 'NULLitem' if */
/* the list is has less then 'k' items.                      */
Item STselect (int k)
{
   int i;
   link t = head;

   for (i = 0; i < k && t->next[0] != NULL; i++)
      t = t->next[0];

   /* If there is less than 'k' items in the list. */
   if (i != k)
      return NULLitem;

   return t->item;

} /* STselect */
```

```
/* ******************************************************** */
/* Visit the items in the order of their keys (calling a     */
/* procedure passed as an argument for each item).           */
void STsort (FILE *std, void (*visit)(FILE *std, Item))
{
   link t = head;

   while (t->next[0] != NULL) {
      t = t->next[0];
      visit(std, t->item); /* call the procedure */
   }

} /* STsort */


/* ******************************************************** */
/* Return the quantity of different items.                   */
int STcount ()
{
   return N;

} /* STcount */


/* ******************************************************** */
/* Returns the key of the highest score item.                */
Key STmaxItem ()
{
   return (key(maxItem->item));

} /* STmaxItem */


/* ******************************************************** */
/* Prints at 'std' the key of the highest score item.        */
void STshowMaxItem (FILE *std)
{
   ITEMshow (std, key(maxItem->item));

} /* STshowMaxItem */


/* ******************************************************** */
/* Returns the score of the highest score item.              */
unsigned long STmaxCont ()
{
   return maxItem->cont;

} /* STmaxCont */


/* ******************************************************** */
/* Returns the sum of the scores of all items.               */
unsigned long STtotalCount ()
{
   unsigned long count = 0;
   link t = head;

   while (t->next[0] != NULL) {
      t = t->next[0];
      count += t->cont;
   }
```

```
   return count;

} /* STtotalCount */


/* ******************************************************** */
/* Frees memory of all nodes (destroys the symbol-table).    */
void STfree ()
{
   link t;

   while (head->next[0] != NULL) {
      t = head->next[0];
      head->next[0] = t->next[0];
      free (t->item); /* frees the node's item */
      free (t->next); /* frees the node's vector of links */
      free (t); /* frees the node */
   }

   free (head); /* finally, frees the head */

} /* STfree */
```

A.10 Item.h

Interface de tipo abstrato de dados.

```
/**  ************************************************  **/
/**  This code is based on R. Sedgewick "Algorithms in C   **/
/**  Parts 1-4", 3rd Edition, Addison-Wesley (1998).       **/
/**  ************************************************  **/
/**  Abstract data type interface for graph items, which   **/
/**  are represented by strings containing '0's and '1's.  **/
/**  Therefore, the associated operations for an item are  **/
/**  those for strings and a few others.                   **/
/**  One of the most common representations for a graphs   **/
/**  is the adjacent matrix, i.e., a square matrix where   **/
/**  an element 'a_ij = 1' indicates that there is an edge **/
/**  from vertex 'i' to vertex 'j' and 'a_ij = 0'          **/
/**  indicates that these vertices are not connected.      **/
/**  However, considering that the items are undirected    **/
/**  graphs, the adjacent matrix is symmetric and it is    **/
/**  necessary only to know the elements above (or below)  **/
/**  the main diagonal. Thus, these items (undirected      **/
/**  graphs) are strings that represents the elements      **/
/**  above the main diagonal. The way of indexing a string **/
/**  V to match the adjacent matrix AM is given by:        **/
/**     for a_ij in AM:  if i < j,  V[i+1+(j+1)*(j-2)/2]   **/
/**                      otherwise, V[j+1+(i+1)*(i-2)/2]   **/
/**  ************************************************  **/

typedef char *Item;
typedef char *Key;

#define key(A) (A)
#define eq(A, B) (strcmp(A,B)==0)
#define less(A, B) (strcmp(A,B)<0)
#define size(A) (strlen(A))
#define copy(A, B) (strcpy(A,B))
```

```
#define comp(A, B) (strcmp(A,B))
#define NULLitem (NULL)
#define idxAdj(i, j) (((i)<(j)) ? (i+1+(j+1)*(j-2)/2) : (j+1+(i+1)*(i-2)/2))

/* Reads a key from standard input. */
int ITEMscan (Item *);

/* Prints at 'std' the key of an item. */
void ITEMshow (FILE *std, Item);

/* Changes the 'idx' element from an item. */
void ITEMgenerator (char *new, int idx);

/* Picks a random element from an item and returns its index */
/* if the element is '0' or the negative value of the index. */
int ITEMrandIdx (char *item);
```

A.11 Item.c

Implementação de tipo abstrato de dados para grafos (i.e. palavras contendo 0s e 1s).

```
/**  ************************************************  **/
/**  This code is based on R. Sedgewick "Algorithms in C    **/
/**  Parts 1-4", 3rd Edition, Addison-Wesley (1998).        **/
/**  ************************************************  **/
/**  Abstract data type implementation for graph items,     **/
/**  which are represented by strings containing '0's and   **/
/**  '1's.                                                  **/
/**  ************************************************  **/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Item.h"
#include "Utils.h"


/* ******************************************************* */
/* Reads a key from standard input and copies it into 'x'.   */
int ITEMscan (char **x)
{
   int t;
   char buf[1000];

   t = scanf ("%s", buf);
   *x = UTILmalloc (size (buf) * sizeof (char));
   (*x)[0] = '\0';
   copy (*x, buf);

   return t;

} /* ITEMscan */


/* ******************************************************* */
/* Prints at 'std' the key of an item 'x'.                   */
void ITEMshow (FILE *std, char *x)
{
   fprintf (std, "%s\n", x);
} /* ITEMshow */
```

```
/* ******************************************************* */
/* Performs the unary operator "not" (logical negation) on   */
/* an element index '|idx|-1' from an item 'new'.            */
/* If 'idx > 0' indicates that the element 'new[idx-1]' is   */
/* '0' and then changes it to '1'. If 'idx < 0' indicates    */
/* that the element 'new[-idx-1]' is '1' and then changes it */
/* to '0'.                                                   */
void ITEMgenerator (char *new, int idx)
{
   if (idx > 0)
      new[idx-1] = '1';
   else
      new[-idx-1] = '0';

} /* ITEMgenerator */


/* ******************************************************* */
/* Generates a pseudo-random index. Receives the length      */
/* 'delta' of a partition, generates a pseudo-random number  */
/* 'u' uniformly distributed in [0,1] and checks to which    */
/* partition 'part' this 'u' belongs (that's the index).     */
static int geraIdx (double delta)
{
   int part;
   double u;

   /* Generates u ~ Unif[0,1]. */
   u = 1.0 * rand() / RAND_MAX;

   /* Checks where is 'u'. */
   for (part = 1; delta * part < 1.0; part++)
      if (u < delta * part)
         return (part - 1);
   return (part - 1);

} /* geraIdx */


/* ******************************************************* */
/* Picks a random element from an item and returns its index */
/* plus 1 ('idx + 1') if the element is '0' or the negative  */
/* value 'idx - 1'. The addition (or subtraction) of '1' is  */
/* necessary to avoid mistake when the index is zero.        */
int ITEMrandIdx (char *item)
{
   int idx;
   double delta;

   /* Computes the length of a partition of [0,1]. */
   delta = 1.0 / size(item);

   /* Generates a pseudo-random index. */
   idx = geraIdx (delta);

   if (item[idx] == '0')
      return (idx + 1);

   return (-idx - 1);

} /* ITEMrandIdx */
```

A.12 Utils.h

Interface para versões alternativas de funções bem conhecidas das bibliotecas “stdio.h” e “stdlib.h” para evitar repetição de código.

```
/**  ************************************************  **/
/**  Interface for alternative versions of well known  **/
/**  functions from 'stdio.h' and 'stdlib.h' to avoid code  **/
/**  repetition.                                       **/
/**  ************************************************  **/

/* Allocates a block of bytes if there are     */
/* enough memory, otherwise exits the program. */
void *UTILmalloc (unsigned int nbytes);

/* Change the size of a block of bytes if there */
/* are enough memory or exits the program.      */
void *UTILrealloc (void *ptr1, unsigned int nbytes);

/* Opens the file named 'filename' in order to    */
/* execute a 'mode' operation and verifies error. */
FILE *UTILfopen (const char *filename, const char *mode);

/* Verifies the returned value of a call to the 'fscanf' */
/* function to read data from the file named 'filename'. */
void UTILcheckFscan (int info, const char *filename);
```

A.13 Utils.c

Implementações alternativas de funções bem conhecidas das bibliotecas “stdio.h” e “stdlib.h” para evitar repetição de código.

```
/**  ************************************************  **/
/**  Implementation of alternative versions of well known  **/
/**  functions from 'stdio.h' and 'stdlib.h' to avoid code  **/
/**  repetition.                                       **/
/**  ************************************************  **/

#include <stdio.h>
#include <stdlib.h>
#include "Utils.h"


/* ******************************************************* */
/* Allocates a block of bytes if there are enough memory.    */
/* Otherwise returns an error message and exits the program. */
void *UTILmalloc (unsigned int nbytes)
{
   void *ptr;
   ptr = malloc (nbytes);

   if (ptr == NULL) {
      fprintf (stderr, "\n Insufficient memory.\n");
      exit(EXIT_FAILURE);
   }
```

```
   return ptr;

} /* UTILmalloc */


/* ******************************************************** */
/* Reallocates a block of bytes pointed by 'ptr' and change  */
/* its size to 'nbytes' if there are enough memory.          */
/* Otherwise returns an error message and exits the program. */
void *UTILrealloc (void *ptr1, unsigned int nbytes)
{
   void *ptr2;
   ptr2 = realloc (ptr1, nbytes);

   if (ptr2 == NULL) {
      fprintf (stderr, "\n Insufficient memory.\n");
      exit (EXIT_FAILURE);
   }

   return ptr2;

} /* UTILrealloc */


/* ******************************************************** */
/* Opens the file named 'filename' in order to execute an    */
/* operation specified by 'mode' (operations of the 'fopen'  */
/* function from 'stdio.h') and verifies error.              */
FILE *UTILfopen (const char *filename, const char *mode)
{
   FILE *pfile;

   pfile = fopen (filename, mode);
   if (pfile == NULL) {
      fprintf (stderr, "\n Error: Unable to open the file '%s'!\n\n", filename);
      exit (EXIT_FAILURE);
   }

   return pfile;

} /* UTILfopen */


/* ******************************************************** */
/* Receives the integer 'info' returned from a call to the   */
/* 'fscanf' function to read data from the file named        */
/* 'filename' and checks if 'info = EOF', which indicates    */
/* that an input failure happened before any data could be   */
/* successfully read.                                        */
void UTILcheckFscan (int info, const char *filename)
{
   if (info == EOF) {
      fprintf (stderr, "\n Error: Unable to read the file '%s'!\n\n", filename);
      exit (EXIT_FAILURE);
   }

} /* UTILcheckFscan */
```

Referências Bibliográficas

[1] S. Ribeiro, X. Shi, M. Engelhard, Y. Zhou, H. Zhang, D. Gervasoni, S.-C. Lin, K. Wada, N. A. M. Lemos e M. A. L. Nicolelis. *Frontiers in Neuroscience* **1**(1), 43–55 (2007).

[2] E. R. Kandel, J. H. Schwartz e T. M. Jessell, editores. *Principles Of Neural Science*. McGraw-Hill, 4ª edição, (2000).

[3] M. F. Bear, B. W. Connors e M. A. Paradiso. *Neuroscience - Exploring The Brain*. Lippincott Williams & Wilkins, 3ª edição, (2006).

[4] M. A. M. Freire, E. Morya, J. Faber, J. R. Santos, J. S. Guimaraes, N. A. M. Lemos, K. Sameshima, A. Pereira, S. Ribeiro e M. A. L. Nicolelis. *PLoS ONE* **6**(11), e27554 (2011).

[5] R. Sedgewick. *Algorithms In C - Parts1-4*. Addison Wesley, 3ª edição, (1998).

[6] A. Galves, C. Galves, N. L. Garcia e F. Leonardi. *Ann. Appl. Stat.* **6**(1), 186–209 (2012).

[7] W. Pugh. *Comm. of the ACM* **33**(6), 668–676 (1990).