Modelowanie i eksploracja sieci neuronów biologicznych w GENESIS

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UMCS

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf".
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

Uniwersytet Marii Curie-Skłodowskiej
Wydział Matematyki, Fizyki i Informatyki
Instytut Informatyki

Modelowanie i eksploracja sieci neuronów biologicznych w GENESIS

Grzegorz M. Wójcik

Lublin 2012

Instytut Informatyki UMCS
Lublin 2012

Grzegorz M. Wójcik

Modelowanie i eksploracja sieci neuronów biologicznych w GENESIS

Recenzent: Andrzej Bobyk

Opracowanie techniczne: Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl.

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Druk FIGARO Group Sp. z o.o. z siedzibą w Rykach
ul. Warszawska 10
08-500 Ryki
www: www.figaro.pl
–

ISBN: 978-83-62773-30-5

Spis treści

Wstęp

W ciągu kilku dekad czyli od momentu kiedy komputery stały się powszechne w uprawianiu nauki – modelowanie i symulacja komputerowa stały się jednymi z głównych narzędzi i metod badawczych wykorzystywanych we wszystkich niemal dyscyplinach eksperymentalnych. W szczególności duże znaczenie ma prowadzenie eksperymentów *in computo* w naukach biologicznych. Często z powodów technologicznych, finansowych a nawet etycznych zdecydowanie łatwiej i taniej jest podglądać zjawiska i zachowania w modelu niż w żywym organizmie.

Ostatnie lata to również czas burzliwego rozwoju neuronauk, w tym neuronauki obliczeniowej zwanej neurocybernetyką [1, 2]. Od dawna bowiem istniały pokusy modelowania skomplikowanych układów biologicznych, jednak dopiero komputery dały badaczom możliwość urzeczywistnienia marzeń. Tak też neurocybernetyka stała się nauką wkraczającą na pola innych dyscyplin, takich jak psychologia, sztuczna inteligencja, medycyna [3], biologia, systemy ekspertowe, nowe metody obliczeniowe, neurologia i wiele innych [4, 5, 6, 7, 8, 9, 10, 11].

Do najbardziej skomplikowanych symulacji biologicznych należą badania zagadnień związanych z funkcjonowaniem mózgu lub wybranych jego fragmentów [12, 13, 14, 15, 3, 16, 17, 18]. W przypadku człowieka, mózg zawierający sto miliardów komórek nerwowych, z których każda jest połączona średnio z tysiącami innych stanowi najbardziej skomplikowany układ w znanym Wszechświecie. Dlatego modelowanie i symulacja tego typu obiektów stanowią nie lada wyzwanie.

Przedstawiono tu kurs programowania w środowisku GENESIS będącym obecnie jednym z najlepszych narzędzi modelowania realistycznych biologicznie komórek nerwowych [19]. Książka przeznaczona jest dla studentów specjalności neuroinformatycznych i pokrewnych. Stanowi też wprowadzenie do modelowania struktur biologicznych dla doktorantów podejmujących problematykę neuronauki obliczeniowej. Wraz z pakietem GENESIS dostarczany jest anglojęzyczny podręcznik „The Book of GENESIS" [19], który ze jednak względu na wysoki poziom zaawansowania może stanowić odstraszającą barierę dla informatyków pragnących od podstaw drążyć pro-

blematykę biologii obliczeniowej. Dlatego niniejszy skrypt powinien stanowić materiał przygotowawczy dla przyszłych neurocybernetyków i łagodnie wprowadzić ich w tematykę zagadnienia.

W pierwszym rozdziale omówiono ideę modelowania neuronów biologicznych.

Rozdział drugi prowadzi czytelnika przez skomplikowany proces kompilacji ze źródeł oraz instalacji środowiska GENESIS w systemie operacyjnym Ubuntu Linux.

W trzecim rozdziale omówiono podstawy języka skryptowego GENESIS, podstawowe polecenia powłoki i strukturę języka.

Rozdział czwarty przedstawia podstawy środowiska graficznego XODUS.

W piątym rozdziale zaprezentowano na prostych przykładach modelowanie pojedynczych komórek nerwowych, w szóstym zaś – prostych sieci neuronowych.

Rozdział siódmy omawia prostą metodę wizualizacji symulowanych sieci z wykorzystaniem bibliotek XODUS.

W rozdziale ósmym omówiono proces kompilacji wersji równoległej – PGENESIS – symulatora na komputerze wielordzeniowym.

W rozdziale ósmym zaprezentowano metody równoleglizacji dużych sieci neuronowych w PGENESIS.

Rozdział dziesiąty prezentuje sposoby modyfikowania kodu źródłowego GENESIS w celu uzyskiwania nowych funkcjonalności.

Na końcu skryptu znajdują się dodatki, w których opisano podstawowe modele, nad którymi w ostatnich latach autor prowadził swoje badania. Zamieszczono również streszczenia najważniejszych artykułów opublikowanych jako rezultat tych badań (Dodatek A). Ze względu na używaną terminologię zdecydowano się na zamieszczenie streszczeń w języku angielskim. Pozwoli to tym czytelnikom, którzy zdecydują się na kontynuowanie tych lub podobnych badań na zapoznanie się z powszechnym w literaturze anglojęzycznej i związanym z tematem neuroobliczeń słownictwem (Dodatek B). W Dodatku C zamieszczono zdjęcia, które z ważniejszych lub mniej ważnych powodów mają dla autora istotne znaczenie.

Rozdział 1

Modelowanie komórek nerwowych

1.1. Wprowadzenie

W tym rozdziale przedstawimy biologiczne podstawy związane z budową neuronów. Minimum wiedzy z zakresu biologii i neurofizjologii układu nerwowego jest niezbędne do zrozumienia dalszych rozdziałów skryptu. Opiszemy zasadę działania pojedynczego neuronu, rodzaje komórek nerwowych oraz model Hodgkina–Huxleya, według którego projektowane są numeryczne modele neuronów oraz sieci neuronowych.

1.2. Neuron biologiczny

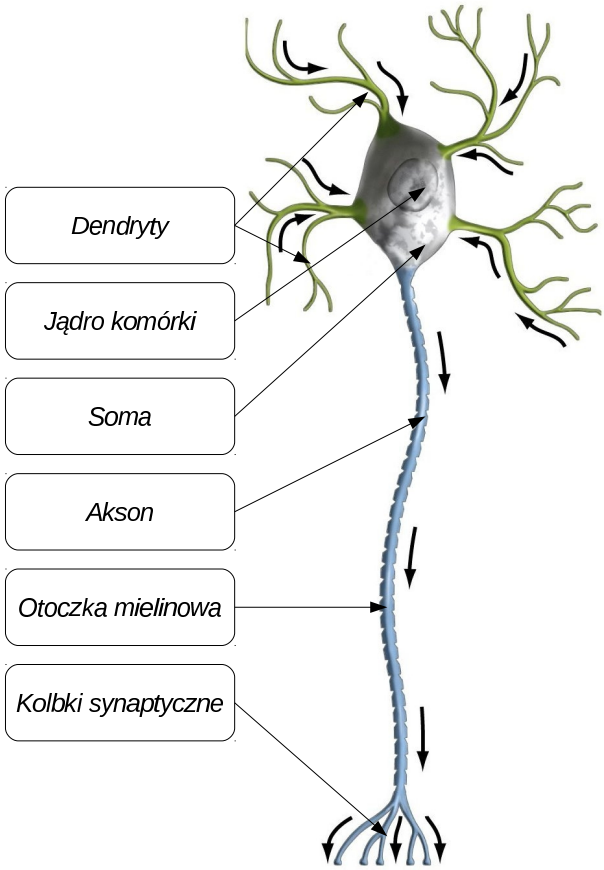
Tkanka nerwowa u ssaków, w tym u naczelnych rozsiana jest w całym ciele, jednak największa jej część znajduje się w jamie czaszki (mózgowie) i w kanale kręgowym (rdzeń kręgowy) tworząc ośrodkowy układ nerwowy. Pozostała tkanka tworzy obwodowy układ nerwowy.

Podstawowym elementem budującym układ nerwowy jest neuron czyli komórka nerwowa [1]. Budowę typowego neuronu przedstawiono na Rys. 1.1 [20]. W dużym skrócie zasadę działania neuronu można przedstawić w bardzo schematyczny i powtarzalny sposób. Najważniejszą częścią neuronu jest ciało komórki (soma), w którym znajduje się jądro komórkowe i inne wewnątrzkomórkowe struktury. Posługując się terminologią komputerową jądro komórki możemy przyrównać do procesora sterującego pracą neuronu [20].

Do ciała komórki wchodzą dendryty (od kilku do kilkunastu tysięcy, niekiedy przypominające drzewiaste struktury – stąd wywodzi się często spotykana nazwa drzewo dendrytyczne). Dendryty stanowią wejścia do neuronu. To do nich dochodzą sygnały od innych neuronów, które analizowane przez jądro decydują o elektrycznym zachowaniu komórki. Skupiska ciał neuronów nazywane są istotą szarą (ang. *gray matter*) stąd potoczna nazwa - „szare komórki" [20].

Z kolei z ciała neuronu wychodzi akson, przewód przypominający swoją zasadą działania drut. Jeżeli w pewnych okolicznościach sygnał napływający przez dendryty do ciała komórki jest wystarczająco silny to we wzgórku aksonu wywoływane zostaje zaburzenie prowadzące do powstania tak zwanego piku potencjału czynnościowego. Zmieniający się potencjał przepływa wzdłuż aksonu (z prędkością od 4 do 120 m/s u ludzi – dzięki otoczce mielinowej wokół aksonu, u bardziej prymitywnych zwierząt bez otoczki mielino-

[1] Opis neuronu biologicznego oraz fragmenty opisu modelu Hodgkina–Huxleya zaczerpnięto z monografii autora zatytułowanej „Obliczenia płynowe w modelowaniu mózgu" [20]. W tym miejscu autor pragnie złożyć serdeczne podziękowania Panu Andrzejowi Langowi z Akademickiej Oficyny Wydawniczej Exit za wyrażenie zgody na przytoczenie niniejszego opisu.



Rysunek 1.1. Schemat neuronu biologicznego (za: [20])

wej od 0,5 do 4 m/s) dochodząc do często wielokrotnie rozgałęzionego końca, który może być połączony za pośrednictwem synaps z dendrytami innych neuronów. Najdłuższe aksony u człowieka mają półtora metra długości (ciągną się np. od końców palców do kręgosłupa, a dzięki temu że nie tracimy czasu na połączeniach z innymi komórkami – możemy stosunkowo szybko cofnąć rękę w chwili oparzenia), jedne z najgrubszych (ok. 1 mm średnicy, można z łatwością wbić w taki neuron elektrody i badać przepływ prądu) posiada kałamarnica olbrzymia (ang. Giant Squid, łac. Giant loligo), dzięki czemu udało się stworzyć empirycznie doskonały, opisywane w następnej sekcji model dynamiki elektrycznej neuronu [20].

Sam przepływ potencjału wzdłuż włókna aksonu utrzymywany jest przez działanie tak zwanych pomp jonowych, które w sprytny sposób zarządzają przepychaniem dodatnich jonów sodu i potasu oraz ujemnych jonów chloru w obrębie neuronu i jego otoczenia. Istota biała (ang. white matter) zbudowana jest z aksonów. Istota szara jest więc ośrodkiem przetwarzania informacji, istota biała stanowi jedynie drogi nerwowe [20].

Mechanizm połączenia komórek nerwowych związany jest z bardzo skomplikowanym procesem neurochemicznym polegającym na przesyłaniu różnego rodzaju neuroprzekaźników pomiędzy kolbkami synaptycznymi tak zwanego neuronu presynaptycznego i postsynaptycznego. Upraszczając nieco sprawę możemy przyjąć, że nie każdy neuroprzekaźnik pasuje do każdej kolbki synaptycznej, podobnie jak nie każdy klucz pasuje do każdego zamka. Pewne neuroprzekaźniki powodują wzmocnienie sygnału (potrzeba wtedy mniej neuroprzekaźnika do wywołania reakcji we wzgórku aksonu neuronu docelowego), inne z kolei mają charakter hamujący, jeszcze inne (takie jak dostarczane do organizmu przez narkotyki) – przez analogię – wyważają drzwi rozrywając zamki. Wiele namacalnych wręcz zachowań takich jak uczucie pobudzenia „po kawie", ospałość, stan zakochania, upojenia alkoholowego lub narkotycznego w dużym stopniu można wyjaśnić na drodze analizy wzajemnego oddziaływania neuronów i wydzielanych przez nie neuroprzekaźników [20].

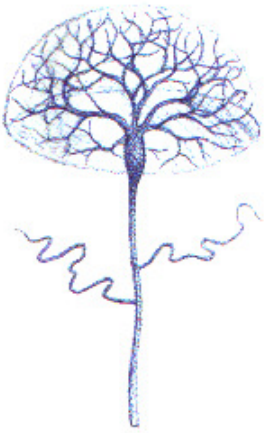
1.3. Rodzaje komórek nerwowych

W różnych partiach mózgu, a w szczególności w korze mózgowej, znajdują się komórki o zróżnicowanej budowie, a co za tym idzie odmiennych funkcjach i właściwościach. Często różnice fizjologiczne komórek należy uwzględniać przy budowie modelu. Istnieje wiele modeli neuronu. Jednym z najważniejszych jest prosty model neuronu progowego uwzględniający podstawowe cechy komórek. Bardziej złożony model Hodgkina-Huxleya znajduje zastosowanie w wiernych symulacjach naśladowczych neuronów.

Najprostszy pod względem formy jest neuron unipolarny, czyli jednobiegunowy (Rys. 1.2). Występuje głównie w drogach czuciowych. Jego ciało w zasadzie nie bierze udziału w przewodzeniu sygnałów. Neurony unipolarne charakteryzują się tym, że jedna wypustka dzieli się na jeden dendryt i jeden akson. Neuron bipolarny (dwubiegunowy) (Rys. 1.3) odbiera sygnały od innych komórek za pośrednictwem wielu dendrytów. W odpowiedzi na sygnał wejściowy neuron dwubiegunowy wysyła pojedynczą odpowiedź wzdłuż swojego aksonu. W korze mózgowej najwięcej jest neuronów piramidalnych (Rys. 1.4). Ich dendryty są zgrupowane w dwóch oddzielnych pęczkach, a akson może dawać odgałęzienia wsteczne do dendrytów. Neurony Golgiego (Rys. 1.5) występują w móżdżku i są odpowiedzialne za koordynację ruchów. Stanowią jedną z najbardziej skomplikowanych struktur jaką mogą stworzyć dendryty i rozgałęziony akson. Neurony gwiaździste (Rys. 1.6) z kory mózgowej są wyposażone w wypustki tworzące kulistą formę przestrzenną. Ich rola w korze polega na przetwarzaniu i wysyłaniu danych do pobliskich neuronów. W siatkówce oka istnieją komórki amokrynowe (Rys. 1.7). Nie posiadają aksonów. Przekazywane przez nie informacje mają postać niewielkich zmian napięcia błony komórkowej. Neurony kłębuszkowe (Rys. 1.8) są komórkami opuszki węchowej leżącymi tuż nad jamą nosową. Neurony Purkinjego (Rys. 1.9), jedne z największych i najbardziej skomplikowanych komórek nerwowych, są położone w zewnętrznej warstwie móżdżku, a ich aksony w jego głębi.

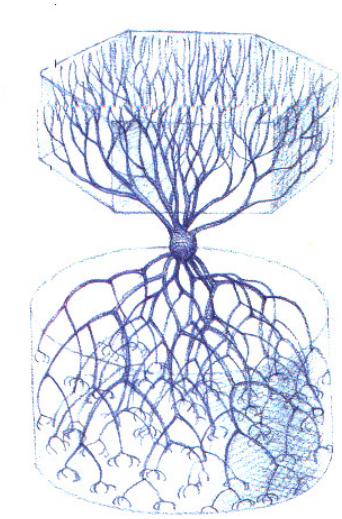
Rysunek 1.2. Neuron unipolarny (jednobiegunowy) (za: [21])

Przedstawiony podział komórek został przeprowadzony w oparciu o ich kształt i złożoność. Istnieje jednak wiele innych podziałów neuronów, np. ze względu na ich funkcje, właściwości fizykochemiczne czy połączenia jakie tworzą z innymi komórkami. Nie będziemy szczegółowo przedstawiać tej systematyki. Jedną z najważniejszych cech prawie wszystkich komórek jest zdolność tworzenia tzw. mikroobwodów. Potwierdzenie m.in. przez grupę Markrama istnienia takich struktur w korach prymitywnych ssaków jest największym wydarzeniem w neurofizjologii w ostatnich latach. Okazuje się, że komórki gromadzą się w grupach tworząc obwody zdolne do wykonywania pewnych obliczeń. Chociaż budowa poszczególnych obwodów jest zbliżona, to funkcja, którą pełnią, jest różna w zależności od miejsca występowania. Tak więc w poszczególnych polach kory wzrokowej znajdują się prawie

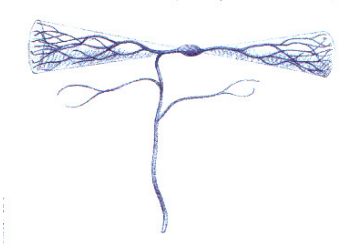


Rysunek 1.3. Neuron bipolarny (dwubiegunowy) (za: [21])

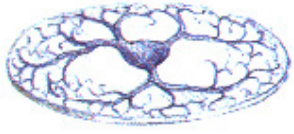
Rysunek 1.4. Neuron piramidalny (za: [21])



Rysunek 1.5. Neuron Golgiego (za: [21])

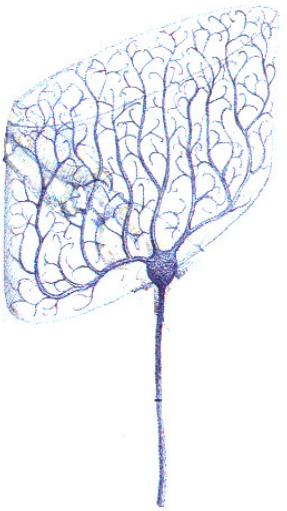


Rysunek 1.6. Neuron gwiaździsty (za: [21])



Rysunek 1.7. Komórka amokrynowa (za: [21])

Rysunek 1.8. Neuron kłębuszkowy (za: [21])



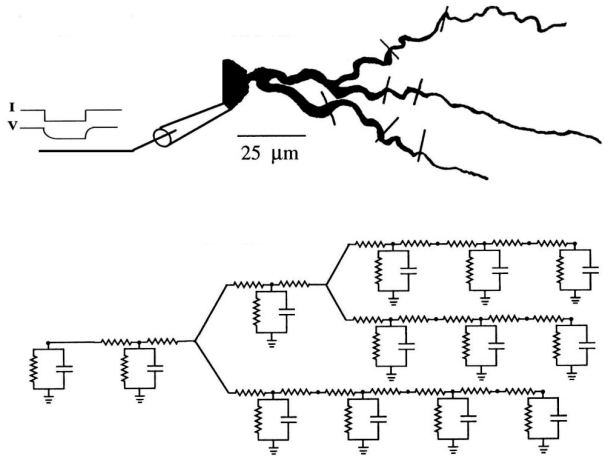
Rysunek 1.9. Neuron Purkinjego (za: [21])

identyczne mikroobwody zawierające nawet do kilku tysięcy komórek, ale zadania jakie wykonują są całkiem odmienne.

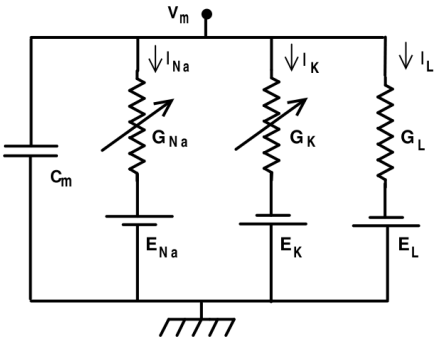
1.4. Model Hodgkina–Huxleya

Przy tworzeniu modelu rzeczywistej komórki nerwowej jedną z podstawowych trudności na jakie napotykamy jest konieczność uwzględnienia złożoności fizjologicznej neuronu. Procesy przebiegające w kanałach jonowych włókien nerwowych można opisać metodami fizyko-matematycznymi. Regułą w tworzeniu modeli jest dokonywanie podziału włókien nerwowych komórek na mniejsze, bardziej elementarne części składowe i opisywanie każdej z nich odpowiednimi równaniami. Rozwiązując układy równań dla poszczególnych części składowych komórki, można skutecznie modelować zachowanie każdego z wyodrębnionych elementów, jak również ich wzajemne oddziaływanie. Dzięki temu uzyskujemy dokładniejszy obraz symulowanego neuronu. Model naśladujący rzeczywiste zachowania neuronu biologicznego został opisany przez Hodgkina i Huxleya w 1952 r. i uhonorowany nagrodą Nobla w dziedzinie medycyny w 1963 r. W modelu tym przyjęto, iż każdy mały element neuronu można opisać przez ekwiwalentny obwód elektryczny. Fizyczne zachowanie takiego obwodu odpowiada omawianym już procesom zachodzącym w fragmentach rzeczywistych komórek. Formalnie obwód opisuje nieliniowe równanie różniczkowe. Takie podejście pozwala symulować nawet bardzo skomplikowane neurony. Istnieją modele pojedynczych komórek nerwowych, opisywane układem kilkudziesięciu tysięcy równań różniczkowych [22, 23].

Badając neurony kałamarnicy olbrzymiej Hodgkin i Huxley stworzyli teorię opisującą zachowania poszczególnych kanałów jonowych, połączeń synaptycznych i oddziaływań pomiędzy poszczególnymi, sąsiadującymi ze sobą fragmentami włókien nerwowych (np. aksonów), a także podali przedziały parametrów elektrycznych, dla których model jest najbliższy rzeczywistości. Kryterium podziału neuronu na fragmenty powinno być to, czy w ekwiwalentnych obwodach RC potencjał jest w przybliżeniu stały. Ponadto opór między poszczególnymi obwodami powinien być tak dobrany, aby zachować maksymalne podobieństwo do topologii dendrytów. Przykładowy podział komórki nerwowej i jej reprezentację obwodem RC przedstawia rys. 1.10 [19, 22, 23]. Typowy obwód elektryczny odpowiadający fragmentowi włókna neuronowego przedstawia rys.1.11. V_m reprezentuje wartość potencjału błony komórkowej1. Uziemienie odpowiada zewnętrznemu punktowi o zerowym potencjale. Wprowadzenie kondensatora o pojemności C_m jest niezbędne do modelowania zdysocjowanych roztworów przepływających przez włókna komórki. W modelu procesowi przepływu jonów odpowiada ładо-



Rysunek 1.10. Model komórki nerwowej oraz schemat jej podziału na elementy składowe (za: [19])



Rysunek 1.11. Element składowy neuronu jako obwód elektryczny. (za: [19])

wanie i rozładowywanie kondensatora. Jeżeli potencjały V'_m i V''_m są różne, to kondensator rozładowuje się przez opór R_a. Może też nastąpić przepływ prądu przez opór R'_a. Rezystor G_k o zmiennej oporności reprezentuje jedną z wielu możliwych przewodności kanałów jonowych. Przewodności te są specyficzne dla poszczególnych kombinacji jonów i neuronów. Wewnątrz i na zewnątrz komórki panują różne koncentracje jonów. Wywołane różnicą koncentracji ciśnienie osmotyczne powoduje, że jony dyfundują wzdłuż gradientu koncentracji. W obwodzie elektrycznym zjawisko to jest modelowane dzięki wprowadzeniu źródła prądu E_k. Potencjał, przy którym przepływ prądu jest zatrzymany nazywa się potencjałem równowagowym E_k. Jeżeli nie ma żadnego impulsu wchodzącego do komórki, to znaczy że komórka nie ma połączeń z innymi neuronami. Wówczas posiada ona tak zwany potencjał spoczynkowy E_{rest}, który w naszym obwodzie wynosi V_m. Opór błony komórkowej na rys. 1.11 oznaczono jako R_m . Opór ten w połączeniu ze źródłem prądu E_m odpowiada za utrzymanie zerowego potencjału w kanałach jonowych. Z kolei I_{inject} reprezentuje natężenie prądu podawanego z zewnątrz, który może pochodzić na przykład od elektrody umieszczonej w ciele komórki [22, 23].

Zgodnie z prawem Kirchhoffa potencjał Vm spełnia nieliniowe równanie różniczkowe [20]:

$$C_m \frac{dV_m}{dt} = -\sum_k I_k + I_{inj.}(t). \tag{1.1}$$

Sumowanie odbywa się po wszystkich rodzajach kanałów jonowych występujących w modelu. Dynamikę kanału upływowego reguluje niezależna od napięcia przewodność. Przewodność pozostałych kanałów zależy od napięcia i zmienia się w czasie. Przy otwartym kanale mamy do czynienia z maksymalnym możliwym przewodzeniem prądu g_{Na} i g_K. Zazwyczaj jednak część kanałów jest zablokowana, a prawdopodobieństwo zablokowania opisują dodatkowe zmienne m, n – dla kanałów sodowych, oraz h – dla kanałów potasowych. Wykorzystując te zmienne, można wyznaczyć sumaryczny prąd jonowy występujący w konkretnym obwodzie i zależność 1.1 przekształcić w główne równanie Hodgkina–Huxleya 1.2 [20]:

$$\begin{aligned} C_m \frac{dV_m}{dt} = &- [G_{Na} m^3 h (V_m - E_{Na}) \\ &+ G_K n^4 (V_m - E_K) + G_L (V_m - E_L)] + I_{inj.}(t). \end{aligned} \tag{1.2}$$

Nieznane funkcje m, n i h regulujące otwieranie kanałów jonowych można dopasować w taki sposób, by odpowiadały danym doświadczalnym [20]:

$$\frac{dm}{dt} = \alpha_m(V_m)(1 - m) - \beta_m(V_m)m, \tag{1.3}$$

$$\frac{dn}{dt} = \alpha_n(V_m)(1-n) - \beta_n(V_m)n, \tag{1.4}$$

$$\frac{dh}{dt} = \alpha_h(V_m)(1-h) - \beta_h(V_m)h. \tag{1.5}$$

Funkcje wykładnicze opisujące zachowanie α i β wyznaczono empirycznie 1.6–1.11 [20]:

$$\alpha_n(V_m) = \frac{0,01(V_m+55)}{[1-\exp(-V_m+55)/10]}, \tag{1.6}$$

$$\alpha_m(V_m) = \frac{0,1(V_m+40)}{[1-\exp(-(V_m+40)/10)]}, \tag{1.7}$$

$$\alpha_h(V_m) = 0,07\exp[-(V_m+65)/20)], \tag{1.8}$$

$$\beta_n(V_m) = 0,125\exp[-(V_m+65)/80], \tag{1.9}$$

$$\beta_m(V_m) = 4\exp[-(V_m+65)/18], \tag{1.10}$$

$$\beta_h(V_m) = \frac{1}{1+\exp[-(V_m+35)/10]}. \tag{1.11}$$

Stałe występujące we wzorach pobrano bezpośrednio z neuronów kałamarnicy olbrzymiej. W temperaturze 280 K przyjmują one wartości [20]:

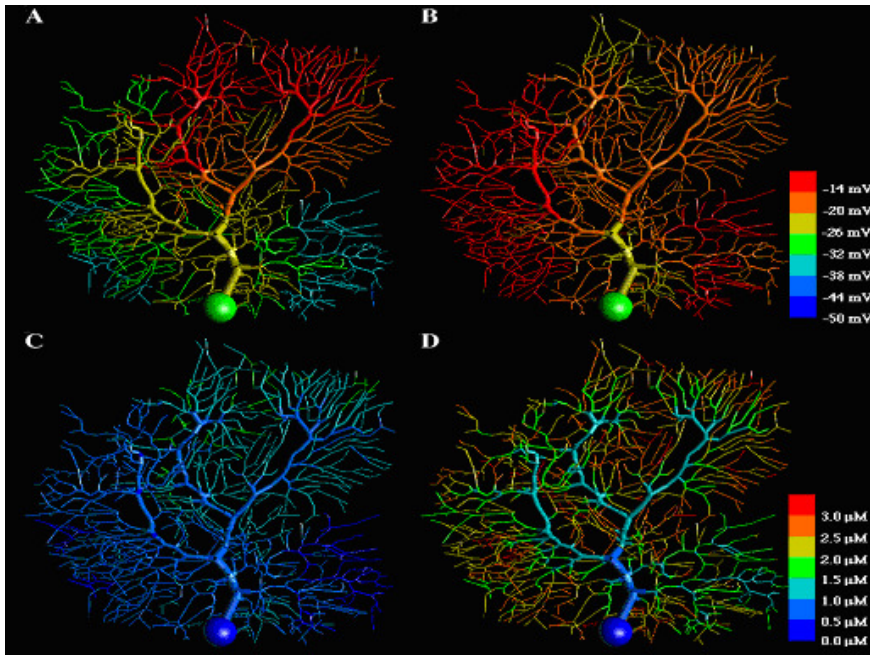
G_{Na}=120 mS/cm^2, G_K=36 mS/cm^2, G_L=0,3 mS/cm^2, E_{Na}=50 mV, E_K=–77 mV, E_L=–54 mV, C_m=1 μF/cm^2 [19, 22, 23].

Układy równań typu 1.2–1.11 można rozwiązywać tylko numerycznie. Należy pamiętać, że równania 1.1–1.11 opisują zachowanie tylko jednego, wyodrębnionego fragmentu neuronu. Układy równań modelujące całą komórkę stanowią więc opis wzajemnego oddziaływania sąsiadujących ze sobą fragmentów. Podobne równania Hodgkina–Huxleya z empirycznie wyznaczonymi parametrami odwzorowują działanie każdej synapsy i stosuje się je podczas konstruowania modeli zawierających więcej niż jedną komórkę. Stopień komplikacji równań sprawia, że model Hodgkina–Huxleya jest bardzo kosztowny obliczeniowo [23].

Mimo, że zachowania grup elementów mikroskopowych takich jak kanały jonowe lub neurotransmitery są opisywane przy pomocy wielkości makroskopowych (opór elektryczny, przewodność), to rezultaty są bardzo bliskie rzeczywistości. Świadczy to o wyjątkowej użyteczności modelu, a w wielu prowadzonych obecnie badaniach wciąż korzysta się z prac Hodgkina–Huxleya datowanych na lata pięćdziesiąte ubiegłego stulecia [20].

Przykładem modelowania pojedynczych, bardzo złożonych komórek, są prace prowadzone przez grupę Erika De Schuttera. Symulowano w nich pojedynczą komórkę Purkinjego (rys. 2.15). Drzewo dendrytów w modelowanej komórce zawiera kilkadziesiąt tysięcy elementów i wypustek. Elementy te

oddziałują elektrycznie między sobą. Przestrzenna mapa rozłożenia potencjałów w symulowanej strukturze przedstawia rys. 1.12. Pojedyncza komórka Purkinjego posiada pewne zdolności obliczeniowe. Impulsy potencjału podawane na wejście takiej komórki ulegają transformacji w drzewie jej dendrytów. Drzewo dendrytów posiada zdolności pamiętania impulsów, które wcześniej przezeń przechodziły. Zdolności te można następnie porównywać ze zdolnościami klasyfikacyjnymi SSN [24].



Rysunek 1.12. Wizualizacja komórki Purkinjego z prac Erika De Schuttera [23]

1.5. Podsumowanie

Przedstawiono zasadę działania neuronu biologicznego, podstawowe typy komórek nerwowych i teorię Hodgkina–Huxleya jako najlepszą z zaproponowanych do tej pory, stosowaną do tworzenia realistycznych biologicznie modeli pojedynczych komórek nerwowych i biologicznych sieci neuronowych.

1.6. Zadania

Zadanie 1

Narysuj schematycznie schemat neuronu i podstawowych typów komó-

rek nerwowych, a następnie zaproponuj ich podział do stworzenia modelu według teorii Hodgkina–Huxleya.

Rozdział 2

Instalacja i konfiguracja GENESIS

2.1. Wprowadzenie

Dla początkujących użytkowników systemu operacyjnego Linux instalacja i konfiguracja środowiska GENESIS może się okazać zadaniem trudnym. Celem niniejszego rozdziału jest przeprowadzenie czytelników przez proces kompilacji i instalacji symulatora w typowych konfiguracjach. Systemem operacyjnym, w którym zainstalujemy GENESIS będzie Ubuntu 10.04 LTS. Zdecydowano się na mającą już ponad rok wersję tej popularnej dystrybucji ze względu na długi czas wsparcia oferowany przez firmę Canonical. Niemniej jednak w nowszych wersjach Ubuntu opisywany proces kompilacji i instalacji przebiega praktycznie rzecz ujmując w identyczny sposób.

2.2. Przygotowanie do instalacji

Najnowszą, stabilną wersję GENESIS oznaczono numerem 2.3. Od kilku lat prowadzone są prace nad zupełnie nową pod względem filozofii implementacji wersją środowiska oznaczoną 3.0, jednak póki co dostępne źródła są w fazie testowania i w czasie pisania niniejszego podręcznika nie dysponujemy jeszcze na przykład możliwością zrównoleglenia GENESIS w wyczekiwanej wersji trzeciej. Nowa wersja, według zapowiedzi, powinna działać znacznie szybciej i oferować nieco więcej możliwości modelowania kompartmentowego (w tym Hodgkina–Huxleya). Jednak składnia poleceń i zasady budowania modeli występujące we wcześniejszych wersjach nie ulegną zmianie.

Przed instalacją GENESIS 2.3 powinniśmy zaopatrzyć się w jego źródła. Najlepiej będzie pobrać je ze strony symulatora znajdującej się pod adresem **http://sourceforge.net/projects/genesis-sim/**. Domyślnie proponowana jest kompilacja przygotowana dla systemu Mac OS X. My jednak powinniśmy podążać za łączem **Other versions - Browse all files** i pobrać plik **genesis-2.3-src.tar.gz** z katalogu **/Source/2.3 Final**. Przy okazji warto zdobyć plik z równoległą wersją GENESIS. Należy więc pobrać archiwum **pgenesis-2.3-src.tar.gz**.

Proponujemy rozpakować oba pliki do katalogu domowego użytkownika. W naszym przypadku będziemy posługiwać się katalogiem **/home/student/**. Po skopiowaniu obu archiwów do katalogu domowego w konsoli systemu wykonujemy następujące polecenia:

```
gunzip genesis-2.3-src.tar.gz
tar xvf genesis-2.3-src.tar
gunzip pgenesis-2.3-src.tar.gz
tar xvf pgenesis-2.3-src.tar
```

Po wykonaniu przytoczonych tu poleceń użytkownik powinien mieć w katalogu domowym rozpakowane źródła zarówno GENESIS w wersji szeregowej **genesis-2.3/genesis/** jak i równoległej **genesis-2.3/pgenesis/**.

2.3. Instalacja dodatków

W tym rozdziale będziemy zajmować się tylko kompilacją klasycznej tj. szeregowej wersji symulatora.

Przed przystąpieniem do pracy powinniśmy doinstalować do systemu niezbędne aplikacje i biblioteki. W przeciwnym przypadku proces kompilacji zakończy się fiaskiem. Problemy związane z niezgodnością wersji poszczególnych bibliotek w różnych dystrybucjach systemów Linux/Unix są najczęstszą przyczyną niepowodzeń instalacji GENESIS. W przypadku Ubuntu sprawę da się rozwiązać stosunkowo łatwo. Podobnie daje się to uczynić we wszelkich dystrybucjach wywodzących się z rodziny Debiana. Zatem niezbędne dodatki instalujemy wykonując w konsoli Linuksa polecenie:

```
sudo apt-get install bison flex libncurses5-dev libxt-dev
```

Jeżeli instalacja niezbędnych rozszerzeń zakończy się pomyślnie – można przystąpić do edycji pliku **Makefile**.

2.4. Edycja Makefile

W pliku **Makefile** definiujemy wszelkie niezbędne opcje kompilacji, charakterystyczne dla naszego systemu operacyjnego. Omawiany plik będzie znajdował się w katalogu **genesis-2.3/genesis/src/**, do którego najpierw powinniśmy się dostać, a następnie sam plik przygotować. Zakładając, że znajdujemy się w katalogu domowym użytkownika, wydajemy następujące polecenia:

```
cd genesis-2.3/genesis/src
```

W katalogu tym znajduje się między innymi plik **Makfile.dist**, który kopiujemy do pliku o nazwie **Makefile**, a następnie edytujemy:

```
cp Makefile.dist Makefile
gedit Makefile
```

Do edycji pliku tekstowego możemy użyć dowolnego edytora zamiast domyślnego notatnika powłoki Gnome. Czasami (na przykład instalując GENESIS na zdalnym serwerze, który nie oferuje środowiska graficznego) jesteśmy wręcz skazani na posługiwanie się edytorami pracującymi w trybie tekstowym, takimi jak **emacs**, **vi**, **vim** albo **mcedit**.

Należy odkomentować sekcję odpowiadającą systemowi operacyjnemu Linux przez usunięcie znaków # z początków każdej linii. Odpowiedni fragment pliku **Makefile** powinien zatem wyglądać następująco dla systemu 32-bitowego:

```
# ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
# System: Linux 1.2.x and up on Intel x86-based, Xeon,
#           and AMD 64-bit systems.
# Compiler: GCC
# ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~

## 2000-05-23
## Termcap/ncurses issues: The shell library makes reference
## termcap library.  Some Linux distributions have an ncurse
## which includes termcap emulation.  GENESIS appears to wor
## properly with the ncurses supplied with Red Hat Linux 5.1
## and Debian Linux (glibc2.1, egcs-2.91.66).  However, link
## ncurses is known to have resulted in core dumps in GENESI
## Linux versions.
##
## If you encounter problems linking with the TERMCAP flags
## or the GENESIS command line interface does not work, try
## following alternatives:
##
## 1) TERMCAP = -ltermcap
##
## 2) (If you are using SuSE Linux)
##    TERMCAP = /usr/lib/termcap/libtermcap.a
##
## 3) (If you are using Red Hat Linux prior to version 6.0)
##    TERMCAP = /usr/lib/libtermcap.a
##

 MACHINE=Linux
 OS=BSD

 XINCLUDE=-I/usr/X11R6/include

## Choose ONE XLIB line to uncomment:
## For 32-bit architectures
 XLIB=/usr/X11R6/lib
## For 64-bit machines, probably need /usr/X11R6/lib64 here.
```

```
# XLIB=/usr/X11R6/lib64

 CC=cc

## Old (and probably broken) gcc installations may need the
## path to cpp (preferably NOT one in /lib). If there isn't
## [link to] cpp in the same directory as 'cc', you should c
## [re]installing a newer gcc.

 CPP=cpp -P

## Choose ONE CFLAGS line to uncomment:
## For 32-bit architectures
 CFLAGS=-O2 -D__NO_MATH_INLINES
## For 64-bit architectures
# CFLAGS=-O2 -D__NO_MATH_INLINES -DLONGWORDS

 LD=ld

## !!!
## Don't uncomment the next line unless you get errors about
## libraries not being found. Setting this path may interfer
## the default (probably correct) operation of the loader, b
## 64-bit architectures may need /usr/lib64 here.
## LDFLAGS=-L/usr/lib

 RANLIB=ranlib
 AR=ar

 YACC=bison -y
 LEX=flex -l
 LEXLIB=-lfl
## Some linuxes (Gentoo?) may require -lSM and -lICE as well
 LIBS= $(LEXLIB) -lm

 TERMCAP=-lncurses
 TERMOPT=-DTERMIO -DDONT_USE_SIGIO

## end Linux 1.2.x and up on Intel x86-based systems
```

W przypadku instalacji GENESIS w systemie 64-bitowym odpowiedni fragment pliku Makefile wygląda następująco:

```
# ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
# System: Linux 1.2.x and up on Intel x86-based, Xeon,
#           and AMD 64-bit systems.
# Compiler: GCC
# ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~

## 2000-05-23
## Termcap/ncurses issues: The shell library makes reference
## termcap library.  Some Linux distributions have an ncurse
## which includes termcap emulation.  GENESIS appears to wor
## properly with the ncurses supplied with Red Hat Linux 5.1
## and Debian Linux (glibc2.1, egcs-2.91.66).  However, link
## ncurses is known to have resulted in core dumps in GENESI
## Linux versions.
##
## If you encounter problems linking with the TERMCAP flags
## or the GENESIS command line interface does not work, try
## following alternatives:
##
## 1) TERMCAP = -ltermcap
##
## 2) (If you are using SuSE Linux)
##    TERMCAP = /usr/lib/termcap/libtermcap.a
##
## 3) (If you are using Red Hat Linux prior to version 6.0)
##    TERMCAP = /usr/lib/libtermcap.a
##

 MACHINE=Linux
 OS=BSD

 XINCLUDE=-I/usr/X11R6/include

## Choose ONE XLIB line to uncomment:
## For 32-bit architectures
# XLIB=/usr/X11R6/lib
## For 64-bit machines, probably need /usr/X11R6/lib64 here.
 XLIB=/usr/X11R6/lib64

 CC=cc

## Old (and probably broken) gcc installations may need the
```

```
## path to cpp (preferably NOT one in /lib). If there isn't
## [link to] cpp in the same directory as 'cc', you should c
## [re]installing a newer gcc.

 CPP=cpp -P

## Choose ONE CFLAGS line to uncomment:
## For 32-bit architectures
# CFLAGS=-O2 -D__NO_MATH_INLINES
## For 64-bit architectures
 CFLAGS=-O2 -D__NO_MATH_INLINES -DLONGWORDS

 LD=ld

## !!!
## Don't uncomment the next line unless you get errors about
## libraries not being found. Setting this path may interfer
## the default (probably correct) operation of the loader, b
## 64-bit architectures may need /usr/lib64 here.
## LDFLAGS=-L/usr/lib

 RANLIB=ranlib
 AR=ar

 YACC=bison -y
 LEX=flex -l
 LEXLIB=-lfl
## Some linuxes (Gentoo?) may require -lSM and -lICE as well
 LIBS= $(LEXLIB) -lm

 TERMCAP=-lncurses
 TERMOPT=-DTERMIO -DDONT_USE_SIGIO

## end Linux 1.2.x and up on Intel x86-based systems
```

W czynnościach przedinstalacyjnych pozostała do zrobienia jeszcze jedna ważna rzecz, bez której proces kompilacji w Ubuntu 10.04 (ale i w nowszych wersjach) zakończy się niepowodzeniem. Należy otworzyć plik **genesis/src/sim/sim_notes.c** w dowolnym edytorze tekstu i zakomentować linię 54. Przed wstawieniem komentarza linia 54 wygląda następująco:

```
#include <unistd.h>
```

Ponieważ plik **sim_notes.c** to w rzeczywistości program napisany w języku C++ – komentarz wstawiamy przez dopisanie na początku linii podwójnego znaku //. Możemy też dopisać informację o tym, kto wstawił komentarz, żeby w przyszłości z łatwością odnaleźć wprowadzone zmiany. Po dodaniu komentarza linia 54. może wyglądać tak:

```
//#include <unistd.h> //by gmwojcik
```

Między innymi takie zabiegi jak powyższy sprawiają, że instalacja GENESIS ze źródeł może stanowić nie lada wyzwanie. Autor pamięta czasy, kiedy kompilując środowisko dla nietypowych systemów klasy UNIX niekiedy mijały tygodnie, zanim poprawne rozwiązanie udawało się wdrożyć. Należy pamiętać, że GENESIS jest rozwijany od 1986 roku. Ćwierć wieku w informatyce to więcej niż cała epoka. Jest rzeczą naturalną, że przez tak wiele lat musiały pojawić się mniejsze lub większe niezgodności w bibliotekach, kompilatorach i w systemie jako takim. Tym większy podziw i satysfakcję powinna wzbudzać poprawnie przeprowadzona instalacja ze źródeł.

2.5. Kompilacja i instalacja

Po prawidłowym wykonaniu wszystkich czynności przedinstalacyjnych możemy przystąpić do procesu kompilacji środowiska. W tym celu powinniśmy przejść do katalogu zawierającego źródła. Bez względu na to, w którym miejscu znajdowaliśmy się aktualnie – wykonanie poniższych poleceń przeniesie nas w pożądane teraz miejsce:

```
cd
cd genesis-2.3/genesis/src
```

Właściwy proces kompilacji rozpoczynamy wydając dwa polecenia:

```
make clean
make all
```

Należy uzbroić się w cierpliwość. W starszych systemach kompilacja trwała około pół godziny. W dzisiejszych czasach, nawet na komputerach osobistych, proces ten nie powinien zająć dłużej niż 5 minut.

Jeżeli wszystko pójdzie dobrze (a powinno!) po kilku minutach z gąszczu wyjściowych komunikatów konsoli naszym oczom powinien ukazać się tekst marzeń:

```
All Libs Compiled
cc -O2 -D__NO_MATH_INLINES -Dnetcdf -DFMT1 -DINCSPRNG
-L/usr/lib sim/simlib.o sys/utillib.o ss/ss.o shell/s
helllib.o par/parlib.o buffer/buflib.o segment/seglib
```

```
.o hh/hhlib.o device/devlib.o out/outlib.o olf/olflib
.o tools/toollib.o concen/conclib.o hines/hineslib.o
user/userlib.o param/paramlib.o pore/porelib.o oldcon
n/axon/axonlib.o oldconn/synapse/synlib.o oldconn/per
sonal/perlib.o oldconn/sim/simconnlib.o oldconn/tools
/toolconnlib.o diskio/interface/netcdf/netcdflib.o di
skio/interface/netcdf/netcdf-3.4/src/libsrc/libnetcdf
.a diskio/interface/FMT1/FMT1lib.o diskio/diskiolib.o
kinetics/kinlib.o newconn/newconnlib.o loadlib.o Xodu
s/_xo/xolib.o Xodus/_widg/widglib.o Xodus/_draw/drawl
ib.o Xodus/Draw/libDraw.a Xodus/Widg/libWidg.a Xodus/
Xo/libXo.a -L/usr/X11R6/lib -lXt -lX11 -lfl -lm sprng
/lib/liblfg.a -lncurses -o genesis
Full GENESIS Compiled -- All Done
```

Tak skompilowaną wersję GENESIS należy teraz zainstalować wykonując polecenie:

```
make install
```

Jeżeli instalacja zakończy się sukcesem otrzymamy komunikat:

```
Done with full install
```

Po zainstalowaniu GENESIS można przejść do czynności poinstalacyjnych.

Zanim to uczynimy pragniemy poinformować czytelników o możliwości kompilacji i instalacji GENESIS na komputerach nie oferujących środowiska graficznego X-Windows. Taka sytuacja występuje prawie zawsze wtedy gdy korzystamy z klastrów obliczeniowych, w tym z wersji równoległej PGENESIS. Między innymi ze względów bezpieczeństwa administratorzy nieczęsto decydują się na instalowanie okienek w superkomputerach. Co zatem zrobić w przypadku, gdy GENESIS podczas kompilacji sporo czasu poświęca na przetwarzanie plików o nazwach rozpoczynających się od „X-”? Jako dobrą wiadomość podajemy, że istnieje możliwość kompilacji i instalacji omawianego środowiska z pominięciem grafiki, oferującego tylko tryb tekstowy. W tym celu należy wykonać wszelkie powyższe przedinstalacyjne zabiegi, włącznie z edycją pliku **Makefile**.

Sama kompilacja i instalacja przebiega po wydaniu poleceń:

```
make clean
make nxall
make nxinstall
```

W przypadku Ubuntu istnieje jeszcze jeden, banalny sposób zainstalowania GENESIS. Wystarczy tylko wykonać polecenie:

```
sudo apt-get install genesis
```

by cieszyć się działającą wersją środowiska 2.3 bez konieczności wykonywania wszystkich czynności, o których mowa od początku rozdziału. W sytuacji, gdy użytkownicy zamierzają przeprowadzać symulacje na komputerze osobistym, bez wykorzystania obliczeń równoległych taka instalacja w zupełności wystarcza. Autor jednak stanowczo zachęca do kompilacji symulatora bezpośrednio ze źródeł. Umożliwia to nie tylko łatwe późniejsze uaktualnienie do wersji równoległej, ale również daje możliwość bezpośredniej ingerencji w „serce" środowiska, co umożliwia wbudowywanie weń nowych funkcjonalności. No i jeszcze ta satysfakcja. . . ; −)

2.6. Czynności poinstalacyjne

Po zakończonej sukcesem instalacji wykonujemy czynności poinstalacyjne. Najpierw należy skopiować do katalogu domowego plik konfiguracyjny **.simrc** albo w przypadku wersji bez X-Windows plik **.nxsimrc**. W tym celu wykonujemy następujące polecenia:

```
cd
cp genesis-2.3/genesis/startup/.simrc .
```

ewentualnie:

```
cd
cp genesis-2.3/genesis/startup/.nxsimrc .
```

Następnie warto dodać ścieżkę do katalogu, w którym znajduje się skompilowany GENESIS do pliku **.bashrc** w taki sposób, aby z każdego miejsca w systemie w łatwy sposób dało uruchamiać się skrypty. W tym celu otwieramy plik konfiguracyjny powłoki w dowolnym edytorze tekstu, np. **gedit**:

```
cd
gedit .bashrc
```

i na samym końcu dopisujemy linię:

```
export PATH=$PATH:/home/student/genesis-2.3/genesis/
```

gdzie zamiast słowa **student** wpisujemy nazwę katalogu domowego użytkownika.

Z punktu widzenia dalszych rozważań wygodnie będzie mieć zainstalowane w systemie narzędzie do wizualizacji wyników. Idealnym kandydatem wydaje się tu być **gnuplot**. Poza tym wygodnie będzie wzbogacić system o starego dobrego **Midnight Commandera** i na przykład kultowy edytor tekstu **emacs**. Wykonanie polecenia:

```
sudo apt-get install gnuplot mc emacs
```

dopełni dzieła instalacji GENESIS w środowisku programisty systemu Ubuntu Linux 10.04.

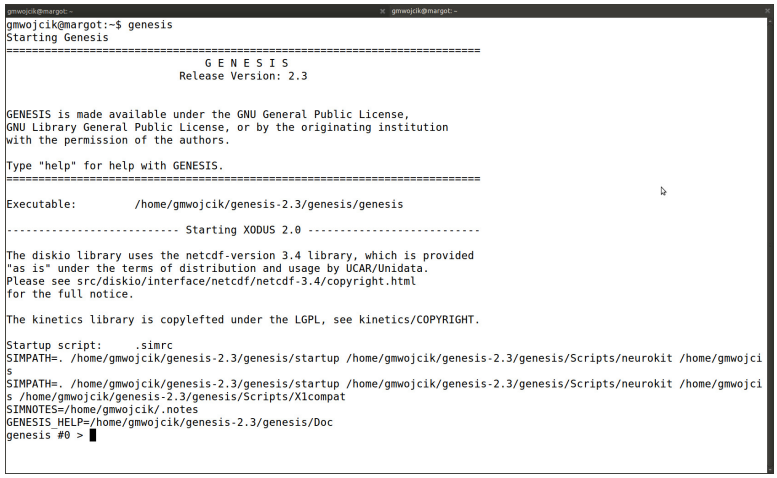
2.7. Sprawdzenie instalacji

Poprawnie zainstalowany GENESIS uruchamiamy z dowolnego miejsca w systemie wykonując polecenie:

```
genesis
```

Uwaga: ze względu na konieczność załadowania utworzonej przed chwilą ścieżki dostępu należy zamknąć terminal i otworzyć go na nowo.

Jeżeli wszystko przebiegło dobrze to oczom użytkownika powinien ukazać się ekran podobny do Rys. 2.1 Pozostaje nam jeszcze sprawdzić czy

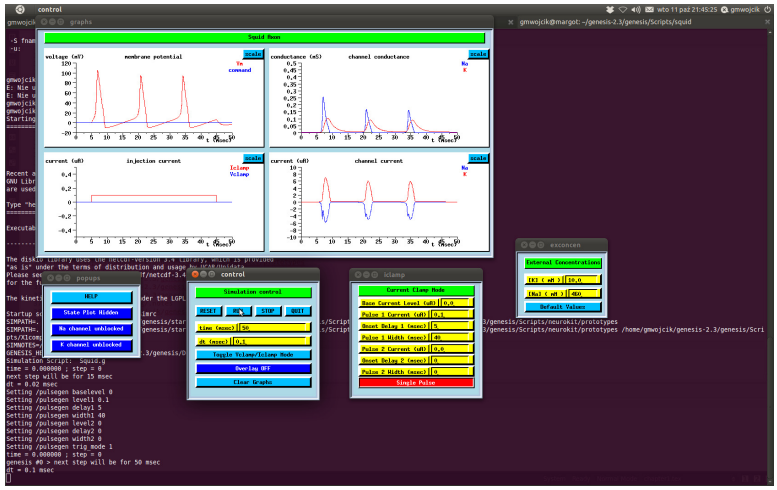


Rysunek 2.1. Konsola nowozainstalowanego GENESIS.

środowisko zainstalowało się również poprawnie w trybie graficznym. W tym celu wykonujemy polecenia:

```
cd
cd genesis-2.3/genesis/Scripts/squid
genesis Squid.g
```

Jeżeli pojawią się staroświeckie okienka i panele kontrolne symulacji - możemy nacisnąć przycisk **RUN**, aby otrzymać efekt podobny do Rys. 2.2. Przycisk **QUIT** powoduje wyjście z tej przykładowej symulacji.



Rysunek 2.2. Nowozaistalowane GENESIS uruchomiony w trybie graficznym.

2.8. Podsumowanie

Przedstawiono proces kompilacji i instalacji środowiska GENESIS v. 2.3 ze źródeł w systemie operacyjnym Ubuntu 10.04 LTS.

2.9. Zadania

Zadanie 1

Zainstaluj ze źródeł środowisko GENESIS 2.3 na domowym komputerze.

Zadanie 2

Zainstaluj oprogramowanie przydatne do pracy w GENESIS.

Rozdział 3

Podstawy języka skryptowego GENESIS

3.1. Wprowadzenie

W tym rozdziale zapoznamy się z podstawami programowania w języku skryptowym GENESIS – SLI (ang. *Script Language Interpreter*). Język ten, którego składnia nieco przypomina C++, sprawia, że GENESIS jest od dwóch dekad jednym z najlepszych środowisk służących modelowaniu i symulacji zarówno pojedynczych komórek nerwowych jak i rozległych sieci neuronowych wielkiej skali.

3.2. Program „Hello World"

Poszczególne polecenia powłoki GENESIS można wydawać bezpośrednio z konsoli symulatora. Nie jest to jednak wygodne rozwiązanie, dlatego że stworzenie dużego modelu wymagałoby bezbłędnego wydania kilkudziesięciu albo kilkuset poleceń. Nietrudno przewidzieć, że w przypadku popełnienia błędu ewentualna jego poprawa byłaby żmudnym albo i niewykonalnym zadaniem. Dlatego skrypty GENESIS powinno się pisać w zwykłych plikach tekstowych (z rozszerzeniem **.g**) wykorzystując dowolny edytor, na przykład **emacs**, **mcedit** albo **gedit**.

Jako pierwszy program w GENESIS uczynimy to co uczynić powinni wszyscy programiści poznający nowy język programowania. Napiszemy program **Hello World** zapisując w pliku **hello.g** następujące polecenie:

```
echo "Hello World"
```

Skrypt uruchamiamy przez wykonanie polecenia w konsoli systemu Linux[1] (nie GENESIS):

```
genesis hello.g
```

Po wykonaniu komendy uruchamia się środowisko GENESIS i po załadowaniu plików konfiguracyjnych w konsoli symultaora pojawiają się następujące wiersze:

```
Simulation Script:  hello.g
Hello World
genesis #0 >
```

Czytelnik, który dokonał powyższego może już uznać się za programistę GENESIS :-).

[1] W tym miejscu mniej zaawansowanym użytkownikom systemu Linux polecamy skrypt pod tytułem „Środowisko programisty" [25], w którym w sposób szczegółowy omówiono nie tylko poruszanie się po konsoli systemowej lecz również przedstawiono wiele aplikacji, które mogą przydać się w codziennym użytkowaniu Ubuntu.

3.3. Deklaracje zmiennych

W kolejnym kroku zapoznamy się z deklaracjami zmiennych występującymi w GENESIS. Na początku proponujemy stworzyć skrypt zawierający następujące linie:

```
echo "Demonstracja zmiennych."

float dt, dx, dy, dz
int i, j, k
float pi=3.1415
int alfa=137

dx = 1e-6
dy = 2e-6
dz = 0.005

i = 7
j = 13
k = 666

echo dx dy dz dt
echo i j k

echo {dx} {dy} {dz} {dt}
echo {i} {j} {k}
```

Znaczenie polecenia **echo** jest intuicyjne, analizując pierwszy przedstawiony tu program wiemy, że służy ono do wyświetlania tekstów na ekranie. Użycie cudzysłowów nie jest to obowiązkowe, jednak będziemy go stosować ze względów estetycznych. Zmienne mogą być nazywane przy pomocy ciągów liter, cyfr i znaku podkreślenia, przy czym nazwa powinna zaczynać się od litery. Zmienne typu rzeczywistego deklarujemy przy użyciu dyrektywy **float**, a całkowitego – **int**. W analizowanym programie zadeklarowaliśmy zmienne rzeczywiste: dt, dx, dy, dz, pi oraz całkowite: i, j, k i alfa. Zmiennym można przypisywać wartości zarówno podczas inicjacji jak też w dowolnym miejscu programu. Przyjemna jest też możliwość przypisywania zmiennym wartości przy pomocy zapisów dziesiętnych typu **1e-4**.

Jednym z aspektów sprawiających wiele trudności na początku pracy z GENESIS jest zagadnienie użycia nawiasów klamrowych. Otóż przedstawiony przykład pokazuje wprost, iż wyświetlenie zawartości zmiennej dajmy na to **dx** przy pomocy polecenia **echo** nastąpi tylko wtedy gdy ujmiemy ją w nawiasy klamrowe. Polecenie **echo dx** spwoduje tylko wyświetlenie w kon-

soli napisu **dx**. Innymi słowy, żeby dobrać się do zawartości zmiennej należy użyć nawiasów klamrowych. Używamy ich zatem przy okazji wyświetlania na ekran, przy przekazywaniu argumentów do funkcji i w innych wymagających tego sytuacjach. Czasami, kiedy jednoznacznie wiadomo, że mamy do czynienia z zawartością zmiennej, a nie z napisem – nawiasy klamrowe można opuścić (na przykład w warunku albo w nagłówku pętli, ale o tym później).

Wykonanie skryptu powinno doprowadzić do wyświetlenia na ekranie konsoli GENESIS następujących wierszy:

```
Simulation Script:  zmienne.g
Demonstracja zmiennych.
dx dy dz dt
i j k
1e-06 2e-06 0.005 0
7 13 666
genesis #0 >
```

Prosimy czytelników o wnikliwe przeanalizowanie treści skryptu oraz wyniku jego działania.

3.4. Operatory i wyrażenia

Kolejny skrypt przedstawia działanie operatorów dodawania, odejmowania, mnożenia i dzielenia w GENESIS.

```
echo "Demonstracja zmiennych i operatorow."

float dt, dx, dy, dz
int i, j, k
float pi=3.1415
int alfa=137

dx = 1e-6
dy = 2e-6
dz = 0.005

i = 7
j = 13
k = 666

echo dx dy dz dt
echo i j k
```

```
echo {dx} {dy} {dz} {dt}
echo {i} {j} {k}

echo {2*dx+2*dy+2*dz}
echo {(i-j)*(i+k)/13}
echo {(i-j)*(i+k)/13.0}
```

Warto zauważyć, że do grupowania działań matematycznych tak jak w większości szanujących się języków programowania służą nawiasy okrągłe. Ostatnie dwie linijki skryptu różnią się tylko dzielnikiem – w pierwszym przypadku trzynastka jest zapisana klasycznie, w drugim jakoby wymuszamy branie jej pod uwagę jako liczby rzeczywistej przez dodanie rozwinięcia dziesiętnego **.0**. Ponieważ zmienne i,j,k są całkowite to pierwsze dzielenie będzie dzieleniem całkowitym, drugie zaś jako wynik da liczbę rzeczywistą.

Efektem działania skryptu będzie wyświetlenie następujących wierszy:

```
Simulation Script:  operatory.g
Demonstracja zmiennych i operatorow.
dx dy dz dt
i j k
1e-06 2e-06 0.005 0
7 13 666
0.010006
-310
-310.6153846
genesis #0 >
```

Czytelników uprasza się o wnikliwe przeanalizowanie treści skryptu oraz wyniku jego działania.

3.5. Instrukcja warunkowa

W tej sekcji przedstawimy składnię instrukcji warunkowej. Prezentowany skrypt sprawdza czy zmienna ma wartość większą od 15, a następnie zmienia jej wartość i dokonuje następnego porównania – tym razem z liczbą 64. Instrukcja warunkowa w GENESIS zawiera kluczowe słowa **if**, **else** oraz słowo ją kończące **end**. Dla pewności treść warunków zapisujemy w nawiasach. Przy okazji warto wspomnieć, że operatory logiczne w instrukcji warunkowej działają tak jak w języku C++, to jest koniunkcja – **&&**, alternatywa – ||, negacja – **!**, porównanie – == no i relacje >, <, >= oraz <=. Należy pamiętać o logicznym stosowaniu nawiasów podczas konstruowania

warunków i jak dobrą radę przyjąć, że lepiej jest mieć o jedną parę nawiasów za dużo niż za mało.

```
echo "Demonstracja instrukcji IF ELSE."

echo "zadeklarujemy x=16."
int x = 16

echo "wchodzimy do instrukcji warunkowej i dostajemy:"

if (x>15)
 echo "x jest wieksze od 16"
else
 echo "x jest mniejsze albo rowne 16."
end

echo "a teraz ustawimy x=64."

x = 64

echo "wchodzimy do podobnej instrukcji warunkowej i dostajemy:"
if (x>64)
 echo "x jest wieksze od 64."
else
 echo "x jest mniejsze albo rowne 64."
end
```

Efektem działania przedstawionego skryptu są następujące wiersze:

```
Simulation Script:  if.g
Demonstracja instrukcji IF ELSE.
zadeklarujemy x=16.
wchodzimy do instrukcji warunkowej i dostajemy:
x jest wieksze od 16
a teraz ustawimy x=64.
wchodzimy do podobnej instrukcji warunkowej i dostajemy:
x jest mniejsze albo rowne 64.
genesis #0 >
```

Czytelników prosimy o rzetelne zapoznanie się z treścią skryptu i wynikiem jego działania.

3.6. Funkcje

Język skryptowy GENESIS umożliwia korzystanie z funkcji. Wykorzystujemy tu kluczowe słowo **function**, a koniec implementacji funkcji oznaczamy słowem **end**. W definicji funkcji – wszystkie argumenty umieszczamy oddzielone przecinkami w nawiasie. Należy jednak pamiętać, że bezpośrednio pod identyfikatorem funkcji musimy podać typy jej argumentów deklarując odpowiednie zmienne.

Popuściwszy wodze wyobraźni postanowiliśmy określić rozwiązywalność równania kwadratowego wykorzystując funkcję napisaną w GENESIS i obliczającą wyróżnik trójmianu kwadratowego (a co!). Treść całego skryptu prezentuje się następująco:

```
function wyroznik(a, b, c)
int a,b,c, delta

delta = {b*b-4*a*c}

 if (delta > 0)
  echo "delta = " {delta} ", beda dwa rozwiazania!"
 else
  if (delta == 0)
   echo "delta = " {delta} ", bedzie jedno rozwiazanie!"
  else
   echo "delta = " {delta} ",nie bedzie rozwiazan!"
  end
 end

end // funkcji

echo "Rownanie kwadratowe - wersja light w GENESIS."

int a=1
int b=-2
int c=1

wyroznik {a} {b} {c}

wyroznik 2 3 4

wyroznik 3 16 1
```

```
echo "KONIEC"
```

Funkcja w programie wywoływana jest trzykrotnie dla przypadków: $A = 1, B = -2, C = 1$; $A = 2, B = 3, C = 4$; $A = 3, B = 16, C = 1$. Za każdym razem funkcja informuje nas o liczbie rozwiązań określonego w zadany sposób równania kwadratowego. Uwaga: podczas wywoływania funkcji listę argumentów wprowadzamy bez nawiasów, oddzielając je spacjami. Pamiętamy też o rozsądnym stosowaniu nawiasów klamrowych.

Wynik działania skryptu przedstawiają poniższe wiersze:

```
Simulation Script:  delta.g
Rownanie kwadratowe - wersja light w GENESIS.
delta =  0 , bedzie jedno rozwiazanie!
delta =  -23 ,nie bedzie rozwiazan!
delta =  244 , beda dwa rozwiazania!
KONIEC
genesis #0 >
```

Czytelnicy zechcą zwrócić uwagę na zagnieżdżoną instrukcję warunkową występującą w ciele funkcji oraz na treść całego skryptu w odniesieniu do wyników jego działania.

3.7. Pętla for

Bardzo przydatną instrukcją, zwłaszcza przy tworzeniu sieci wielu neuronów, jest pętla **for**. Nagłówek pętli przypomina składnię języka C++, należy jednak pamiętać, że całość zakończona jest kluczowym słowem **end**. Przykładowy program wykorzystujący pętlę **for** przedstawia listing:

```
echo "Demonstracja dzialania petli FOR."

int i

for (i=1; i<=10; i=i+1)
 echo {i}
end
```

Ten kawałek kodu powoduje wyświetlenie na ekranie kolejnych liczb całkowitych od 1 do 10. Wynikiem działania skryptu ilustrującego działanie pętli **for** jest więc:

```
Simulation Script:  for.g
Demonstracja dzialania petli FOR.
1
```

```
2
3
4
5
6
7
8
9
10
genesis #0 >
```

Dla treningu możemy teraz napisać skrypt, który wyświetli na ekranie kolejne (pierwsze 9) wyrazy ciągu Fibonacciego (jak szaleć to szaleć!).

```
echo "Ciag Fibonacciego w GENESIS"

int licznik, a0, a1, a2

a0 = 0
a1 = 1

echo {a0}
echo {a1}

for (licznik = 1; licznik < 8; licznik = licznik + 1)

a2 = a0 + a1
echo {a2}

a0 = a1
a1 = a2

end
```

Efektem działania skryptu będzie wyświetlenie ciągu liczb całkowitych, z których pierwsze dwie to 0 i 1, a każda następna jest sumą dwóch bezpośrednio ją poprzedzających:

```
Simulation Script:  fibon.g
Ciag Fibonacciego w GENESIS.
0
1
1
2
```

```
3
5
8
13
21
genesis #0 >
```

Petle **for** podobnie jak instrukcje warunkowe mogą być wielokrotnie zagnieżdżone. Należy tylko pamiętać o blokowaniu instrukcji przy pomocy słówka **end**. czytelnicy proszeni są o uważne zapoznanie się z przykładami użycia pętli **for**.

3.8. Inne ważne polecenia

Istnieje jeszcze wiele ważnych instrukcji języka skryptowego GENESIS. Autor starał się przedstawić tylko te najważniejsze, niezbędne do rozpoczęcia pracy z symulatorem. W miarę wprowadzania nowych treści – kolejne instrukcje będą przemycane do świadomości czytelników. Warto tu wspomnieć, że GENESIS posiada sprawnie działający mechanizm obsługi plików tekstowych, generator liczb pseudolosowych, a informacja między poszczególnymi elementami modelu przesyłana jest za pomocą wiadomości (ang. *messages*).

3.9. Podsumowanie

W tym rozdziale przedstawiono najważniejsze instrukcje skryptowego języka GENESIS. czytelnicy po zapoznaniu się z treścią niniejszego rozdziału powinni już potrafić deklarować zmienne, wyświetlać ich zawartość na ekranie, korzystać z instrukcji warunkowej oraz pętli. Do konstruowania warunków potrafią wykorzystywać operatory relacji oraz logiczne.

3.10. Zadania

Zadanie 1

Napisz skrypt w GENESIS wyświetlający na ekranie kilka zwrotek Twojej ulubionej piosenki.

Zadanie 2

Przepisz ze zrozumieniem wszystkie skrypty z niniejszego rozdziału i sprawdź czy wynik ich działania jest taki jak przedstawiony w treści.

Zadanie 3

Napisz dowolny skrypt prezentujący działanie instrukcji warunkowej.

Zadanie 4

Napisz skrypt wyświetlający na ekranie dziesięć wyrazów ciągu arytmetycznego o zadanym pierwszym wyrazie i różnicy.

Zadanie 5

Napisz skrypt wyświetlający na ekranie dwanaście wyrazów ciągu geometrycznego o zadanym pierwszym wyrazie i ilorazie.

Zadanie 6

Wykorzystując zagnieżdżone pętle for napisz skrypt wyświetlający wyrazy tabliczki mnożenia z zakresu od 1×1 do 10×10.

Rozdział 4

Interfejs graficzny XODUS – podstawy

4.1. Wprowadzenie

W tym rozdziale będzie więcej rysunków niż treści. Standardowy GENESIS oferuje użytkownikowi pracę w środowisku graficznym o nazwie XODUS (ang. *X-Windows Output Display Utility System*). Warto zwrócić uwagę na swego fantazję twórców środowiska – oto bowiem GENESIS to biblijna Księga Rodzaju, XODUS - Księga Wyjścia. Dla porządku podamy jeszcze, że grupa dyskusyjna użytkowników GENESIS nosi nazwę BABEL.

XODUS sporo potrafi jednak ze względu na swój wiek wygląda tak, że może podobać się tylko najbardziej zagorzałym fanom komputerów Commodore 64. Obecnie sami w dowolnym języku programowania możemy napisać sobie programy lepiej wizualizujące aktywność komórek, a także w zaawansowanych edytorach ustawiać parametry symulacji.

Jednak przez szacunek dla twórców pakietu pokażemy trzy najbardziej widowiskowe modele wykorzystujące środowisko graficzne XODUS.

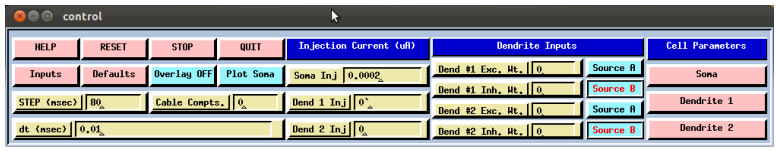
4.2. Praca z komórką – Neuron.g

Na początku pobawimy się pojedynczą komórką. W tym celu należy dostać się do katalogu **genesis/Scripts/neuron** i wykonać polecenie:

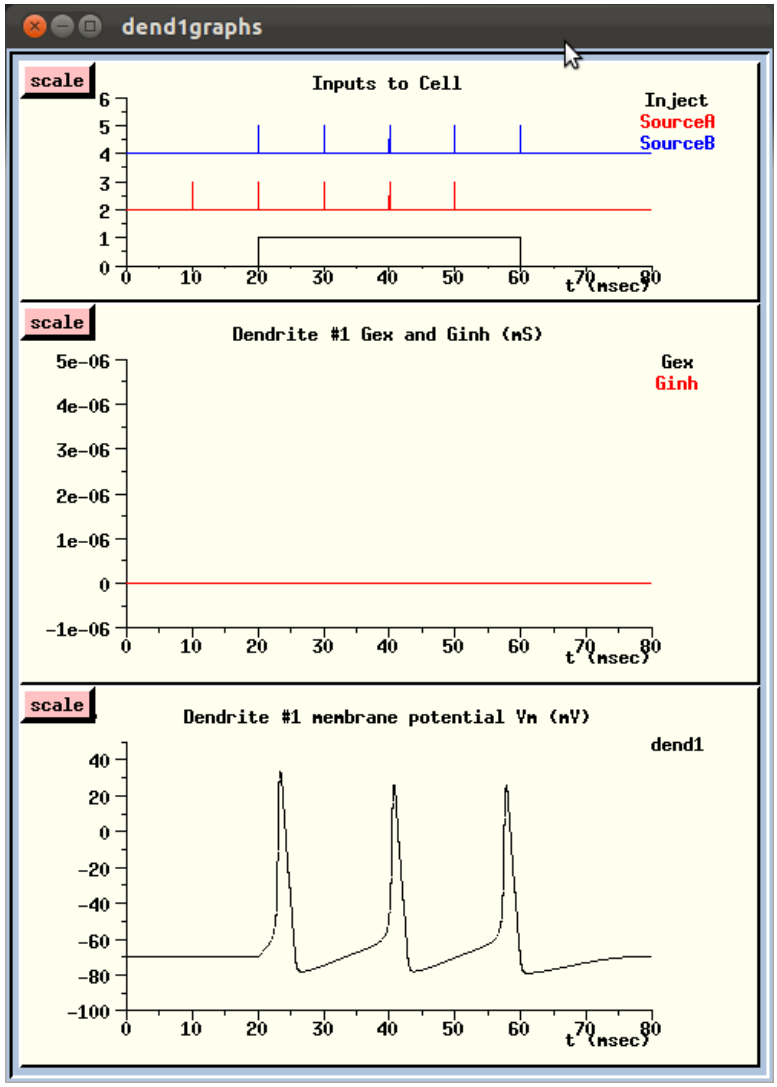
```
genesis Neuron.g
```

Powinien wyświetlić się zestaw oldschoolowych okienek (Rys. 4.1-4.3), wśród których najważniejszy jest panel sterowania **control** (Rys. 4.1). Symulację uruchamiamy wciskając przycisk **step**. W ten sposób możemy bawić się bez końca pamiętając jednak, że przed ponownym uruchomieniem należy wcisnąć przycisk **reset** (tylko nie ten na obudowie komputera!).

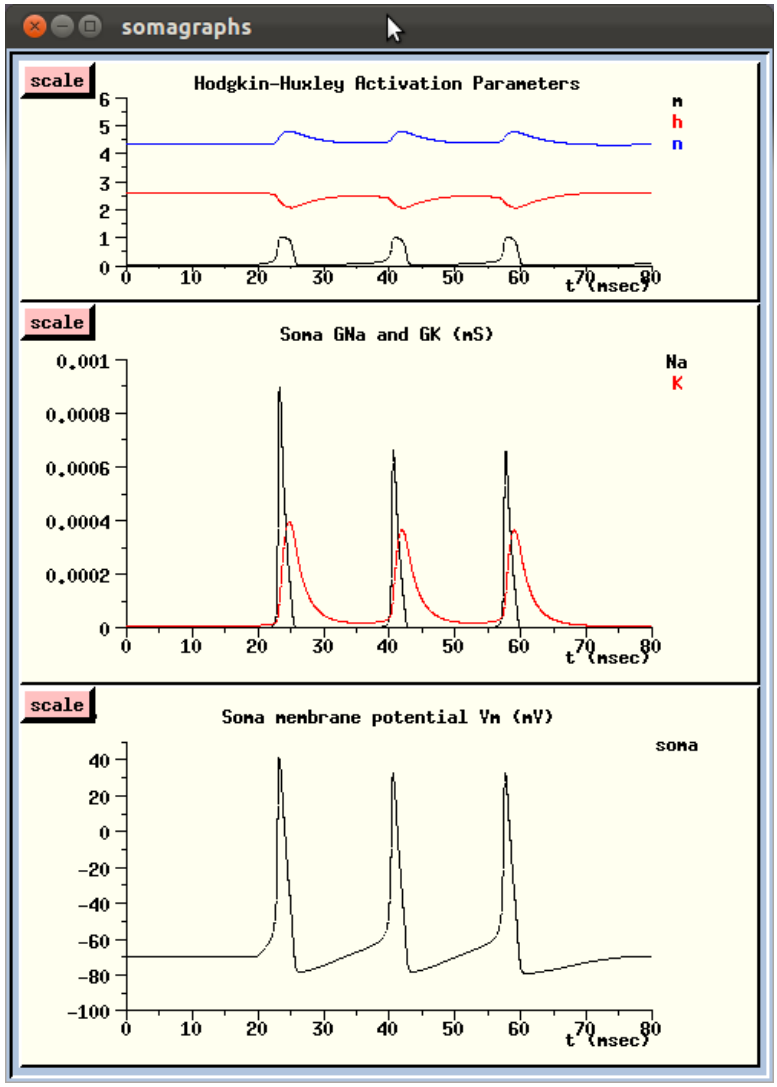
Odpowiednie różowe przyciski przy wykresach pozwalają zmienić skalę dla wyświetlanych na osiach wartości. Możemy poeksperymentować z wartościami prądów podawanych na poszczególne komórki, po czym wyjść z modelu wciskając przycisk **quit**.



Rysunek 4.1. Panel sterowania symulacją Neuron.g



Rysunek 4.2. Wykresy aktywności wejścia, przewodności oraz potencjału dendrytu w symulacji Neuron.g



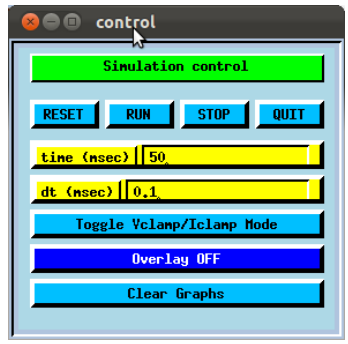
Rysunek 4.3. Wykresy parametrów aktywacji kanałów HH, przewodności oraz potencjału błony komórkowej w symulacji Neuron.g

4.3. Doświadczenia na kałamarnicy – Squid.g

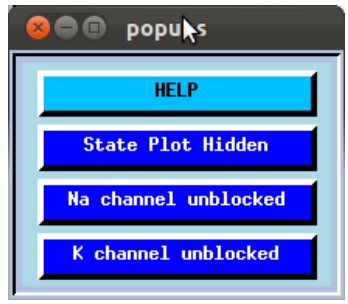
Eksperymenty na neuronach kałamarnicy z doświadczeń Hodgkina i Huxleya wykonujemy na modelu znajdującym się w katalogu **genesis/Scripts/squid**:

```
genesis Squid.g
```

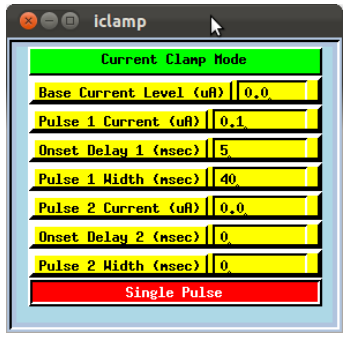
Tu przyciskiem uruchamiającym symulację jest **run** znajdujący się w panelu sterowania **control** (Rys. 4.4). Warto również przetestować zachowanie modelu zmieniając parametry w rozmaitych polach, pamiętając jednak o tym, że w przestarzałych interfejsach tego typu po dokonaniu zmiany trzeba nacisnąć klawisz **enter** na klawiaturze komputera i to w chwili, gdy kursor myszki znajduje się nad edytowanym polem. Przycisk **quit** podobnie jak poprzednio służy do zamknięcia wszystkich okienek i opuszczenia modelu.



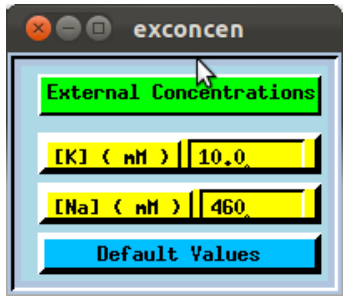
Rysunek 4.4. Panel sterowania symulacją Squid.g



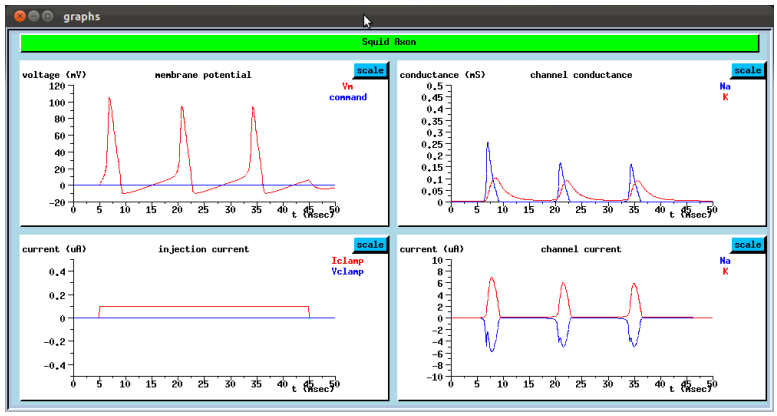
Rysunek 4.5. Kontrola kanałów sodowych i potasowych w symulacji Squid.g



Rysunek 4.6. Kontrola impulsów wejściowych w symulacji Squid.g



Rysunek 4.7. Kontrola zewnętrznych stężeń jonowych w symulacji Squid.g



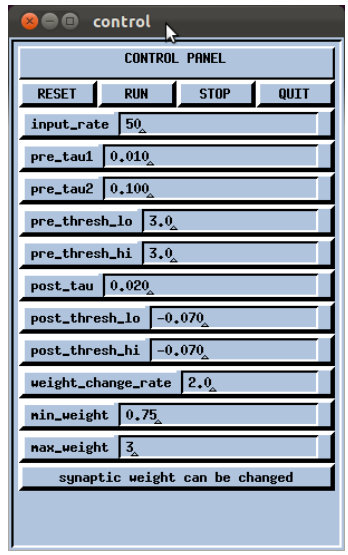
Rysunek 4.8. Wykresy aktywności wejścia oraz potencjału błony, przewodności i prądu w kanałach jonowych w symulacji Squid.g

4.4. Mechanizm uczenia – Hebb.g

Ciekawy przykład modelu obrazującego mechanizm uczenia według Donalda Hebba znajduje się w katalogu **genesis/Scripts/examples/hebb**, uruchamiany przez:

```
genesis hebb.g
```

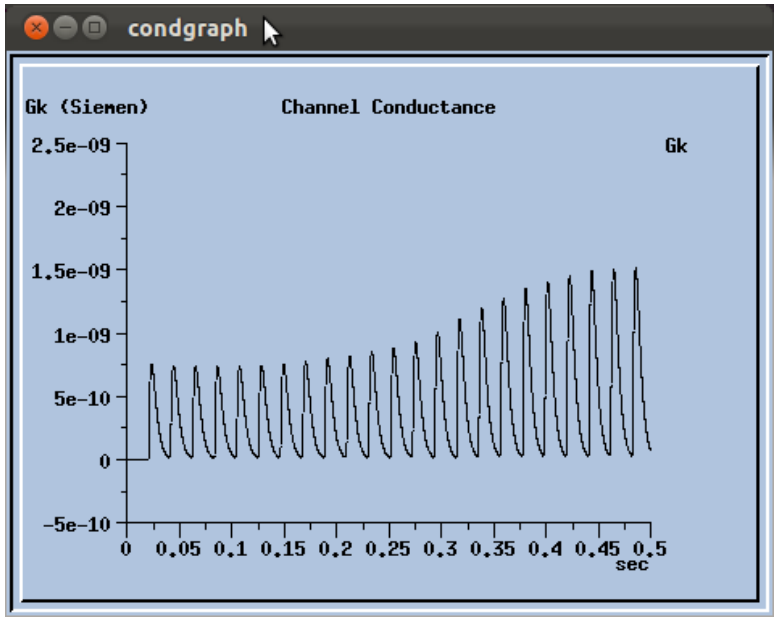
Symulację rozpoczynamy przez wciśnięcie przycisku **run** na panelu sterowania **control** (Rys. 4.9). Oprócz innych ciekawych wielkości warto przyjrzeć się wykresowi zmiany wagi synapsy Hebba (Rys. 4.13). Waga ta najpierw narasta powoli, potem następuje gwałtowny wzrost, po czym ma miejsce wyraźne wysycenie. Zupełnie jak w życiu, gdzie przyswajanie jakiejś konkretnej wiedzy na początku idzie nam opornie, potem szybko uczymy się wraz z doświadczeniem nowych rzeczy, ale żeby dojść do perfekcji i mistrzowskiego poziomu, na przykład w opanowaniu języka – mijają całe lata.



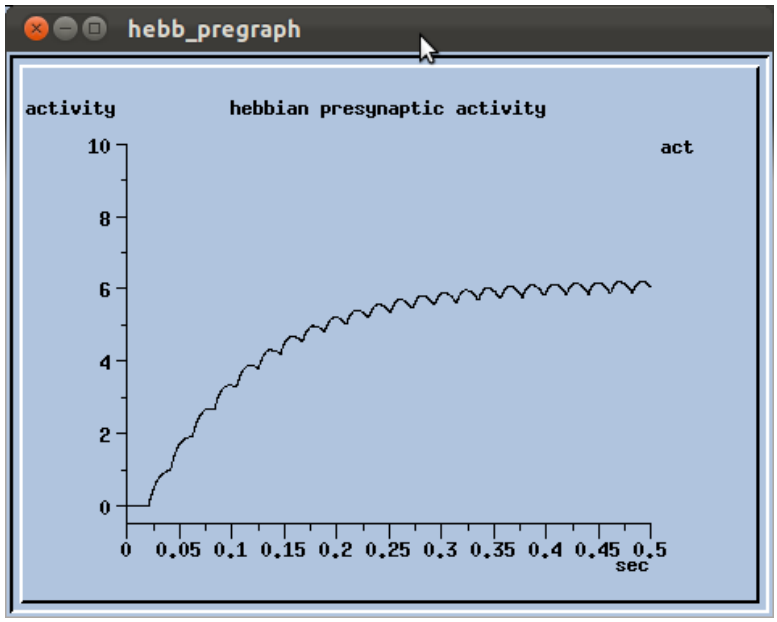
Rysunek 4.9. Panel sterowania symulacją Hebb.g

4.5. Podsumowanie

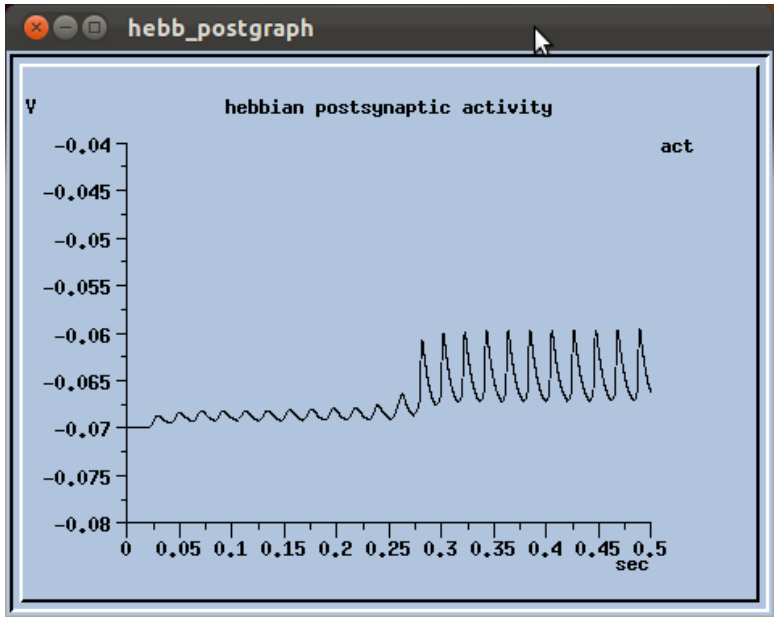
Przedstawiono zarys możliwości pracy z interfejsem graficznym XODUS symulatora GENESIS. Szczegółowy opis tworzenia poszczególnych okienek oraz znaczenie obecnych w nich elementów znajduje się w [19]



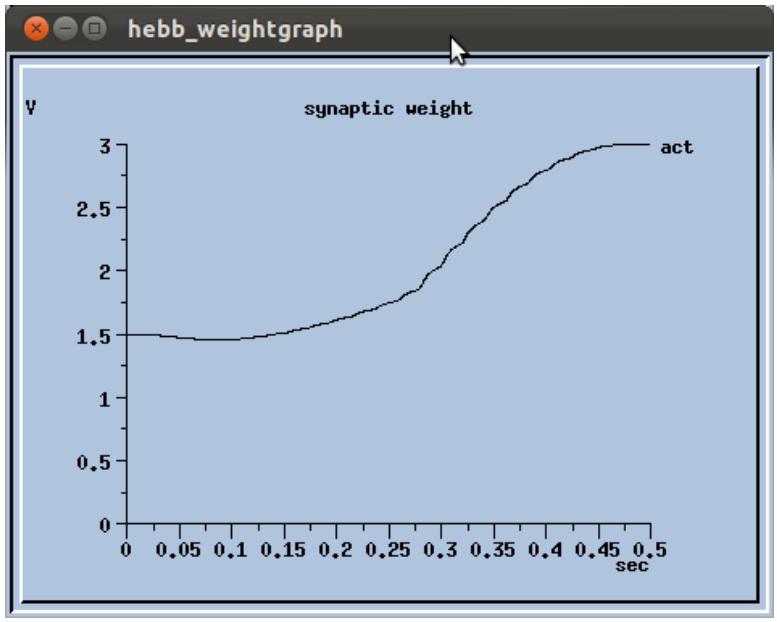
Rysunek 4.10. Wykres przewodności kanału w symulacji Hebb.g



Rysunek 4.11. Wykres aktywności presynaptycznej w symulacji Hebb.g



Rysunek 4.12. Wykres aktywności postsynaptycznej w symulacji Hebb.g



Rysunek 4.13. Wykres zmieniającej się wagi synaptycznej w symulacji Hebb.g

4.6. Zadania

Zadanie 1

Przetestuj model **Neron.g**. Zmieniaj skale wykresów i parametry symulacji w taki sposób, żeby zaobserwować jakiekolwiek różnice. Nie musisz zagłębiać się w neurofizjologię komórek.

Zadanie 2

Przetestuj model **Squid.g**. Zmieniaj skale wykresów i parametry symulacji w taki sposób, żeby zaobserwować jakiekolwiek różnice. Nie musisz zagłębiać się w neurofizjologię komórek.

Zadanie 3

Przetestuj model **hebb.g**. Zmieniaj skale wykresów i parametry symulacji w taki sposób, żeby zaobserwować jakiekolwiek różnice. Nie musisz zagłębiać się w neurofizjologię komórek.

Rozdział 5

Modelowanie pojedynczej komórki nerwowej

5.1. Wprowadzenie

Celem rozdziału jest nauczenie czytelników modelowania pojedynczej komórki nerwowej. Naukę rozpoczynamy od modelowania tylko ciała komórki bez uwzględniania standardowych jednostek układu SI. Następnie wprowadzamy te jednostki i definiujemy zegary sterujące krokami poszczególnych aspektów symulacji. W kolejnym podrozdziale wprowadzamy kanały jonowe do modelowanego ciała komórki. Czytelnicy poznają również zasady zapisywania wartości potencjału błony komórkowej do pliku oraz czasu pojawiania się potencjałów czynnościowych na symulowanej komórce. Na zakończenie pokazana jest przykładowa konstrukcja bardziej złożonej komórki z wykorzystaniem metody **Cell Reader**. Wyniki prostych symulacji wizualizujemy w programie **gnuplot**.

5.2. Modelowanie ciała komórki

Skrypty GENESIS zapisywane są w plikach, którym tradycyjnie nadajemy rozszerzenie **.g**. Przykłady omawiane w tym rozdziale zadziałają nawet wtedy, gdy każde polecenie wydamy bezpośrednio z wiersza poleceń środowiska. Przypominamy, że nie jest to jednak zalecane ze względu na fakt, iż w razie pomyłek może zdarzyć się konieczność ponownego wpisywania wszystkiego od początku. Dlatego polecamy zapisać nasze rozważania w pliku o nazwie np. **1cell.g**, a następnie z poziomu konsoli systemu Linux uruchamiać skrypty wykonując polecenie:

```
genesis 1cell.g
```

pamiętając, żeby wcześniej przejść do katalogu, w którym znajduje się skrypt.

Budowę modelu pojedynczej komórki nerwowej rozpoczynamy od tak zwanej definicji elementu neutralnego:

```
create neutral /cell
```

Element neutralny o dowolnej nazwie, w tym przypadku **cell**, będzie elementem nadrzędnym nad wszystkimi pozostałymi elementami wchodzącymi w skład modelowanej komórki. W przypadku odwoływania się do komórki będziemy podawać nazwę elementu neutralnego. Ciało oraz dendryty komórki będą znajdowały się w strukturze modelu niejako pod elementem neutralnym, podobnie jak zagnieżdżone podkatalogi w dowolnym katalogu systemu plików.

W następnym kroku definiujemy ciało komórki jako takie. Tu obowiązkowo należy użyć zarezerwowanej nazwy **soma**.

```
create compartment /cell/soma
```

Ustawiamy wartości pól odpowiadających parametrom Hodgkina–Huxleya odpowiednio na $R_m = 10$, $C_m = 2$, $E_m = 25$ oraz $I_{inj.} = 5$, na razie nie zawracając sobie głowy jednostkami układu SI.

```
setfield /cell/soma Rm 10 Cm 2 Em 25 inject 5
```

Następnie zapewniamy zapis uzyskanych w symulacji wartości potencjału elektrycznego błony komórkowej do pliku tekstowego **soma1.vm**.

```
create asc_file /soma_potential
setfield /soma_potential filename soma1.vm append 0
addmsg /cell/soma /soma_potential SAVE Vm
```

W powyższym fragmencie kodu należy zwrócić uwagę na skojarzenie obiektu **soma_potential** z plikiem tekstowym (pierwsza linia), ustawienie parametrów w poszczególnych polach obiektu skojarzonego z plikiem (w tym fizycznej nazwy pliku na dysku – linia druga) oraz w trzeciej linii wysłanie wiadomości z ciała komórki do zmiennej skojarzonej z plikiem tekstowym przy jednoczesnym wydaniu dyrektywy **SAVE** dla potencjału błony V_m.

Pod koniec każdego skryptu warto sprawdzić czy zaprojektowane przez nas struktury tworzą poprawną logicznie całość:

```
check
```

oraz zresetować inicjacyjne parametry komórek:

```
reset
```

Polecenie **step** służy do wykonania zadanej jako parametr liczby kroków symulacji:

```
step 100
```

Warto zapewnić powrót ze skryptu do powłoki systemu Linux wydając polecenie:

```
quit
```

Gotowy skrypt przyjmie teraz postać:

```
create neutral /cell
create compartment /cell/soma

setfield /cell/soma Rm 10 Cm 2 Em 25 inject 5

create asc_file /soma_potential
setfield /soma_potential filename soma1.vm append 0
addmsg /cell/soma /soma_potential SAVE Vm
```

```
check
reset
step 100

quit
```

przy czym od początku warto pamiętać o systematycznym komentowaniu poszczególnych linijek tworzonych programów.

Uruchamiamy skrypt poleceniem:

```
genesis 1cell.g
```

GENESIS uruchamia się, wykonuje 100 kroków symulacji, po czym wraca do konsoli systemowej. Jeżeli wszystko zadziałało poprawnie to w katalogu, w którym rezyduje skrypt powinien pojawić się nowy plik **soma1.vm**.

Program **gnuplot** jest doskonałym narzędziem do wizualizowania zwracanych przez GENESIS plików. W tym celu przygotujemy skrypt, który wykreśli nam zawartość otzrymanego z symulacji pliku i wyeksportuje grafikę do pliku postscriptowego **wykres1.eps**

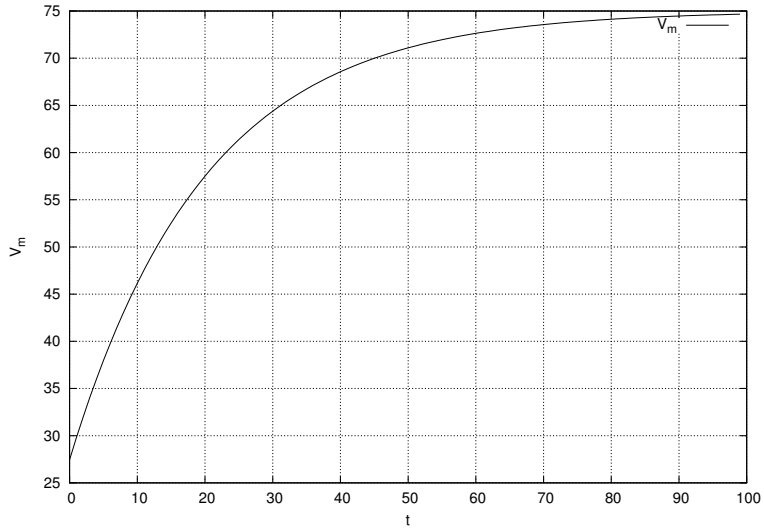
```
set term postscript enhanced  monochrome
set grid
set xlabel "t"
set ylabel "V_{m}"

set output "wykres1.eps"
plot "soma1.vm" u 1:2 w l t "V_{m}"
```

Skrypt **gnuplot** uruchamiamy wykonując polecenie:

```
gnuplot nazwa_skryptu_gnuplotowego
```

bezpośrednio z konsoli Linuksa. Należy pamiętać o wcześniejszym zainstalowaniu gnuplota, ale o tym była już mowa przy okazji instalacji i konfiguracji GENESIS. Przytoczony skrypt zapewnia, że na wykresie pojawi się elegancka siatka, osie będą odpowiednio podpisane i wykreślimy drugą kolumnę pliku **soma1.vm** względem pierwszej nadając powstałej krzywej właściwy podpis V_m w legendzie.

Na Rys. 5.1 stosunkowo niewiele się dzieje. Ponieważ nie zaimplementowaliśmy jeszcze kanałów jonowych obserwujemy tylko narastanie potencjału w kolejnych stu krokach symulacji. Nie mamy jeszcze uwzględnionej normalizacji do standardowych jednostek układu SI, zatem można powiedzieć, że wzrost wartości V_m w czasie jest wręcz zbyt „elegancki”.



Rysunek 5.1. Potencjał błony komórkowej bez kanałów jonowych i bez uwzględnienia jednostek SI.

5.3. Wprowadzenie zegarów i jednostek SI

Wierniejsza symulacja ciała pojedynczej komórki wymaga uwzględnienia jednostek układu SI oraz zainicjowania poszczególnych zmiennych w modelu oryginalnymi wartościami z teorii Hodgkina–Huxleya. W tym celu na początku skryptu z ulepszonym modelem dodajemy blok instrukcji:

```
float   PI = 3.14159

// Parametry ciala komorki w jednostkach SI
float RM = 0.33333 // opor blony (ohm m^2)
float CM = 0.01 // pojemnosc (farad/m^2)
float RA = 0.3 // opor aksjalny (ohm m)
float EREST_ACT = -0.07  // potenc. spoczynkowy (volt)
float Eleak = EREST_ACT + 0.0106  // potenc. uplywu (volt)
float ENA   = 0.045  // potenc. rown. sodowy
float EK    = -0.082 // potenc. rown. potasowy

// wymiary komorki (m)
float soma_l = 30e-6 // dlugosc walca
float soma_d = 30e-6 // srednica walca
```

gdzie w formie komentarzy opisano znaczenie poszczególnych parametrów.

Kolejnym krokiem wzbogacającym omawiany model i ważnym posunięciem z punktu widzenia uruchamiania symulacji w GENESIS jest definicja kroku czasowego symulacji oraz ustawienie tak zwanych zegarów. Mniejszy krok symulacji prowadzi do dokładniejszych wyników. Niestety związane jest to z wydłużeniem czasu obliczeń. Twórcy modeli powinni skupić się zatem na optymalnym doborze kroku symulacji, który godzi dokładność rezultatów z akceptowalnym czasem przebiegu symulacji. Do ustawienia głównego zegara modelu używa się poleceń, gdzieś na początku skryptu:

```
float dt = 0.00005 // krok symulacji w sek.
setclock  0  {dt}  // glowny zegar symulacji
```

przy czym można definiować wiele zegarów numerowanych kolejnymi liczbami całkowitymi. Później przekonamy się, że inne zegary mogą być wykorzystane do prowadzenia symulacji jako takie (rozwiązywania układów równań różniczkowych), a inne na przykład do zapisywania wyników na dysk. W ten sposób można zapewnić wierne obliczenia i rzadsze próbkowanie. Często takie podejście jest wręcz wskazane.

Przyszła pora, by na prostym przykładzie nauczyć się definiować funkcje w GENESIS. Stworzymy funkcję, która przyjmując kilka parametrów stworzy z nich element modelowanej komórki nerwowej:

```
function makecompartment(path, length, dia, Erest)
    str path
    float length, dia, Erest
    float area = length*PI*dia
    float xarea = PI*dia*dia/4

    create      compartment     {path}
    setfield   {path}              \
                Em      { Erest }   \          // volts
                Rm      { RM/area } \          // Ohms
                Cm      { CM*area } \          // Farads
                Ra      { RA*length/xarea }    // Ohms
end
```

Jak wiemy - definicję funkcji rozpoczynamy od słowa kluczowego **function**, a kończymy słowem **end**. Argumenty podajemy w nawiasach. W prezentowanym przykładzie mamy ponadto do czynienia z zmiennymi typu łańcuchowego **str** i zmiennoprzecinkowymi **float**. Pamiętajmy też, że wszelkie odniesienia do zmiennych, które nie przywędrowały do funkcji jako argument muszą być poprzedzone ujęciem identyfikatora zmiennej w nawiasy klamrowe. Czytelnik proszony jest o zwrócenie uwagi na symbol \ służący do przełamywania linii.

Dalsza część skryptu wygląda bardzo podobnie do pierwszego omawianego przypadku. Po utworzeniu elementu neutralnego wywołujemy zadeklarowaną wcześniej funkcję:

```
create neutral /cell
makecompartment /cell/soma {soma_l} {soma_d} {Eleak}
```

i zapewniamy pubudzanie ciała komórki prądem z zewnątrz:

```
setfield /cell/soma inject  0.3e-9 // prad o nat. 0.3 nA
```

Zapewniamy zapis wyników na dysk do pliku **soma2.vm**, sprawdzamy poprawność, resetujemy aktywność:

```
create asc_file /soma_potential
useclock /soma_potential 0
setfield /soma_potential filename soma2.vm append 0
addmsg /cell/soma /soma_potential SAVE Vm

check
reset
```

i uruchamiamy symulację w taki sposób, by po zakończeniu powrócić do konsoli systemu Linux:

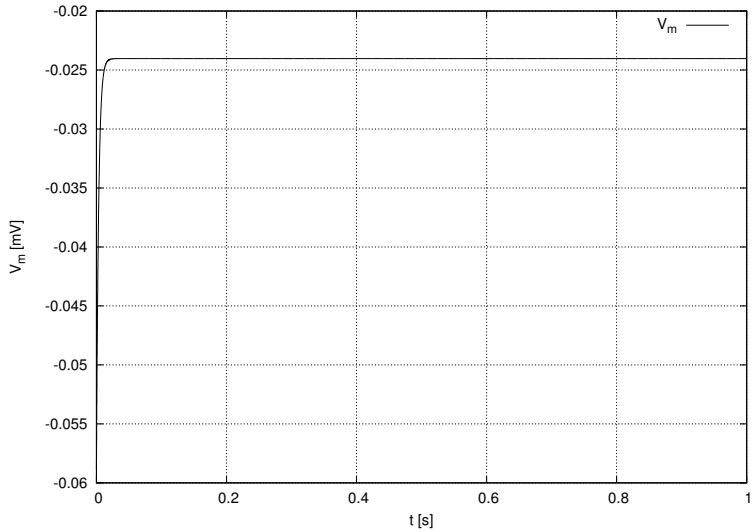
```
step 1 -time
quit
```

Parametr **-time** przy poleceniu step informuje GENESIS, że symulujemy – w tym przypadku – jedną sekundę biologicznej aktywności komórki. Oznacza to, że przy zdefiniowanym wcześniej kroku symulacji wynoszącym 0.00005 s system będzie musiał wykonać 20000 kroków. Zauważmy, że przy zapewnianiu zapisu wyników na dysk poinformowaliśmy GENESIS, że do czynności tej powinien używać zegara identyfikowanego przez liczbę **0**.

Po zakończeniu symulacji możemy wykreślić nowopowstały plik. Skrypt **gnuplota** wzbogacamy o jednostki SI w podpisach osi, pamiętamy też, że zarówno plik z danymi jak i wyjściowy powinny mieć inne nazwy.

```
set xlabel "t [s]"
set ylabel "V_{m} [V]"

set output "wykres2.eps"
plot "soma2.vm" u 1:2 w l t "V_{m}"
```

Rys. 5.2 jest znormalizowany w stosunku do Rys. 5.1, ale wciąż jeszcze nie obserwujemy pików potencjału czynnościowego jako że wciąż nie mamy wprowadzonych do ciała komórki kanałów jonowych.



Rysunek 5.2. Potencjał błony komórkowej bez kanałów jonowych, ale z uwzględnieniem jednostek SI.

5.4. Wprowadzenie kanałów jonowych

Kanały jonowe wprowadza się ręcznie do ciała modelowanej komórki w trochę zawiły sposób, jednak jak przekonamy się wkrótce cały proces tworzenia komórek da się elegancko zautomatyzować. W celu wzbogacenia ciała komórki o mechanizmy sodowo–potasowe na początku skryptu należy załączyć bibliotekę funkcji **hhchan**:

```
include hhchan
```

Następnie, tuż przed definicją kroku symulacji powinniśmy zadeklarować zmienną konieczną do poprawnego działania funkcji z załączonej wcześniej biblioteki:

```
float SOMA_A = soma_l*PI*soma_d //pow. somy dla hhchan.g
```

Po stworzeniu elementu neutralnego i właściwego ciała komórki budujemy kanały jonowe wspomagając się poleceniami **pushe** i **pope** służącymi do wchodzenia wgłąb i wychodzenia na zewnątrz aktualnego elementu. Same kanały tworzone są przez funkcje z **hhchan**.

```
pushe /cell/soma
make_Na_squid_hh
make_K_squid_hh
pope
```

Teraz już tylko musimy powiązać kanały jonowe z ciałem komórki przez zapewnienie przesyłania odpowiednich wiadomości:

```
addmsg /cell/soma/Na_squid_hh /cell/soma CHANNEL Gk Ek
addmsg /cell/soma /cell/soma/Na_squid_hh VOLTAGE Vm
addmsg /cell/soma/K_squid_hh /cell/soma CHANNEL Gk Ek
addmsg /cell/soma /cell/soma/K_squid_hh VOLTAGE Vm
```

Treść całego skryptu zawierającego model ciała komórki wraz z zaimplementowanymi kanałami jonowymi wygląda następująco:

```
include hhchan

float   PI = 3.14159

float RM = 0.33333
float CM = 0.01
float RA = 0.3
float EREST_ACT = -0.07
float Eleak = EREST_ACT + 0.0106
float ENA   = 0.045
float EK    = -0.082

float soma_l = 30e-6
float soma_d = 30e-6

float SOMA_A = soma_l*PI*soma_d

float dt = 0.00005
setclock  0  {dt}

function makecompartment(path, length, dia, Erest)
    str path
    float length, dia, Erest
    float area = length*PI*dia
    float xarea = PI*dia*dia/4

    create      compartment     {path}
    setfield   {path}              \
                Em      { Erest }   \
                Rm      { RM/area } \
                Cm      { CM*area } \
                Ra      { RA*length/xarea }
```

```
end

create neutral /cell
makecompartment /cell/soma {soma_l} {soma_d} {Eleak}
setfield /cell/soma inject  0.3e-9

pushe /cell/soma
make_Na_squid_hh
make_K_squid_hh
pope

addmsg /cell/soma/Na_squid_hh /cell/soma CHANNEL Gk Ek
addmsg /cell/soma /cell/soma/Na_squid_hh VOLTAGE Vm
addmsg /cell/soma/K_squid_hh /cell/soma CHANNEL Gk Ek
addmsg /cell/soma /cell/soma/K_squid_hh VOLTAGE Vm

create asc_file /soma_potential
useclock /soma_potential 0
setfield /soma_potential filename soma3.vm append 0
addmsg /cell/soma /soma_potential SAVE Vm

check
reset

step 1 -time
quit
```

Dla przejrzystości kodu pominięto wskazane i zawsze zalecane komentarze.

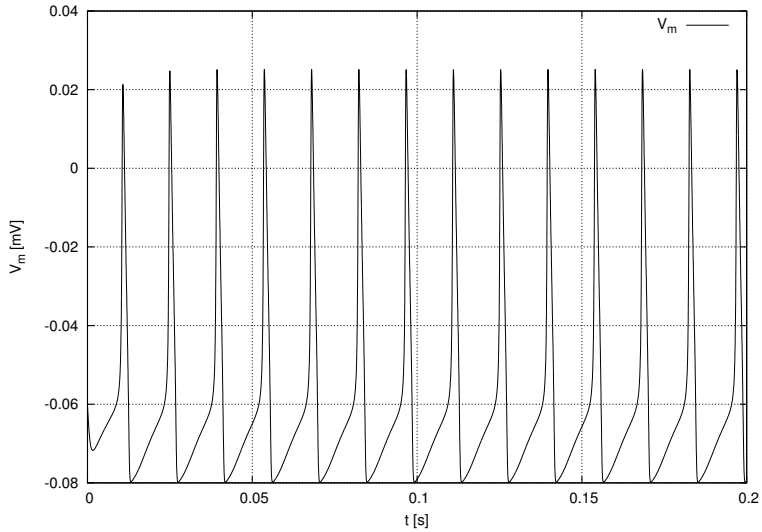
Rys. 5.3 przedstawia wykres zmieniającego się potencjału błony komórkowej w pierwszych 200 ms symulowanej aktywności biologicznej neuronu. Wykres ten uzyskujemy przez dopisanie do skryptu **gnuplota** dwóch linijek:

```
set output "wykres3.eps"
plot [0:0.2] "soma3.vm" u 1:2 w l t "V_{m}"
```

i ponownym uruchomieniu generatora wykresów z nazwą pliku ze skryptem jako parametrem.

5.5. Automatyzacja modelowania komórek

Budowę nawet bardzo złożonych komórek nerwowych da się zautomatyzować. W tym celu wykorzystujemy mechanizm wbudowany w GENESIS, tak zwany **Cell Reader**.



Rysunek 5.3. Potencjał błony komórkowej z kanałami jonowymi i z uwzględnieniem jednostek SI.

Definicje komórek umieszcza się w pliku o specjalnej strukturze i rozszerzeniu **.p**. Rozważmy następujący plik:

```
// cell.p - Cell parameter file used in Tutorial #5

// Format of file :
// x,y,z,dia are in microns, all other units are SI (Meter K
// In polar mode 'r' is in microns, theta and phi in degrees
// Control line options start with a '*'
// The format for each compartment parameter line is :
//name parent r theta phi d ch dens
//in polar mode, and in cartesian mode :
//name parent x y z d ch dens
// For channels, "dens" =  maximum conductance per unit area
// For spike elements, "dens" is the spike threshold
// Coordinate mode
*relative
*cartesian
*asymmetric

// Specifying constants
*set_compt_param RM 0.33333
*set_compt_param RA 0.3
```

```
*set_compt_param CM 0.01
*set_compt_param     EREST_ACT -0.07

// For the soma, use the leakage potential (-0.07 + 0.0106)
*set_compt_param     ELEAK -0.0594
soma none 30 0 0 30 Na_squid_hh 1200 K_squid_hh 360 spike 0.

// The dendrite has no H-H channels, so ELEAK = EREST_ACT
*set_compt_param     ELEAK -0.07
dend soma 100 0 0 2 Ex_channel 0.795775
```

W tym przypadku zachowaliśmy oryginalne i wiele mówiące komentarze napisane przez twórców GENESIS. Z naszego punktu widzenia najważniejsze są dwie linie rozpoczynające się od słów **soma none** i **dend soma** znajdujące się pod koniec pliku. W ten sposób informujemy **Cell Readera**, że **soma** nie posiada elementu nadrzędnego, natomiast dendryt **dend** wychodzi z **soma**. Zwróćmy też uwagę na kanały wzbudzający **Ex_channel** i hamujący **Inh_channel** zdefiniowane w obrębie dendrytu. W analogiczny sposób możemy konstruować komórki zawierające wiele wielokrotnie rozgałęziających się dendrytów. Pozostałe linie pliku **cell.p** zawierają standardowe wartości parametrów Hodgkina–Huxleya i przynajmniej na początku nie powinny być zmieniane przez niedoświadczonych użytkowników.

Korzystanie z mechanizmu **Cell Reader** niesie najwięcej zalet wtedy, gdy w modelu różne występują komórki. Wtedy każda komórka zdefiniowana jest w osobnym pliku. Wczytanie komórki podczas symulacji dokonuje się przez wykonanie polecenia **readcell** i wcześniejszym załączeniu biblioteki **protodefs**:

```
include protodefs.g
readcell cell.p /cell
```

Uwaga: Bibliotekę **protodefs.g** musimy najpierw przekopiować do katalogu z tworzonym modelem z katalogu **genesis/Scripts/tutorials**[1]. Cieszmy się zatem, że nie musimy już za każdym razem wpisywać bloku ze zmiennymi w układzie SI, definiować funkcji tworzącej elementy itp. Cały skrypt wykorzystujący **Cell Reader** przyjmuje następującą postać:

```
float dt = 0.00005
setclock  0  {dt}
```

[1] Jeżeli instalowaliśmy GENESIS z gotowego pakietu **deb**, to biblioteka **protodefs.g** może znajdować się w trudnym do znalezienia katalogu, gdzieś w głębokich czeluściach systemu operacyjnego. Istnieje duża szansa, że uda nam się ją przekopiować do katalogu bieżącego za pomocą polecenia: **cp /usr/share/genesis-2.3/Scripts/tutorials/protodefs.g .** i gotowe.

```
include protodefs.g
readcell cell.p /cell

setfield /cell/soma inject 0.3e-9

create asc_file /soma_potential
useclock /soma_potential 0
setfield /soma_potential filename soma4.vm append 0
addmsg /cell/soma /soma_potential SAVE Vm

reset
check

step 1 -time
quit
```

Po wykonaniu skryptu w katalogu z modelem powstaje plik **soma4.vm**, który możemy wykreślić.

5.6. Zapis czasu powstawania piku

W symulacji komórek nerwowych często nie zależy nam na dokładnym odwzorowaniu przebiegu wartości potencjału błony, a jedynie na zarejestrowaniu czasu występowania poszczególnych pików potencjału czynnościowego. Podejście takie pozwala znacząco skrócić czas symulacji i oszczędza miejsce na dysku. GENESIS pozwala na zapis omawianej wielkości w stosunkowo prosty sposób. Fragment skryptu:

```
create spikehistory /soma_spikes
setfield ^ filename soma4.spike append 0 ident_toggle 1
addmsg  /cell/soma/spike /soma_spikes SPIKESAVE
```

tworzy historię pojawiania się pików w obiekcie **soma_spikes**, który z kolei kojarzy z plikiem (tu. **soma4.spike**) na dysku i przesyła wiadomość **SPIKESAVE** z elementu **spike** ciała komórki do rzeczonego obiektu z historią.

Plik **soma4.spike** ma postać:

```
/cell/soma/spike     0.348050
/cell/soma/spike     0.363750
/cell/soma/spike     0.379450
/cell/soma/spike     0.395150
/cell/soma/spike     0.410800
/cell/soma/spike     0.426500
```

gdzie w prawej kolumnie zapisane są czasy występowania pików potencjału czynnościowego.

5.7. Podsumowanie

Przedstawiono szczegółowo zasadę projektowania modelu ciała pojedynczej komórki nerwowej. Opisano również mechanizm **Cell Reader** pozwalający w łatwy spsoób konstruować komórki zawierające dendryty. załączono treść najważniejszych skryptów oraz opisano metody wizualizacji zapisanych na dysk wyników z wykorzystaniem programu **gnuplot**.

5.8. Zadania

Zadanie 1

Napisz skrypt zawierający ciało komórki z wbudowanymi kanałami jonowymi sodowymi i potasowymi w standardzie jednostek SI. Unikaj stosowania metody „kopiuj – wklej".

Zadanie 2

Napisz skrypt, który tworzy dwie komórki zbudowane z ciała i dwóch dendrytów każda z wykorzystaniem mechanizmu **Cell Reader**. Zapisz aktywność obu komórek na dysk i wykreśl przebiegi wartości potencjału błony w pierwszych 300 ms symulacji.

Zadanie 3

Wzbogać skrypt z **Zadania 2** o zapis czasów pojawiania się pików potencjału czynnościowego.

Rozdział 6

Modelowanie prostych sieci neuronowych w GENESIS

6.1. Wprowadzenie

W tym rozdziale zapoznamy się z metodologią tworzenia połączeń między komórkami. W pierwszej sekcji zdefiniujemy funkcję przydatną w tworzeniu synaps. Następnie poznamy generator losowych pobudzeń, który można będzie podłączać za pomocą synapsy do modelowanych komórek. Najprostsza sieć neuronowa będzie zbudowana z dwóch elementów. Nauczymy się ponadto w sposób świadomy konstruować modele dwuwymiarowych i trójwymiarowych sieci neuronowych oraz zapisywać i wizualizować ich aktywność w środowisku **gnuplot**.

6.2. Tworzenie synaps

Idea synapsy została wbudowana bezpośrednio w symulator i podczas implementacji połączeń w model nie musimy martwić się o szczegółowy i skomplikowany numerycznie mechanizm Hodgkina–Huxleya, podobnie jak nie musimy wprost kodować równań różniczkowych opisujących zachowanie fragmentów poszczególnych komórek.

Każda synapsa użyta w GENESIS charakteryzuje się dwoma parametrami: wagą i opóźnieniem synaptycznym wyrażanym w sekundach. Do stworzenia synapsy warto napisać funkcję **make_synapse**:

```
function make_synapse(pre,post,weight,delay)
 str pre,post
 float weight,delay
 int syn_num

   addmsg {pre} {post} SPIKE
   syn_num = {getfield {post} nsynapses} - 1
   setfield {post} synapse[{syn_num}].weight {weight} \
                   synapse[{syn_num}].delay {delay}
   echo {pre} "--->" {post} {weight} {delay}

end
```

W powyższym kodzie **pre** i **post** to argumenty, w których do funkcji przesyłane są elementy presynaptyczny i postsynaptyczny. Ponadto parametrami funkcji są waga **weight** i opóźnienie **delay**. Zmienna **syn_num** jest pomocnicza i służy do przechowywania liczby synaps dochodzących do danej komórki w taki sposób, by tworzona nowa synapsa zajęła pierwsze, wolne i dostępne miejsce w strukturze synaps modelowanego neuronu. Po utworzeniu synapsy na konsolę GENESIS wysyłamy komunikat informacyjny o połączonych właśnie elementach presynaptycznym i postsynaptycznym.

6.3. Generator losowych pobudzeń

We wszelkiego rodzaju modelach dobrze jest zapewnić zmieniające się w czasie pobudzenie wybranej komórki bądź grupy komórek. W tym celu generujemy losowo zmieniające się w czasie ciągi pobudzeń symulujące piki potencjałów czynnościowych wirtualnych komórek (ang. *random spike trains*) za pomocą obiektu **randomspike**. Deklaracja losowego ciągu tego typu impulsów może wyglądać następująco:

```
create randomspike /input
setfield ^ min_amp 1 max_amp 1 rate 200 reset 1 reset_value 0
```

Poszczególne pola generatora oznaczają:
— **min_amp** – minimalną amplitudę zdarzenia,
— **max_amp** – maksymalną amplitudę zdarzenia,
— **rate** – średnią częstość generatora wyrażoną w Hz,
— **reset** – flagę włączającą bądź wyłączającą resetowanie generatora po każdym zdarzeniu,
— **reset_value** – wartość do jakiej resetować,
— **state** – aktualny stan obiektu,
— **abs_refract** – minimalny czas między zdarzeniami.
Pokazuje to, że w przypadku stworzonego przed chwilą generatora o nazwie **input** użyliśmy tylko kilku z nich.

Warto też zwrócić uwagę na symbol daszka oznaczający w języku skryptowym GENESIS odwołanie do poprzednio zdefiniowanego obiektu.

Załóżmy, że w pewnym modelu tworzymy przy pomocy mechanizmu **Cell Reader** komórkę **cell** składającą się z ciała i jednego dendrytu nazwanego **dend**. Podłączenie generatora impulsów **input** do kanału wzbudzającego dendrytu komórki **cell** przyjmuje następującą postać:

```
make_synapse /input /cell/dend/Ex_channel 2 0
```

Zauważmy, że opóźnienie synaptyczne między generatorem a komórką wynosi 0 przy wadze połączenia standardowo w takich sytuacjach ustawionej na 2. Waga informuje nas w pewnym sensie o stopniu wzmocnienia sygnału wysyłanego przez generator bądź inną komórkę. W zależności od parametrów komórki postsynaptycznej – wagi o konkretnej wartości mogą doprowadzić do jej wzbudzenia, inne zaś nie wzmacniają sygnału komórki presynaptycznej do takiego stopnia, który wywołałby wyładowanie. Cały zaś skrypt (przy założeniu, że korzystamy z pliku **cell.p**, w którym definiujemy odpowiednią komórkę) przedstawia się następująco:

```
float dt = 0.00005
setclock  0  {dt}
```

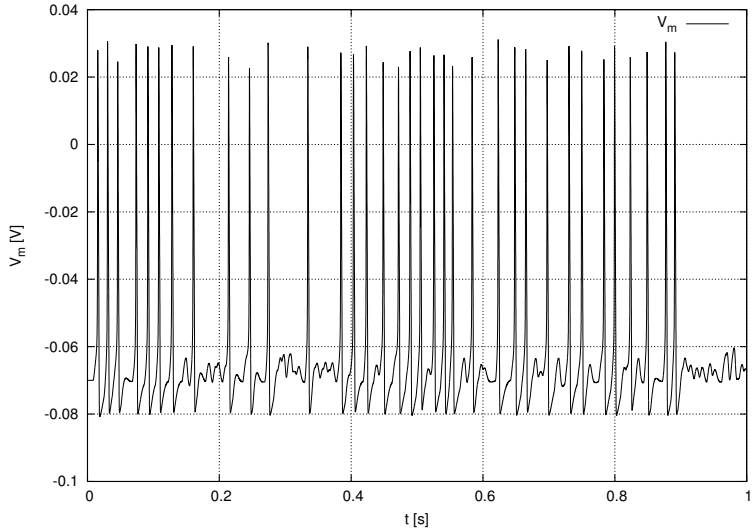
```
include functions.g // nasze zdefiniowane funkcje
include protodefs.g

readcell cell.p /cell // plik z definicja komorki

randseed // generator liczb pseudolosowych

create randomspike /input
setfield ^ min_amp 1 max_amp 1 rate 200 reset 1 \
           reset_value 0

make_synapse /input /cell/dend/Ex_channel 2 0

create asc_file /soma_potential
useclock /soma_potential 0
setfield /soma_potential filename soma5.vm append 0
addmsg /cell/soma /soma_potential SAVE Vm

create spikehistory /soma_spikes
setfield ^ filename soma.spike append 0 ident_toggle 1
addmsg  /cell/soma/spike /soma_spikes SPIKESAVE

reset
check

step 1 -time
quit
```

Należy zauważyć, że w trzeciej linii korzystamy ze znanego już polecenia **include** za pomocą którego dołączamy do skryptu plik **functions.g**, w którym będą znajdować się definicje wykorzystywanych przez nas funkcji. W omawianym przypadku w pliku tym powinna być już obecna definicja funkcji tworzenia synapsy. Czytelnicy niebawem zauważą, że warto posiadać swój osobisty plik **functions.g**, który będzie rozrastał się i wzbogacał o nowe funkcje wraz z osobistym zaawansowaniem w GENESIS.

Ponadto użyte zostało polecenie **randseed** inicjujące generator liczb pseudolosowych, konieczne do odmiennego za każdym razem działania generatora **input**.

Po wykonaniu skryptu powstaną pliki **soma.vm** z przebiegiem potencjału na ciele komórki oraz **soma.spike** z zanotowanym czasem wystąpienia pików potencjału czynnościowego w komórce **cell**.

Przebieg potencjału na modelowanej w ten sposób komórce przedstawiono na Rys. 6.1



Rysunek 6.1. Potencjał błony komórkowej neuronu podłączonego do generatora typu **randomspike**

6.4. Sieć zbudowana z dwóch komórek

Na obecnym poziomie zaawansowania możemy pokusić się o stworzenie najprostszej sieci neuronów Hodgkina–Huxleya zawierającej dwie połączone komórki i generator. Niech będzie dana komórka A z jednym dendrytem podłączona do generatora losowych impulsów oraz taka sama komórka B podłączona do komórki A.

Za pomocą znanych nam metod powinniśmy stworzyć dwie komórki korzystając z **Cell Readera** oraz generator. Następnie łączymy stworzony generator z kanałem wzbudzającym dendrytu komórki A z wagą $w = 0$ i opóźnieniem $d = 0$. Podłączając komórkę B do A wiążemy synapsą kanał wzbudzający jej jedynego dendrytu z polem **spike** ciała neuronu presynaptycznego z wagą $w = 2.8$ i opóźnieniem $d = 0.0004$ s. Należy jeszcze tylko zapewnić zapis aktywności obu komórek do plików **soma_A.vm**, **soma_B.vm**, **soma_A.spike** i **soma_B.spike**, sprawdzić wewnętrzną spójność modelu **check**, zresetować sieć **reset** i uruchomić symulację jednej sekundy aktywności biologicznej takiego układu **step 1 -time**. Gotowy skrypt powinien wyglądać następująco:

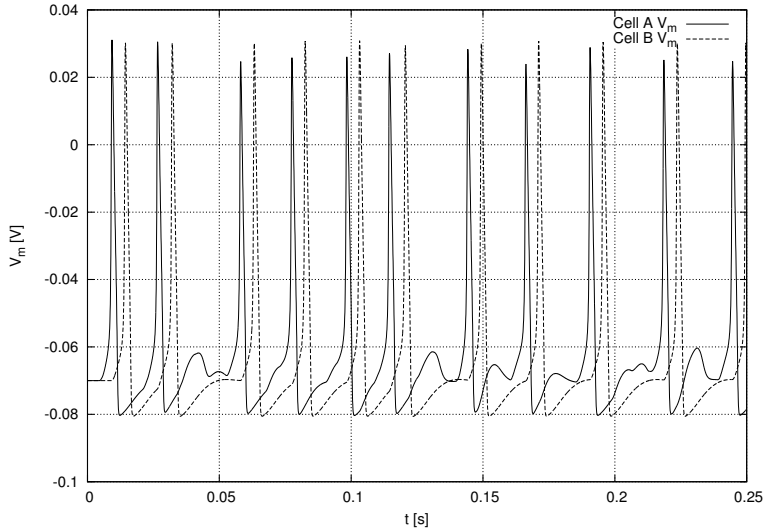
```
float dt = 0.00005            // simulation time step in sec
```

```
setclock  0  {dt}       // set the simulation clock

include functions.g
include protodefs.g

readcell cell.p /cell_A
readcell cell.p /cell_B

randseed

create randomspike /input
setfield ^ min_amp 1 max_amp 1 rate 200 reset 1 \
           reset_value 0

make_synapse /input /cell_A/dend/Ex_channel 2 0
make_synapse /cell_A/soma/spike \
             /cell_B/dend/Ex_channel 2.8 1e-04

create asc_file /soma_A_potential
useclock /soma_A_potential 0
setfield /soma_A_potential filename soma_A.vm append 0
addmsg /cell_A/soma /soma_A_potential SAVE Vm

create asc_file /soma_B_potential
useclock /soma_B_potential 0
setfield /soma_B_potential filename soma_B.vm append 0
addmsg /cell_B/soma /soma_B_potential SAVE Vm

create spikehistory /soma_A_spikes
setfield ^ filename soma_A.spike append 0 ident_toggle 1
addmsg  /cell_A/soma/spike /soma_A_spikes SPIKESAVE

create spikehistory /soma_B_spikes
setfield ^ filename soma_B.spike append 0 ident_toggle 1
addmsg  /cell_B/soma/spike /soma_B_spikes SPIKESAVE

reset
check

step 1 -time
quit
```

Przebieg wartości potencjału na obu modelowanych komórkach przedstawiono na Rys. 6.2. Wykres wygenerowano wydając polecenie:

```
plot [0:0.25] "soma_A.vm" u 1:2 w l t "Cell A V_{m}", \
              "soma_B.vm" u 1:2 w l t "Cell B V_{m}"
```

w programie **gnuplot**.Warto zwrócić uwagę na związane z opóźnieniem synaptycznym przesunięcie pików potencjału czynnościowego komórki B względem przebiegu komórki A.



Rysunek 6.2. Potencjał błon komórkowych neuronów **cell_A** i **cell_B** w pierwszych 250 ms symulacji

6.5. Sieć dwuwymiarowa I

Potrafimy zbudować już sieć z dwóch neuronów. W tym miejscu warto zastanowić się jak budować sieci zawierające wiele takich samych komórek i połączonych według określonej reguły. Z punktu widzenia zagadnień równoleglizacji modeli warto budować dwuwymiarowe struktury na prostokątnych siatkach. Zakładamy, że neurony w siatce będą indeksowane liczbami całkowitymi. W tym celu gdzieś na początku modelu warto zadeklarować globalne zmienne **array_minx** oraz **array_miny** określające początkowe indeksy komórek w wierszach i kolumnach. Przez analogię określamy też zmienne **sep_x** i **sep_y**, które determinują odległość poszczególnych komórek wzdłuż obu osi.

Ponieważ często będziemy tworzyć sieci dwuwymiarowe – warto do biblioteki naszych własnych funkcji dopisać funkcję **make_circuit_2d**. Nowa funkcja jako argumenty przyjmuje prototyp komórki do kopiowania, pierwszy człon nazwy tworzonej sieci oraz jej wymiary.

```
function make_circuit_2d(protocell, net, nx, ny)
str protocell
int i,j

 for (i=1; i<={nx};i={i+1})
  for (j=1; j<={ny};j={j+1})

 copy {protocell} {net}_{i}_{j}

 position {net}_{i}_{j} { {array_minx} + ({sep_x} * {i}) } \
                        { {array_miny} + ({sep_y} * {j}) } \
                        { 0 }
  end
 end

end
```

W funkcji tej korzystamy z podwójnej, zagnieżdżonej pętli **for** oraz funkcji **copy** i **position** odpowiednio tworzących i ustawiających kolejne komórki na siatce w węzłach indeksowanych przez **i** oraz **j**.

Zapis aktywności sieci wykonujemy korzystając ze zdobytej dotychczas wiedzy i tworzymy funkcję, która automatyzuje cały proces:

```
function make_circuit_2d_output(net, nx, ny, filename)
int i,j

 create spikehistory {net}-history
 setfield ^ filename {filename}.spike append 0 ident_toggle 1

    for (i=1; i<={nx}; i={i+1})
     for (j=1; j<={ny}; j={j+1})

     addmsg {net}_{i}_{j}/soma/spike {net}-history SPIKESAVE

     end
    end

echo {net} spike activity saved to file {filename}.spike
```

```
end
```

Zauważmy, że jako argument funkcji - oprócz pierwszego członu nazwy sieci, z której zbieramy aktywność i jej wymiarów podajemy nazwę pliku **filename**, do której w ciele funkcji dołączone zostanie jeszcze rozszerzenie **spike**. Po wykonaniu symulacji w plikach o tym rozszerzeniu będzie zapisana aktywność poszczególnych komórek sieci.

Przykładowy fragment kodu wykorzystujący obie funkcje i konieczne deklaracje może wyglądać następująco:

```
include protodefs.g
int array_minx =  1
int array_miny =  1

float sep_x = 40e-6
float sep_y = 40e-6

include functions.g

readcell cell.p /cell
make_circuit_2d /cell /net2d 16 16
make_circuit_2d_output /net2d 16 16 rec-net2d
```

Fragment kodu, który powinien zrealizować 10% tzw. połączenia pełnego (ang. *full connection*) może wyglądać następująco:

```
randseed
float probability = 0.1
int i1, j1, i2, j2

for (i1=1; i1<=16; i1={i1+1})
 for (j1=1; j1<=16; j1={j1+1})
   for (i2=1; i2<=16; i2={i2+1})
     for (j2=1; j2<=16; j2={j2+1})

       if ( {rand 0 1} < {probability} )
        make_synapse /net2d_{i1}_{j1}/soma/spike \
                    /net2d_{i2}_{j2}/dend/Ex_channel 2.8 1e-04
       end

    end
   end
  end
end
```

Mamy tu aż cztery zagnieżdżone pętle, bo po każdej komórce algorytm musi przecież przejść dwa razy – traktując ją jako presynaptyczną i postsynaptyczną. Synapsy tworzymy korzystając ze zdefiniowanej wcześniej funkcji. Oprócz tego deklarujemy zmienną określającą prawdopodobieństwo, a losowania dokonujemy przy pomocy funkcji **randseed** z określonym przedziałem jako parametrem, pamiętając o wcześniejszej inicjacji generatora liczb pseudolosowych **randseed** (Uwaga: wystarczy to uczynić tylko raz w programie). Zakładając, że komórki wykorzystywane do tworzenia opisywanych tu sieci zostały zaprojektowane i sparametryzowane tak jak w niniejszym podręczniku - optymalna waga dla połączeń między nimi wynosi **w=2.8**, a opóźnienie **d=0.0001 s**. Pamiętajmy też, że żeby uzyskać jakikolwiek zapis aktywności sieci – powinniśmy zapewnić stymulację wybranego neuronu lub neuronów z zewnętrznego źródła, na przykład z generatora losowych impulsów.

```
create randomspike /input
setfield ^ min_amp 1 max_amp 1 rate 200 reset 1 \
reset_value 0

make_synapse /input /net2d_8_8/dend/Ex_channel 2 0
```

6.6. Sieć trójwymiarowa I

W analogiczny sposób budujemy sieci trójwymiarowe. Sieci takie tworzymy na siatce prostopadłościanu (mimo to często nazywane są kolumnami neuronalnymi). Przyjemność z analizy treści funkcji **make_circuit_3d** pozostawiamy czytelnikowi.

```
function make_circuit_3d(protocell, net, nx, ny, nz)
str protocell
int i,j,k

 for (i=1; i<={nx};i={i+1})
  for (j=1; j<={ny};j={j+1})
    for (k=1; k<={nz};k={k+1})

copy {protocell} {net}_{i}_{j}_{k}

position {net}_{i}_{j}_{k} \
                      { {array_minx} + ({sep_x} * {i}) } \
                      { {array_miny} + ({sep_y} * {j}) } \
                      { {array_minz} + ({sep_z} * {k}) }
```

```
    end
  end
 end
end
```

Podobnie do przypadku 2D zapewniamy zapis aktywności trójwymiarowej sieci do pliku:

```
function make_circuit_3d_output(net, nx, ny, nz, filename)
int i,j,k

create spikehistory {net}-history
setfield ^ filename {filename}.spike append 0 ident_toggle 1

    for (i=1; i<={nx}; i={i+1})
        for (j=1; j<={ny}; j={j+1})
            for (k=1; k<={nz};k={k+1})

             addmsg {net}_{i}_{j}_{k}/soma/spike \
                    {net}-history SPIKESAVE

            end
        end
    end

echo {net} spike activity saved to file {filename}.spike
end
%
```

Przykładowy fragment kodu wykorzystujący obie funkcje i konieczne deklaracje może wyglądać następująco:

```
include protodefs.g
int array_minx =  1
int array_miny =  1
int array_minz =  1

float sep_x = 40e-6
float sep_y = 40e-6
float sep_z = 40e-6

include functions.g
readcell cell.p /cell
make_circuit_3d /cell /net3d  2 3 4
```

```
make_circuit_3d_output /net3d 2 3 4 rec-net3d
```

Łączenie zaś komórek w obrębie kolumny z określonym prawdopodobieństwem wymaga zagnieżdżenia sześciu pętli:

```
randseed
float probability = 0.1
int i1, i2, j1, j2, k1, k2

for (i1=1; i1<=2; i1={i1+1})
 for (j1=1; j1<=3; j1={j1+1})
   for (k1=1; k1<=4; k1={k1+1})
     for (i2=1; i2<=2; i2={i2+1})
      for (j2=1; j2<=3; j2={j2+1})
        for (k2=1; k2<=4; k2={k2+1})

       if ( {rand 0 1} < {probability} )

     make_synapse /net3d_{i1}_{j1}_{k1}/soma/spike \
      /net3d_{i2}_{j2}_{k2}/dend/Ex_channel 2.8 1e-04

      end
     end
    end
   end
  end
 end
end
```

Czytelnik zechce zauważyć, że wagi połączeń między konkretnymi komórkami nie muszą być zawsze stałe. Stosunkowo łatwo uzależnić je na przykład od odległości euklidesowej lub innych, własnoręcznie zdefiniowanych parametrów modelu. Oczywiście musimy pamiętać, że w celu obserwacji niezerowej aktywności sieci – musimy zapewnić stymulację wybranego neuronu lub grupy neuronów z zewnętrznego źródła, na przykład z generatora losowych impulsów.

6.7. Alternatywny sposób tworzenia sieci 2D

Warto wiedzieć, że GENESIS oferuje możliwość konstruowania dwuwymiarowych sieci neuronowych przy pomocy funkcji **createmap**. Przykładowe wywołanie tej funkcji ma postać:

```
createmap /cell /net2d 20 5 -delta {sep_x} {sep_y}
```

gdzie utworzono sto komórek na siatce o wymiarach 20×5 i oddalonych o predefiniowane przez nas wcześniej odległości. Niestety z punktu widzenia struktury danych w modelu – komórki te przechowywane są w tablicy indeksowanej od **0** do **99**.

Zapis aktywności utworzonej w ten sposób sieci zapewniamy wykorzystując napisaną specjalną funkcję:

```
function make_output_2d(net, filename)
 str net, filename

create spikehistory {net}-spikes
setfield ^ filename {filename} append 0 ident_toggle 1
addmsg {net}[]/soma/spike {net}-spikes SPIKESAVE
end
```

gdzie warto zwrócić uwagę na puste w środku nawiasy kwadratowy [] służące do odwoływania się do sieci jako całości. Wywołanie funkcji z poziomu głównego skryptu modelu ma postać:

```
make_output_2d /net2d/cell net2d.spike
```

Połączenia w sieci zapewniamy przez wywołanie funkcji **make_synapse**, na przykład łączymy drugą komórkę z siódmą wykonując polecenie:

```
make_synapse /net2d/cell[2]/soma/spike \
             /net2d/cell[7]/dend/Ex_channel \
             {weight} {delay}
```

lub w dowolny inny zautomatyzowany sposób.

GENESIS zapewnia również interesujące metody nadawania struktury wag i opóźnień dla sieci jako całości tworzonych przy pomocy finkcji **createmap**. Polecenia **planarweight** i **planardelay** zostały szczegółowo opisane w osiemnastym rozdziale [19].

6.8. Podsumowanie

W tym rozdziale przedstawiono metody tworzenia sieci neuronowych. Po zapoznaniu się z tym rozdziałem czytelnik powinien potrafić zbudować zarówno proste sieci zawierające kilka komórek jak również skomplikowane dwu– i trójwymiarowe struktury. Przedstawiono treść kilku pożytecznych funkcji: do tworzenia synaps, obwodów dwuwymiarowych, trójwymiarowych i zapisywania aktywności. Potrafimy też tworzyć generatory losowych impulsów i wizualizować aktywność wybranych komórek przy pomocy programu **gnuplot**.

6.9. Zadania

Zadanie 1

Stwórz model, który zawiera trzy komórki: A, B i C. Komórka A jest podłączona do generatora, komórka B pobiera pobudzenia od komórki A, a komórka C od komórki B. Napisz skrypty, które przygotują zapis aktywności poszczególnych komórek w odrębnych plikach **.spike** oraz przebiegi potencjałów w plikach **.vm**.

Zadanie 2

Napisz model zawierający tablicę 100 komórek ułożonych na kwadratowej siatce 10×10. Zrealizuj 15% połączenia pełnego. Zapisz aktywność sieci do pliku **.spike**.

Zadanie 3

Napisz model zawierający tablicę 100 komórek ułożonych na kwadratowej siatce 10×10 korzystając z funkcji **createmap**. Zrealizuj 25% połączenia pełnego. Zapisz aktywność sieci do pliku **.spike**. Do realizacji połączeń możesz wykorzystać dowolną metodę: przedstawioną w tym rozdziale lub **planarweight** i **planardelay**. Podłącz trzy wybrane komórki do generatora.

Zadanie 4

Napisz model zawierający tablicę 512 komórek ułożonych na trójwymiarowej siatce $8 \times 8 \times 8$. Uwaga w symulowanej sieci zaaranżuj 80% połączeń wzbudzających (do **dend/Ex_channel**) i 20% hamujących (do **dend/Inh_channel**). Zapisz aktywność sieci do pliku **.spike**. Podłącz pięć wybranych komórek do generatora.

Rozdział 7

Wizualizacja aktywności sieci

7.1. Wprowadzenie

W tym rozdziale nauczymy się wizualizować aktywność symulowanych sieci neuronowych za pomocą wbudowanych w GENESIS gotowych narzędzi. O ile wizualizacja na przykład zmian potencjału wybranej komórki może być z łatwością przeprowadzona w środowisku **gnuplot**, a wykorzystywane przez GENESIS obiekty XODUS wydają się być przestarzałe, o tyle wizualizacja w postaci mapy dwuwymiarowej dynamiki neuronalnej sieci zawierającej kilkadziesiąt neuronów w symulatorze GENESIS wygląda dostatecznie ładnie i czasami nie warto tracić czasu na tworzenie nowej aplikacji li tylko do pokazania migoczących kwadratów.

7.2. Przykładowa sieć

Zanim pokażemy jak wizualizować aktywność komórek stworzymy prosty model zawierający dwie sieci **net1** i **net2**, w każdej po 64 neurony rozmieszczone na siatce o wymiarach 8×8. W obrębie każdej sieci, a także między nimi prawdopodobieństwo utworzenia synapsy ustalono na 10%. Sieci tworzymy korzystając z poznanego wcześniej polecenia **createmap** rozmieszczając komórki na mapach w odległościach **sep_x** o **sep_y**. Aktywność sieci zapisujemy do plików z rozszerzeniem **.spike** – funkcja **make_mapoutput**. Do ustawienia losowych połączeń w obrębie każdej sieci wykorzystujemy funkcję **connect_map**, do połączenia pierwszej sieci z drugą korzysta się z funkcji **connect_two_maps**. Stymulujemy zerową komórkę pierwszej sieci z generatora losowych impulsów. Symulujemy jedną sekundę aktywności biologicznej systemu. Opisany tu model tworzony jest w następującym skrypcie:

```
float sep_x=0.001  // odl. komorek w  kierunku x w [m]
float sep_y=0.001  // odl. komorek w  kierunku y w [m]

float genrate=200

setclock 0 0.000005
setclock 1 {1.0/{genrate}}  // for generator

include functions-wizualizacja.g
include protodefs.g

readcell cell.p /cell

createmap /cell /net1 8 8 -delta {sep_x} {sep_y}
```

```
createmap /cell /net2 8 8 -delta {sep_x} {sep_y}

create randomspike /input
setfield ^ min_amp 1 max_amp 1 rate {genrate} reset 1 \
          reset_value 0 abs_refract 0
useclock /input 1

create spikehistory /input-spikes
setfield ^ filename input.spike append 0 ident_toggle 1
addmsg /input /input-spikes SPIKESAVE

connect_map /net1 8 8 0.1
connect_map /net2 8 8 0.1
connect_two_maps /net1 /net2 8 8 0.1

make_synapse /input /net1/cell[0]/dend/Ex_channel 10 0 0

make_mapoutput /net1/cell net1.spike
make_mapoutput /net2/cell net2.spike

check
reset

step 1 -time
```

gdzie wszystkie funkcje zawarte w pliku **functions_wizualizacja.g** przyjmują następującą postać:

```
function make_synapse(pre,post,weight,delay)
 str pre,post
 float weight,delay
 int syn_num

   addmsg {pre} {post} SPIKE
   syn_num = {getfield {post} nsynapses} - 1
   setfield {post} synapse[{syn_num}].weight {weight} \
           synapse[{syn_num}].delay {delay}
   echo {pre} "--->" {post} {weight} {delay}

end

function make_mapoutput(net, filename)
 str net, filename
```

```
create spikehistory {net}-spikes
setfield ^ filename {filename} append 0 ident_toggle 1
addmsg {net}[]/soma/spike {net}-spikes SPIKESAVE

end

function connect_map (net, nx, ny, prob)
int i,j, netdim

netdim = {nx} * {ny}

for (i=0;i<netdim;i=i+1)
 for (j=0;j<netdim;j=j+1)

   if ({rand 0 1} < {prob} )
   make_synapse {net}/cell[{i}]/soma/spike \
                {net}/cell[{j}]/dend/Ex_channel 1.8 1e-04
   end
end
end

end

function connect_two_maps (net1, net2, nx, ny, prob)
int i,j, netdim

netdim  = {nx} * {ny}

for (i=1;i<netdim;i=i+1)
 for (j=1;j<netdim;j=j+1)

   if ({rand 0 1} < {prob} )
    make_synapse {net1}/cell[{i}]/soma/spike \
                 {net2}/cell[{j}]/dend/Ex_channel 1.8 1e-04
   end
end
end

end
```

Po uruchomieniu opisywanego skryptu na dysku powstaną dwa pliki

net1.spike i **net2.spike**, w których zapisana zostanie aktywność obu sieci. Na razie bez śladu wizualizacji.

7.3. Wizualizacja

Korzystając z przykładów zamieszczonych w kodzie GENESIS do pliku z funkcjami trzeba teraz dopisać funkcję odpowiedzialną za wizualizację aktywność sieci:

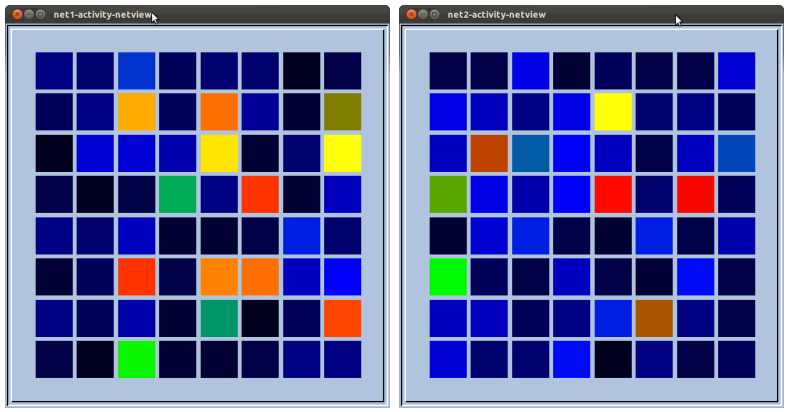
```
function make_netview (network, Nx, Ny, name, x, y, w, h)
// sets up xview widget to display Vm of each cell

int x,y,w,h

    create xform /{name}-netview [{x},{y},{w},{h}]
    create xdraw /{name}-netview/draw [0%,0%,100%, 100%]
// Make the display region a bit larger than the cell array
setfield /{name}-netview/draw \
        xmin {-sep_x} xmax {Nx*sep_x} \
ymin {-sep_y} ymax {Ny*sep_y}
create xview /{name}-netview/draw/view
setfield /{name}-netview/draw/view path \
         {network}/soma field Vm \
value_min -0.08 value_max 0.03 \
        viewmode colorview sizescale {0.9*sep_x}
    xshow /{name}-netview

end
```

a w kodzie zawierającym model jako taki trzeba ją po prostu wywołać:

```
make_netview /net1/cell[] 8 8 "net1-activity" 50 50 300 300
make_netview /net2/cell[] 8 8 "net2-activity" 350 50 300 300
```

określając jako parametry rozmiary sieci, tytuł okienka oraz współrzędne lewych górnych rogów wyświetlanych okienek wraz z ich rozmiarami w pionie i w poziomie.

Teraz wywołanie skryptu spowoduje pojawienie się na ekranie okienek wizualizujących aktywność z migoczącymi kwadracikami odpowiadającymi każdemu z symulowanych neuronów Rys. 7.1.



Rysunek 7.1. Wizualizacja aktywności sieci w XODUS.

7.4. Podsumowanie

Przedstawiono prostą metodę wizualizacji aktywności sieci neuronów modelowanych w GENESIS skompilowanym z bibliotekami graficznymi XODUS. często taka wizualizacja wystarcza, żeby prześledzić podstawowe zachowania występujące w badanym modelu. W bardziej skomplikowanych przypadkach zaleca się tworzenie własnej, odrębnej aplikacji z danymi wejściowymi w postaci plików **.spike** i wizualizującej aktywność sieci w pożądany sposób.

7.5. Zadania

Zadanie 1

Stwórz model zawierający trzy kwadratowe sieci **net1**, **net2** oraz **net3** o wymiarach 10×10 połączone wewnątrz i między sobą z pewnym, zdefiniowanym przez ciebie prawdopodobieństwem. Wizualizuj aktywność modelu.

Zadanie 2

Zmodyfikuj funkcje z poprzedniego zadania w taki sposób, żeby tworzone i wizualizowane sieci mogły być prostokątne. Stwórz wizualizację aktywności sieci 8×5.

Zadanie 3

Napisz aplikację zewnętrzną służącą do wizualizacji aktywności sieci na podstawie plików **.spike**.

Rozdział 8

Instalacja i konfiguracja PGENESIS

8.1. Wprowadzenie

Prawdziwą potęgę GENESIS pokazuje w wersji równoległej. Dysponując klastrami obliczeniowymi można projektować modele zawierające dziesiątki, a nawet setki tysięcy neuronów z liczbą synaps rzędu miliona. Ograniczeniem jest oczywiście zdolność obliczeniowa maszyny, na której symulacja zostanie uruchomiona. Niemniej jednak już wykorzystanie dwurdzeniowych procesorów pozwala w sposób istotny zmniejszyć czas wykonywanych obliczeń numerycznych, czyli szybciej rozwiązać duży układ równań różniczkowych Hodgkina–Huxleya. W tym rozdziale nauczymy się kompilować ze źródeł równoległą odmianę GENESIS zwaną PGENESIS oraz przygotować komputer wielordzeniowy do prowadzenia symulacji równoległych.

8.2. Instalacja zależności

Zanim rozpoczniemy instalacje PGENESIS i konfigurację maszyny równoległej powinniśmy doinstalować do systemu kilka programów (kompilator **g++** i powłokę **csh**) oraz bibliotek graficznych wraz z ich narzędziami deweloperskimi. Instalacja powyższych jest warunkiem koniecznym do poprawnej kompilacji środowiska w systemie operacyjnym Ubuntu 10.04 i wyższych.

W konsoli systemu wykonujemy polecenie:

```
sudo apt-get install csh g++ libxt-dev libxt6 libxtst6 \
libxtst-dev libxmu-dev
```

8.3. Instalacja MPICH2

Komputer równoległy zbudujemy w oparciu o ogólnodostępną, darmową i przenośną implementację standardu MPI dla systemu Linux. W tym celu skorzystamy z biblioteki MPICH2 [26]. Instalacja MPICH2 w Ubuntu jest niezwykle prosta. Wystarczy wykonać polecenie:

```
sudo apt-get install mpich2
```

Nie zawsze jednak to co najprostsze jest najlepsze. Autor z własnego doświadczenia może potwierdzić, że najlepszym rozwiązaniem będzie kompilacja najnowszej wersji biblioteki wprost ze źródeł. Pozwoli to uniknąć często nieprzewidzianych i niepożądanych kłopotów. Problemy mogą wynikać bezpośrednio z uszkodzonych zależności między pakietami, niewiedzy lub niedostosowania predefiniowanej konfiguracji do potrzeb indywidualnych użytkowników, w tym wypadku programistów PGENESIS.

Źródła biblioteki można pobrać z [26] lub wykonując bezpośrdenio z konsoli polecenie (długi adres www oczywiście wpisujemy w jednej linii):

```
wget http://www.mcs.anl.gov/research/projects/mpich2/
            downloads/tarballs/1.4.1p1/mpich2-1.4.1p1.tar.gz
```

Po zapisaniu pliku na dysk proponujemy przeniesienie go do katalogu domowego użytkownika oraz rozpakowanie:

```
mv mpich2-1.4.1p1.tar.gz ~
cd
tar xvfz mpich2-1.4.1p1.tar.gz
```

W dalszej kolejności utworzymy katalog, w którym będzie rezydować instalacja MPICH2.

```
mkdir mpich2-install
```

Następnie należy wejść do katalogu zawierającego źródła biblioteki i rozpocząć proces konfiguracji:

```
./configure --prefix=/home/student/mpich2-install \
            --disable-f77 --disable-fc
```

Wykonując polecenie **./configure** informujemy konfigurator o położeniu katalogu, w którym znajdzie się skompilowana biblioteka oraz prosimy go by nie brał pod uwagę aspektów związanych z kompilatorem języka FORTRAN. Póki co nie będzie nam on w PGENESIS potrzebny.

Po udanej konfiguracji powinniśmy otrzymać na ekranie konsoli komunikat:

```
Configuration completed.
```

Bibliotekę MPICH2 należy teraz skompilować wydając polecenie:

```
make
```

i zainstalować:

```
make install
```

Otrzymujemy odpowiednio komunikaty:

```
Make completed
```

oraz

```
Installed MPE2 in /home/student/mpich2-install
```

Proces instalacji MPICH2 kończymy przez dopisanie w sekcji pliku **.bashrc** (rozpoczynającej się od **export PATH=**) definiującej ścieżkę dostępu następującej frazy:

```
:/home/student/mpich2-install/bin
```

bezpośrednio za ostatnim katalogiem zdefiniowanym w ścieżce.

8.4. Edycja Makefile

Proces instalacji PGENESIS rozpoczynamy od pobrania wersji równoległej środowiska bezpośrednio ze strony twórców [27]. Obecnie najnowsza wersja PGENESIS oznaczona jest numerem **2.3**. Powinniśmy więc pobrać plik o nazwie **pgenesis-2.3-src.tar.gz** (większość użytkowników pewnie już pobrała ten plik wcześniej). Plik ten należy przenieść do katalogu głównego i rozpakować:

```
mv pgenesis-2.3-src.tar.gz ~
tar xvfz pgenesis-2.3-src.tar.gz
```

Jeżeli instalacja i konfiguracja GENESIS w wersji standardowej przebiegła pomyślnie (a autor nawet nie dopuszcza myśli, że mogło tak nie być) to powinniśmy otrzymac katalog **pgenesis** w katalogu **genesis-2.3** obok katalogu **genesis** ze standardową wersją.

Wchodzimy do katalogu z źródłami wersji równoległej:

```
cd genesis-2.3/pgenesis
```

i kopiujemy plik **Makefile.dist** do **Makefile**:

```
cp Makefile.dist Makefile
```

Następnie należy dokonać edycji pliku **Makefile** w taki sposób, by przystosować go do używanego systemu operacyjnego.

Edycję rozpoczynamy od usunięcia komentarzy z linii **147** i **148** pliku **Makefile**. Musimy poinformować kompilator, że skorzystamy z biblioteki MPICH2. Po usunięciu komentarzy linie wyglądają tak:

```
 USE_MPI = 1
 MPI_LIB =
```

Linie od **159** do **164** również powinny zostać odkomentowane:

```
 MPI_CMD = 'mpirun -np $$num_nodes $$exec $$nargv'
 MPI_DEBUG_CMD = 'mpirun -gdb -np $$num_nodes $$exec $$nargv
# # these last three are needed even with MPI to keep the sc
 PVM_ROOT = $(PWD)
 PVM_ARCH = $(MACHINE)
 SET_PVM_ARCH = $(PVM_ARCH)
```

W dalszej kolejności należy odszukać sekcję konfiguracyjną Linuksa i usunąć odpowiednie komentarze w taki sposób, by fragment pliku **Makefile** wyglądał tak (wersja 32 bitowa systemu):

```
# ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
# System: Linux 1.2.x and up on Intel x86-based, Xeon,
```

```
#           and AMD 64-bit systems.
# Compiler: GCC
# ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~

## 2000-05-23
## Termcap/ncurses issues: The shell library makes reference
## termcap library.  Some Linux distributions have an ncurse
## which is includes termcap emulation.  GENESIS appears to
## properly with the ncurses supplied with Red Hat Linux 5.1
## and Debian Linux (glibc2.1, egcs-2.91.66).  However, link
## ncurses is known to have resulted in core dumps in GENESI
## Linux versions.
##
## If you encounter problems linking with the TERMCAP flags
## or the GENESIS command line interface does not work, try
## following alternatives:
##
## 1) TERMCAP = -ltermcap
##
## 2) (If you are using SuSE Linux)
##    TERMCAP = /usr/lib/termcap/libtermcap.a
##
## 3) (If you are using Red Hat Linux prior to version 6.0)
##    TERMCAP = /usr/lib/libtermcap.a
##

 MACHINE=Linux
 OS=BSD

 XINCLUDE=-I/usr/X11R6/include

## 64-bit machines probably need /usr/X11R6/lib64 here.
 XLIB=/usr/X11R6/lib

 CC=mpicc

## Old (and probably broken) gcc installations may need the
## path to cpp (preferably NOT one in /lib). If there isn't
## [link to] cpp in the same directory as 'cc', you should c
## [re]installing a newer gcc.
 CPP=cpp -P
 CFLAGS=-O2 -D__NO_MATH_INLINES
```

```
## For 64-bit architectures
# CFLAGS=-O2 -D__NO_MATH_INLINES -DLONGWORDS

 LD=ld

## !!!
## Don't uncomment the next line unless you get errors about
## libraries not being found. Setting this path may interfer
## the default (probably correct) operation of the loader, b
## 64-bit architectures may need /usr/lib64 here.
## LDFLAGS=-L/usr/lib

 RANLIB=ranlib
 AR=ar
 CPLIB=cp

 YACC=bison -y
 LEX=flex -l
 LEXLIB=-lfl
 LIBS= $(LEXLIB) -lm

 TERMCAP=-lncurses
 TERMOPT=-DTERMIO -DDONT_USE_SIGIO

## end Linux 1.2.x and up on Intel x86-based systems
```

Zwróćmy uwagę, że przypomina on znacząco analogiczny fragment pliku konfiguracyjnego klasycznego GENESIS.

Dla systemów 64 bitowych odpowiadająca sekcja w pliku **Makefile** wygląda tak:

```
# ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
# System: Linux 1.2.x and up on Intel x86-based, Xeon,
#           and AMD 64-bit systems.
# Compiler: GCC
# ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~  ~

## 2000-05-23
## Termcap/ncurses issues: The shell library makes reference
## termcap library.  Some Linux distributions have an ncurse
## which is includes termcap emulation.  GENESIS appears to
## properly with the ncurses supplied with Red Hat Linux 5.1
```

```
## and Debian Linux (glibc2.1, egcs-2.91.66).  However, link
## ncurses is known to have resulted in core dumps in GENESI
## Linux versions.
##
## If you encounter problems linking with the TERMCAP flags
## or the GENESIS command line interface does not work, try
## following alternatives:
##
## 1) TERMCAP = -ltermcap
##
## 2) (If you are using SuSE Linux)
##    TERMCAP = /usr/lib/termcap/libtermcap.a
##
## 3) (If you are using Red Hat Linux prior to version 6.0)
##    TERMCAP = /usr/lib/libtermcap.a
##

MACHINE=Linux
OS=BSD

XINCLUDE=-I/usr/X11R6/include

## 64-bit machines probably need /usr/X11R6/lib64 here.
XLIB=/usr/X11R6/lib

CC=mpicc

## Old (and probably broken) gcc installations may need the
## path to cpp (preferably NOT one in /lib). If there isn't
## [link to] cpp in the same directory as 'cc', you should c
## [re]installing a newer gcc.
CPP=cpp -P
CFLAGS=-O2 -D__NO_MATH_INLINES

## For 64-bit architectures
CFLAGS=-O2 -D__NO_MATH_INLINES -DLONGWORDS

LD=ld

## !!!
## Don't uncomment the next line unless you get errors about
## libraries not being found. Setting this path may interfer
```

```
## the default (probably correct) operation of the loader, b
## 64-bit architectures may need /usr/lib64 here.
## LDFLAGS=-L/usr/lib

RANLIB=ranlib
AR=ar
CPLIB=cp

YACC=bison -y
LEX=flex -l
LEXLIB=-lfl
LIBS= $(LEXLIB) -lm

TERMCAP=-lncurses
TERMOPT=-DTERMIO -DDONT_USE_SIGIO

## end Linux 1.2.x and up on Intel x86-based systems
```

Najważniejszą czynnością, o której często zapominamy i której nie było w wersji szeregowej GENESIS jest usunięcie komentarza z linii **1353**:

```
# when the Makefile has been configured, uncomment this vari
 EDITED = yes  // to jest linia 1353!
```

informującego program **make**, że dokonano edycji pliku **Makefile**.

8.5. Kompilacja i instalacja

Tak przygotowany plik **Makefile** pozwala rozpocząć proces kompilacji i instalacji PGENESIS.

```
make clean
make install
```

Proces ten trwa znacznie krócej niż kompilacja właściwego GENESIS, nie kończy się jakimś specjalnie radosnym komunikatem.

Jeżeli wszystko przebiegło pomyślnie to w katalogu **pgenesis/bin** powinien pojawić się wykonywalny plik **pgenesis**.

W tym miejscu warto wspomnieć, że gdyby zaszła konieczność instalacji PGENESIS w wersji bez okienek – należy najpierw skompilować do takiej wersji używane dotychczas GENESIS. Instalacja PGENESIS bez okienek dokonuje się po wykonaniu polecenia:

```
make nxinstall
```

Instalację GENESIS bez XODUS opisano wcześniej.

Co więcej, autor zaleca kompilację PGENESIS w wersji bez obsługi X-Windows ze względu na częste problemy pojawiające się przy okazji współpracy przestarzałego interfejsu XODUS z najnowszymi bibliotekami GTK. Poza tym obliczenia równoległe wykonuje się najczęściej na klastrach obliczeniowych, na których ze względów bezpieczeństwa i tak środowisko graficzne nie jest zainstalowane.

Wszystkie przykładowe skrypty zawierające równoleglizację będą tworzone i testowane dla PGENESIS w odmianie bez okienek.

Na tym etapie najważniejszą część instalacji można uznać za zakończoną.

8.6. Czynności poinstalacyjne

Po zainstalowaniu PGENESIS należy skopiować plik **.psimrc** (albo **.nxpsimrc**) z katalogu **pgenesis/startup** do katalogu głównego:

```
cd startup
cp .psimrc ~  // albo cp .nxpsimrc ~
```

Oczywiście do ścieżki dostępów w pliku **.bashrc** dodajemy frazę:

```
:/home/student/genesis-2.3/pgenesis/bin
```

żeby z każdego miejsca w systemie można było uruchamiać wersję równoległą.

8.7. Testowanie instalacji

Po kompletnym przygotowaniu i kompilacji środowiska można sprawdzić z dowolnego miejsca w konsoli systemu czy PGENESIS skompilował się poprawnie. Należy więc wykonać komendę:

```
pgenesis
```

i jeśli wszystko poszło dobrze ba ekranie otrzymujemy informację:

```
performing checks...
starting pgenesis executable
num_nodes: Undefined variable.
```

Należy pamiętać, że po dodaniu PGENESIS do ścieżki dostępu powinniśmy zamknąć konsolę Linuksa i otworzyć ją raz jeszcze. To najwygodniejszy sposób przeładowania ścieżki, zapewniający **pgenesis** widoczność z dowolnego miejsca w systemie.

8.8. Podsumowanie

Przedstawiono krok po kroku metodologię kompilacji i instalacji biblioteki MPICH2 oraz równoległej wersji symulatora GENESIS. Czytelnikom zaleca się przygotowanie PGENESIS w wersji bez środowiska X-WINDOWS.

8.9. Zadania

Zadanie 1

Skompiluj ze źródeł i zainstaluj najnowszą wersję biblioteki MPICH2.

Zadanie 2

Skompiluj ze źródeł i zainstaluj PGENESIS w wersji standardowej.

Zadanie 3

Skompiluj i zainstaluj ze źródeł PGENESIS w wersji bez XODUS. Pamiętaj o wcześniejszej rekompilacji GENESIS do wersji bez wsparcia systemu X. Możesz stworzyć sobie dodatkowe katalogi na wersje NX obu symulatorów, pamiętaj jednak o konieczności dodawania odpowiednich ścieżek w pliku **.bashrc**.

Rozdział 9

Symulacje równoległe w PGENESIS

9.1. Wprowadzenie

W tym rozdziale nauczymy się rozdzielać zadania symulatora na poszczególne procesory albo rdzenie procesorów maszyny równoległej. Zaczniemy od najprostszego modelu zawierającego dwa neurony, skończymy na modelu sieci zawierającej 128 neuronów symulowanych na dwóch procesorach. Okazuje się, że w PGENESIS występuje zaledwie kilka sztuczek służących rozplataniu algorytmów na wiele wątków. Sposób równoleglizacji ograniczony jest tylko wyobraźnią projektanta modelu.

9.2. Dwa neurony

Każdy skrypt, który ma być rozwidlony na kilka współbieżnych procesów powinien rozpoczynać się od linii:

```
paron -parallel -nodes 3
```

W przytoczonym tu przypadku rozpraszamy symulację na trzy nody. Warto zapamiętać, że w równoległym PGENESIS powinniśmy dbać o pozostawienie jednego dodatkowego nodu na wszelke czynności związane z zarządzaniem symulacją.

PGENESIS oferuje możliwość tworzenia elementów i uruchamiania poleceń na wskazanych nodach symulacji. Służy do tego operator @.

I tak na przykład polecenie:

```
readcell@1 cell.p /CellA
```

spowoduje utworzenie komórki **CellA** przy pomocy mechanizmu **cell reader** ale tylko na nodzie pierwszym uruchomionej symulacji.

Podobnie polecenie:

```
le@1
```

wyświetli nam elementy rezydujące na nodzie pierwszym.

Warto mieć na uwadze, że polecenia wydane w PGENESIS bez operatora @ wykonują się na wszystkich nodach.

PGENESIS inicjuje również domyślną zmienną **mynode**, w której przechowywany jest numer nodu na każdym nodzie symulacji z osobna.

Jeżeli chcemy więc wykonać jakieś polecenie, ale tylko w sytuacji gdy znajdujemy się na określonym nodzie - możemy skorzystać z instrukcji warunkowej:

```
if ({mynode}==1)
create@1 randomspike /input
setfield@1 ^ min_amp 1 max_amp 1 rate 200 reset 1 reset_value 0
end
```

W tym wypadku stworzono generator losowych impulsów wejściowych na pierwszym nodzie symulacji.

Polecenie

```
barrier
```

ustawia barierę na wszystkich nodach i powoduje, że wszystkie one czekają na siebie wzajemnie aż osiągną określoną linię w realizacji zadań zdefiniowanych w skrypcie.

Prosty skrypt, w którym tworzymy dwie komórki **CellA** i **CellB** reazydujące odpowiednio na nodach 1 i 2 oraz generator losowych impulsów na pierwszym nodzie przyjmuje następującą postać:

```
paron -parallel -nodes 3
int NodesNumber = 3
setclock 0 0.0001

include functions.g
include pfunctions.g
include protodefs.g

if ({mynode}==1)
readcell@1 cell.p /CellA
end

if ({mynode}==2)
readcell@2 cell.p /CellB
end

if ({mynode}==1)
create@1 randomspike /input
setfield@1 ^ min_amp 1 max_amp 1 rate 200 reset 1 \
             reset_value 0
end

barrier

if ({mynode}==1)
 make_synapse /input /CellA/dend/Ex_channel 2 0
 pmake_synapse /CellA/soma/spike 1 \
               /CellB/dend/Ex_channel 2 15 0
end
```

```
barrier

if ( {mynode}==1 )
 create spikehistory net-history
 setfield ^ filename CellA-at1.spike append 0 ident_toggle 1
 addmsg /CellA/soma/spike net-history SPIKESAVE
end

barrier

if ( {mynode}==2 )
 create spikehistory net-history
 setfield ^ filename CellB-at2.spike append 0 ident_toggle 1
 addmsg /CellB/soma/spike net-history SPIKESAVE
end

barrier

reset
step 0.1 -time
paroff
quit
```

Zauważmy, że kiedy jesteśmy na pierwszym nodzie – tworzymy synapsę z generatora losowych impulsów **input** do komórki **CellA**. Przy pomocy funkcji **pmake_synapse** tworzymy połączenie z komórki **CellA** do **CellB**. Zapis wyników na dysk realizujemy na dwóch nodach, dla każdej komórki z osobna.

Na początku zkryptu dołączamy doń plik **pfunctions.g** zawierający funckje równoległe:

```
function pmake_synapse(pre,npre,post,npost,weight,delay)
str pre,post
float weight,delay
int syn_num,npre,npost

raddmsg {pre} {post}@{npost} SPIKE
syn_num = {getfield@{npost} {post} nsynapses} - 1
setfield@{npost} {post} synapse[{syn_num}].weight {weight} \
                  synapse[{syn_num}].delay {delay}
    echo {pre} ==>> {post} {weight} {delay}
end
```

Zauważmy, że do przesyłania wiadomości między nodami służy polecenie:

```
raddmsg
```

Pozostała część funkcji jest analogiczna do klasycznej funkcji tworzącej synapsę między dwoma neuronami. Pilik **functions.g** zawarty na początku skryptu jest budowaną przez nas od pierwszych rozdziałów nieniejszej książki biblioteką funkcji własnych użytkownika GENESIS.

Główny skrypt programu kończy się wyłączeniem równoleglizacji na wszystkich węzłach poleceniem:

```
paroff
```

Skrypty równoległe uruchamiamy poleceniem:

```
pgenesis -nodes 3 -nox nazwa_skryptu.g
```

pamiętając, że liczba nodów zadeklarowanych na początku uruchamianego skryptu powinna być taka sama jak liczba nodów podanych w parametrze **-nodes**.

Po wykonaniu obliczeń na dysku pojawią się dwa pliki z zapisem aktywności komórek: **CellA.spike** oraz **CellB.spike**.

Mamy namacalny dowód, że komórka stymulowana przez generator na nodzie pierwszym przekazała sygnały do komórki na nodzie drugim, przy czym oba nody niezależnie od siebie zapisywały aktywność przechowywanych na sobie komórek na dysk.

9.3. Przykładowa sieć rozległa

Teraz zrównoleglimy znany nam skądinąd model zawierający dwie kwadratowe (8×8) sieci neuronów w taki sposób, aby każda z nich rezydowała ma odrębnym nodzie w symulacji. Siatki **net1** i **net2** tworzymy wykonując polecenie **createmap** na nodach pierwszym i drugim, podobnie zresztą czynimy z zapisem aktywności. Generator impulsów stymuluje pierwszą sieć i rezyduje na pierwszym nodzie. Sieci połączone są w obrębie samych siebie z prawdopodobieństwem 0.1. Dodatkowo realizowane jest 10% połączenia pełnego między sieciami w kierunku od **net1** do **net2**. Model tworzony jest przy pomocy przedłożonego tu skryptu:

```
paron -nodes 3 -parallel

float sep_x=0.001  // odl. komorek w  kierunku x w [m]
float sep_y=0.001  // odl. komorek w  kierunku y w [m]

float genrate=200
```

```
setclock 0 0.000005
setclock 1 {1.0/{genrate}}  // for generator

include functions-wizualizacja.g
include pfunctions.g
include protodefs.g

readcell cell.p /cell

if ({mynode} == 1)
createmap /cell /net1 8 8 -delta {sep_x} {sep_y}
end

if ({mynode} == 2)
createmap /cell /net2 8 8 -delta {sep_x} {sep_y}
end

if ({mynode} == 1)
create randomspike /input
setfield ^ min_amp 1 max_amp 1 rate {genrate} reset 1 \
           reset_value 0 abs_refract 0
useclock /input 1

create spikehistory /input-spikes
setfield ^ filename input.spike append 0 ident_toggle 1
addmsg /input /input-spikes SPIKESAVE
end


if ({mynode} == 1)
connect_map /net1 8 8 0.1
end

if ({mynode} == 2)
connect_map /net2 8 8 0.1
end

if ({mynode} == 1)
pconnect_two_maps /net1 1 /net2 2 8 8 0.1
make_synapse /input /net1/cell[0]/dend/Ex_channel 10 0 0
end
```

```
if ({mynode} == 1)
make_mapoutput /net1/cell net1.spike
end

if ({mynode} == 2)
make_mapoutput /net2/cell net2.spike
end

check
reset

step 1 -time

paroff
quit
```

przy czym do pliku zawierającego bibliotekę funkcji równoległych użytkownika – **pfunctions.g** dopisano funkcję łączącą dwie mapy:

```
function //uwaga: ponizsze argumenty w jednej linii
   pconnect_two_maps (net1, npre, net2, npost, nx, ny, prob)
int i,j, netdim

netdim  = {nx} * {ny}

for (i=1;i<netdim;i=i+1)
 for (j=1;j<netdim;j=j+1)

   if ({rand 0 1} < {prob} )
    pmake_synapse {net1}/cell[{i}]/soma/spike {npre} \
                  {net2}/cell[{j}]/dend/Ex_channel {npost} \
                  1.8 1e-04
   end
end
end
```

Po wykonaniu skryptu na dysku powstają dwa pliki: **net1.spike** oraz **net2.spike** z zapisem aktywności obu sieci symulowanych równolegle oraz plik **input.spike** z historią pobudzeń generatora.

9.4. Kontrola symulacji

W tym miejscu warto opisać kilka prostych sztuczek, które mogą ułatwić programowanie równoległych modeli w PGENESIS oraz zarządzanie symulacją. Tworząc równoległe modele należy pamiętać, że:

— struktury modelu, które charakteryzują się odrębnością i dużą gęstością połączeń (na przykład mikroobwody neuronalne) warto umieszczać na tych samych nodach,

— dużo czasu procesorów marnujemy przesyłając wiadomości między nodami dlatego warto określić miejsca w modelu, w których komunikacja będzie ograniczona (np. rzadkie połączenia między mikroobwodami) i to dla nich rezerwować międzynodową wymianę informacji,

— optymalnym rozwiązaniem jest rozproszenie symulacji na tyle nodów iloma procesorami (rdzeniami) dysponujemy. Oczywiście należy zachować umiar i nie ma sensu rozpraszanie sieci zawierającej sześć prostych komórek na sześć procesorów.

Kiedy uruchomimy już symulację warto sprawdzić obciążenie poszczególnych procesorów (lub rdzeni) maszyny. Przydatnym, aczkolwiek niezainstalowanym od samego początku w Ubuntu programem jest **htop**. Instalujemy go w najprostszy z możliwych sposobów:

```
sudo apt-get install htop
```

i uruchamiamy poleceniem:

```
htop
```

Uruchomiony **htop** wygląda jak na Rys. 9.1.

Czasami zdarzają się sytuacje, w których chcemy sprawdzić jak długo wykonywał się dany skrypt. Warto posłużyć się napisanym własnoręcznie w powłoce **bash** skryptem **start.sh** o następującej treści:

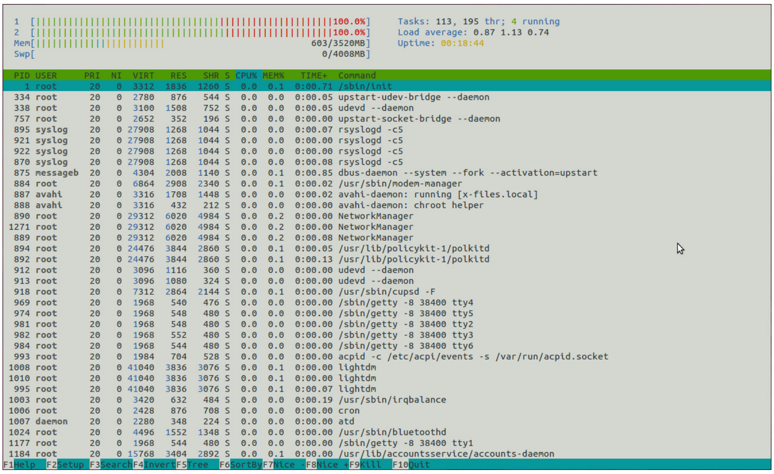
```
#!/bin/bash
date > time.out
pgenesis -nodes 3 -nox nazwa_skryptu.g
date >> time.out
```

Każdemu plikowi w systemie Linux można nadać atrybut wykonywalności, w tym wypadku wykonując polecenie:

```
chmod +x start.sh
```

Chcąc uruchomić symulację wykonujemy skrypt (pamiętamy, że musi on rezydować w katalogu z modelem):

```
./start.sh
```



Rysunek 9.1. Równomierne obciążenie obu rdzeni procesora podczas wykonywania symulacji równoległej

i czekamy na zakończenie.

W wyniku działania skryptu na dysku powstaje dodatkowy plik **time.out** o przykładowej treści:

```
Mon Aug 23 18:29:01 CEST 2010
Mon Aug 23 19:56:54 CEST 2010
```

z którego łatwo można wyciągnąć informację o czasie wykonania symulacji z dokładnością do jednej sekundy.

9.5. Podsumowanie

Przedstawiono metodologię równoleglizacji skryptów GENESIS do pracy w środowisku PGENESIS. Należy zapamiętać, że semantycznie do równoleglizacji służy zaledwie kilka poleceń, metody optymalnego zrównoleglania modeli zależą zaś od wyobraźni programisty. Potrafimy tworzyć elementy i uruchamiać polecenia PGENESIS na wskazanych nodach maszyny równoległej. Potrafimy też przesyłać informacje między nodami.

9.6. Zadania

Zadanie 1

Stwórz sieć zawierającą trzy komórki: **CellA**, **CellB** oraz **CellC**. Umieść poszczególne komórki na różnych nodach w symulacji. Połącz komórkę **Cel-**

lA z **CellB** oraz **CellB** z **CellC**. Zaplanuj zapis wyników na dysk wykonywany przez poszczególne nody. Uruchom symulację na czterech nodach. Przy pomocy polecenia **le** wyświetl elementy na poszczególnych nodach.

Zadanie 2

Zmodyfikuj model przedstawiony w tym rozdziale w taki sposób, aby tworzył prostokątne siatki neuronów. Wykonaj symulacje i zmierz czas ich trwania.

Zadanie 3

Sprawdź czas obliczeń symulacji modelu z **Zadania 2** dla kilku różnych wartości prawdopodobieństwa utworzenia synapsy między neuronami poszczególnych sieci.

Rozdział 10

Dostosowywanie GENESIS

10.1. Wprowadzenie

W tym rozdziale nauczymy się ingerować bezpośrednio w kod źródłowy GENESIS. Nie jest to sprawa banalna i wymaga trochę doświadczenia. Jednak głębsza analiza problemu modyfikacji symulatora pozwala nam stworzyć zupełnie nowe możliwości, a jedynym ograniczeniem będzie tu wyobraźnia. Po uważnej lekturze niniejszego rozdziału każdy czytelnik będzie mógł poczuć dumę GENESIS-owego hakera.

10.2. Prawdopodobieństwo egzocytozy

Mimo, że w dotychczasowych modelach wykorzystywaliśmy generator liczb pseudolosowych na przykład do generowania impulsów wejściowych to działanie modeli przez nas projektowanych było z grubsza rzecz biorąc wysoce powtarzalne. W naturze o taką powtarzalność trudno, żywy organizm nie jest automatem działającym w sposób tak precyzyjny jak werk mechanicznego zegarka.

Wiele ciekawych procesów rozgrywa się na poziomie synaptycznym. Sygnał z komórki A przekazywany jest do komórki B w wyniku zajścia tak zwanej egzocytocy. Upraszczając nieco to niezwykle skomplikowane zjawisko można powiedzieć, że egzocytoza to przekazanie pewnego pęcherzyka (neuroprzekaźnika) z jednej kolbki synaptycznej do drugiej. Okazuje się, że mimo sprzyjających warunków – egzocytoza nie za każdym razem ma miejsce. Zdefiniujmy więc prawdopodobieństwo zajścia egzocytocy jako szansę na przekazanie sygnału między neuronami nawet wtedy, gdy wszelkie warunki (w tym wypadku waga połączenia synaptycznego) temu sprzyjają.

Do tej pory każdą synapsę określały dwa parametry: waga – **weight** oraz opóźnienie – **delay**. Zmodyfikujemy kod GENESIS w taki sposób, by do funkcji tworzącej połączenie synaptyczne trzeba było dodać parametr określający prawdopodobieństwo – **probability**.

10.3. Edycja plików źródłowych

W tym celu należy zmienić nieco dwa pliki w kodzie źródłowym GENESIS. Na początku musimy dostać się do katalogu **genesis/src** i wykonać polecenie:

```
make clean
```

Następnie w katalogu **genesis/src/newconn** odnajdujemy plik **newconn_struct.h** i dopisujemy do linii **146** jedną linijkę kodu (nie zapomina-

jąc o przełamaniu jej popularnym backslashem) w taki sposób, by interesujący nas fragment pliku (linie od 143 do 147) wyglądały następująco:

```
#define SYNAPSE_TYPE     \
    MsgIn* mi;      \
    float weight;  \
    float delay;  \
    float       probability; // by gmwojcik 31.08.2011
```

Warto dodać komentarz z własnym pseudonimem, żeby później było wiadomo co i gdzie dopisaliśmy.

Następnie odnajdujemy plik **synchan.c** znajdujący się w tym samym katalogu i rozpoczynamy taniec z prawdopodobieństwem (linia **343**) w taki sposób, by linie od 343 do 354 wyglądały jak tu:

```
if ( frandom(0,1.0) <= channel->synapse[syn].probability )
   channel->activation += channel->synapse[syn].weight / dt;
    } //modified by gmwojcik 31.08.2011

  if ( next_event != NULL) {
    if (nothsolve) {
      --next_event->time;
    } else {
      channel->time_last_event=SimulationTime();
    }
  }
}
```

Pamiętamy o dopisaniu odpowiedniego komentarza do dopisanej linijki. W gruncie rzeczy dopisaliśmy tylko jedną linię kodu w języku c++ – wywołanie odpowiedniej, zależnej od pola **probability** instrukcji warunkowej.

Po zapisaniu obu plików przechodzimy z powrotem do katalogu **genesis/src** i dokonujemy ponownej kompilacji i instalacji środowiska:

```
make nxall
make nxinstall
```

w tym wypadku bez XODUS. Pamiętajmy, że gdybyśmy chcieli korzystać z GENESIS w wersji równoległej – należy również przekompilować i zainstalować ponownie PGENESIS. Teraz jednak będzie nam trochę łatwiej gdyż odpadają nudne i żmudne czynności związane z dopisywaniem ścieżek dostępu do **.bashrc** i kopiowaniem plików typu **.simrc** do katalogu domowego.

10.4. Konsekwencje

W konsekwencji otrzymaliśmy zupełnie nowy, taki tylko nasz GENESIS z nowym typem synapsy uwzględniającym prawdopodobieństwo zajścia egzocytozy. W podobny sposób można definiować sobie inne typy synaps.

Niestety skrypty napisane do tej pory nie będą już działać w takim symulatorze bez drobnej modyfikacji. (Dlatego warto rozważyć kompilację zmodyfikowanego GENESIS w jakimś innym katalogu). Od tej pory funkcja tworząca synapsę musi być wzbogacona o trzecie pole – **probability**:

```
function make_synapse(pre,post,weight,delay, prob)
 str pre,post
 float weight,delay
 int syn_num

   addmsg {pre} {post} SPIKE
   syn_num = {getfield {post} nsynapses} - 1
   setfield {post} synapse[{syn_num}].weight {weight} \
                   synapse[{syn_num}].delay {delay} \
                   synapse[{syn_num}].probability {prob}
   echo {pre} "--->" {post} {weight} {delay} {prob}

end
```

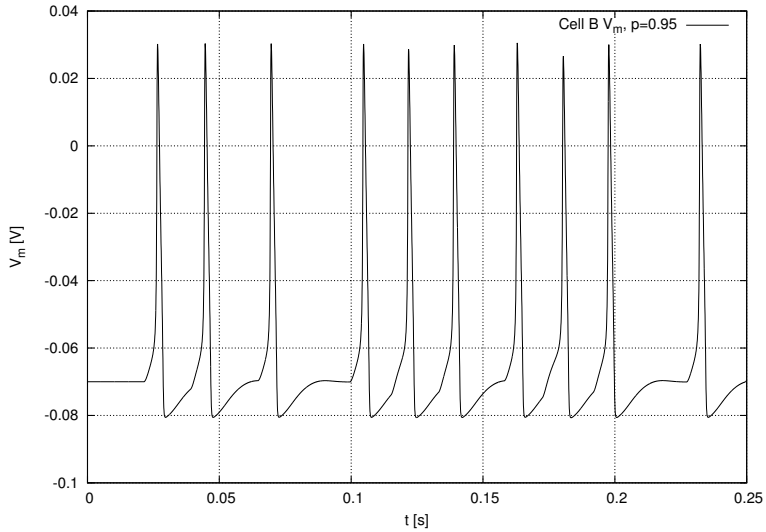
i zauważmy, że chociaż nie musimy tego robić – wyświetlamy wartość prawdopodobieństwa obok wagi i opóźnienia.

Wywołania takiej funkcji gdzieś z poziomu skryptu w modelu mają postać:

```
make_synapse /cell_A/soma/spike \
             /cell_B/dend/Ex_channel 2.8 1e-04 0.95
make_synapse /cell_A/soma/spike \
             /cell_B/dend/Ex_channel 2.8 1e-04 0.25
```

dla prawdopodobieństw egzocytozy ustawionych odpowiednio na $p = 0,95$ i $p = 0,25$

Na potrzeby niniejszego rozdziału stworzono model zawierający dwie komórki połączone ze sobą synapsą o takich właśnie prawdopodobieństwach egzocytozy. Symulację uruchomiono dwa razy (dla każdej wartości po jednym razie). Przebieg potencjału na drugiej komórce przedstawiają Rys. 10.1 i 10.2. Co oczywiste – komórka zasilana synapsą o mniejszym prawdopodobieństwie p „pika" rzadziej, jednak przy odpowiednio długim czasie symulacji biologicznej aktywności systemu – za każdym razem inaczej.



Rysunek 10.1. Przebieg potencjału błony komórkowej neuronu pobudzanego synapsą o prawdopodobieństwie zajścia egzocytozy $p = 0,95$

10.5. Podsumowanie

Przedstawiono krok po kroku proces ingerencji w kod źródłowy GENESIS prowadzący do zmiany charakterystyki typowej synapsy. Uzależnienie przekazywania sygnału od wprowadzenia prawdopodobieństwa egzocytozy ma istotne znaczenie wszędzie tam gdzie chcemy badać statystykę zachowania modelu zwierającego dużą liczbę symulowanych neuronów. W podobny sposób można pomyśleć o innych typach synaps i zaimplementować je we wskazane miejsca w kodzie.

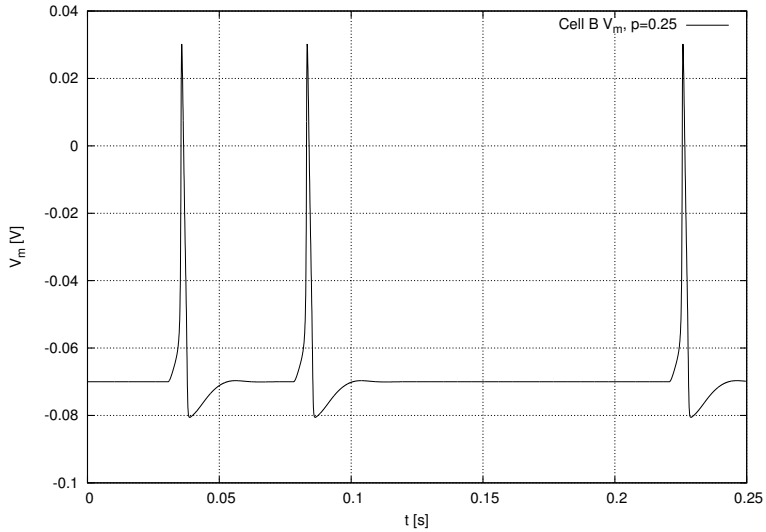
10.6. Zadania

Zadanie 1

Skompiluj zmodyfikowaną wersję GENESIS zawierającą prawdopodobieństwo zajścia egzocytozy wbudowane w synapsę. Możesz zbudować nową wersję GENESIS w innym katalogu niż oryginał, pamiętaj jednak wtedy o zmianie ścieżki w **.bashrc** i podmianie plików typu **.simrc** w katalogu domowym.

Zadanie 2

Przeprowadź kilka symulacji wybranego spośród omawianych w tej książ-



Rysunek 10.2. Przebieg potencjału błony komórkowej neuronu pobudzanego synapsą o prawdopodobieństwie zajścia egzocytozy $p = 0,25$

ce modelu z uwzględnieniem prawdopodobieństwa zajścia egzocytozy. Porównaj rozmiary plików z zapisem aktywności komórek dla różnych wartości prawdopodobieństwa.

Rozdział 11

Uwagi końcowe

W książce przedstawiono kurs modelowania w symulatorze GENESIS, zarówno w jego wersji klasycznej jak i w równoległej.

Czytelnik miał okazję poznać podstawy modelowania komórek biologicznych ze szczególnym uwzględnieniem modelu Hodgkina–Huxleya.

W przypadku GENESIS praktykę modelowania rozpoczyna się od tworzenia modeli pojedynczych komórek, a następnie ich sieci. Duże sieci neuronów biologicznych można programować w taki sposób, by symulacja mogła odbywać się na klastrach obliczeniowych.

Przedstawiono metody wizualizacji aktywności sieci z wykorzystaniem wbudowanych w GENESIS narzędzi, a także programów zewnętrznych oferujących sporo możliwości w generowaniu reprezentacji dynamiki pojedynczych komórek.

Wskazano też możliwe kierunki modyfikowania oprogramowania w celu uzyskania nowych funkcjonalności.

Książka pozwala w sposób łagodny wejść w problematykę modelowania w GENESIS, autor wyraża nadzieję iż stanie się ona przydatnym podręcznikiem dla wszystkich tych, którzy pragną rozpocząć badania w zakresie neuronauki obliczeniowej.

Dodatek A

Ważniejsze modele autora

Opis modeli wykorzystywanych w prowadzonych przez autora eksperymentach zaczerpnięto z monografii pod tytułem „Obliczenia płynowe w modelowaniu mózgu" [20]. Zainteresowani czytelnicy mogą znaleźć tam również szczegółowo opisane wyniki badań prowadzonych przez autora na przedstawianych tu modelach.

A.1. Model kory baryłkowej szczura

Model kory baryłkowej szczura zbudowano w środowisku GENESIS. Na siatce 45×45 skonstruowano sieć zawierającą 2045 neuronów, nazywaną od tej pory modelem 2k [20].

Para liczb naturalnych z przedziału od 0 do 44 w sposób jednoznaczny identyfikowała każdy neuron. Wszystkie komórki podzielono na 22 sekcje zwane warstwami numerowanymi od 1 do 22 (rys. A.1). Komunikację między neuronami ustalono według określonych zasad. Sygnał z każdej komórki z warstwy m był transmitowany na zewnątrz do komórek z warstw $m+1, m+2, \ldots, m+N_s$, gdzie N_s, zwane liczbą sąsiedztw, było zawsze nie większe niż liczba warstw. Struktura taka dość wiernie naśladowała gęste pierścienie dendrytyczne w baryłkach kory. Ponieważ symulacje były prowadzone z wykorzystaniem szesnastu procesorów, całą siatkę podzielono na 15 stref, zostawiając jeden procesor do zarządzania całą symulacją [20].

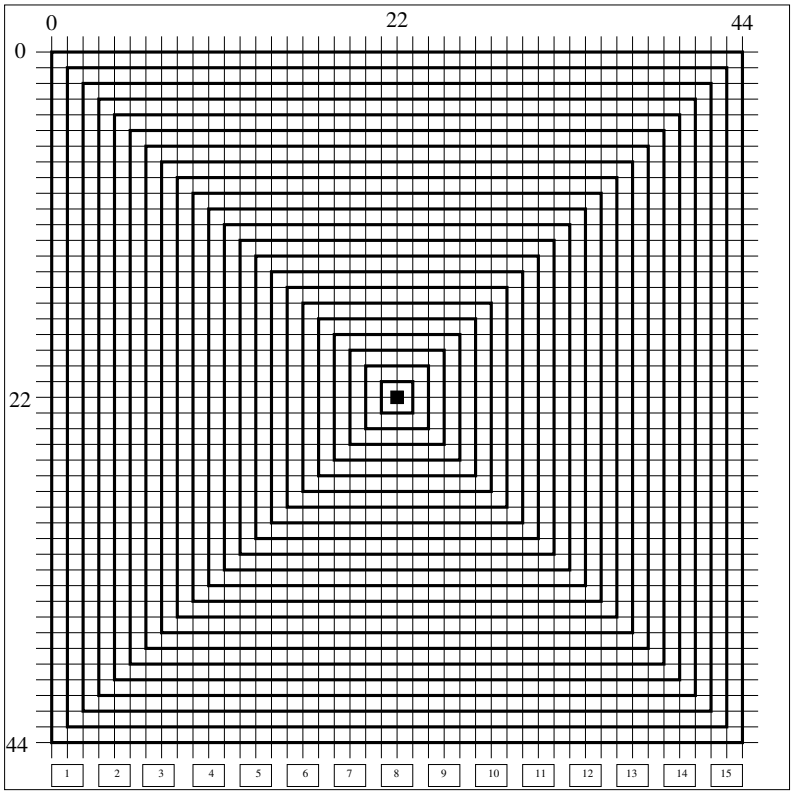
Połączenia synaptyczne w modelu charakteryzowane są przez dwa parametry zależne od odległości między neuronami: wagę $w = w_0/|m-n|$ oraz opóźnienie $\tau = |m-n|10^{-4}$ s. parametr w_0 we wszystkich symulacjach przyjmował wartość 2. Układ jest stymulowany za pośrednictwem generatora wysyłającego impulsy z częstotliwością f=80 Hz, podłączonego do centralnego neuronu $N_{23,23}$. Stymulacja ta przypomina pobudzanie szczurzego wibrysa za pomocą rurki kapilarnej lub elektrody wpiętej bezpośrednio w korę czuciowo-somatyczną [20].

Dodatkowo symulacja charakteryzowana jest przez parametr T oznaczający czas biologicznej pracy układu, u nas w większości przypadków $T = 15$ s [20].

Pobudzenie neuronu centralnego było transmitowane zgodnie z architekturą połączeń. Jako lawinę potencjału czynnościowego zdefiniowano liczbę neuronów w stanie wzbudzenia (wystrzeliwujących iglicę potencjału czynnościowego) w określonym interwale czasowym $t = 1$ ms [20].

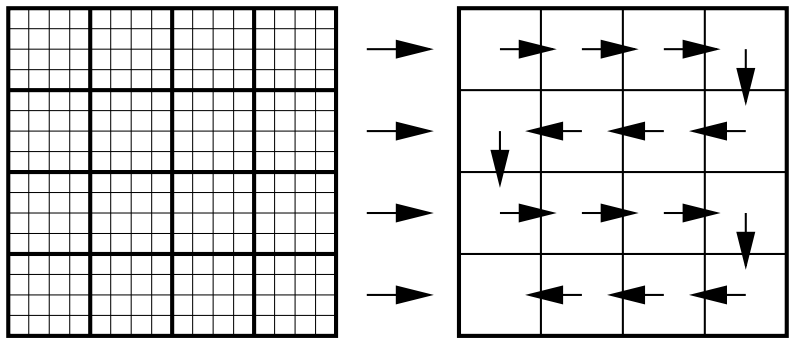
A.2. Model kory wzrokowej zawierającej maszyny LSM

Podstawowe serie eksperymentów polegających na badaniu zdolności separacji maszyn płynowych prowadzono na modelu układu wzrokowego za-



Rysunek A.1. Schemat modelu 2k. Warstwy oznaczono liniami pogrubionymi, neuron stymulujący czarnym kwadratem. Współrzędne neuronów zaznaczono na górze oraz po lewej stronie schematu. W każdej strefie (zaznaczonej na dole) znajdują się trzy kolumny neuronów [20]

wierającego około szesnastu tysięcy komórek HH i stąd nazywanego modelem 16k. W tym modelu (rys. A.2) siatkówka zbudowana jest z 256 neuronów rozmieszczonych na siatce 16×16 [20]. Siatkówka została podzielona na 16



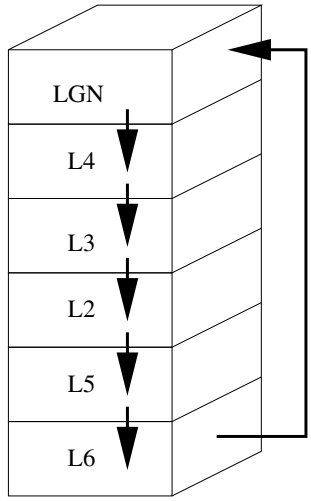
Rysunek A.2. Schemat modelu 16k. Zaznaczono przykładową strukturę połączeń między kolumnami [20]

pól recepcyjnych rozpiętych na siatce o wymiarach 4×4. Każde z pól recepcyjnych (również o wymiarach 4×4) jest połączone z jedną z szesnastu kolumn HHLSM (Hodgkin–Huxley Liquid State Machine) odpowiadających ciału kolankowatemu bocznemu i fragmentowi kory wzrokowej. W obrębie kolumn HHLSM zaaranżowano strukturę (rys. A.3) odpowiadającą rzeczywistym połączeniom międzywarstwowym w korze mózgu. Na rys. A.3 warto zwrócić uwagę na połączenie zwrotne wyprowadzone z warstwy L6 do LGN. Każda kolumna HHLSM jest niezależną maszyną płynową i zawiera 1024 komórki rozmieszczone na trójwymiarowej siatce $8 \times 8 \times 16$. Każda z warstw L2–L6 zawiera neurony położone na siatce $8 \times 8 \times 3$. Warstwa LGN to układ o wymiarach $8 \times 8 \times 1$ [20].

Neurony w obrębie warstwy mogą być połączone na zasadzie każdy z każdym (co niemal gwarantuje szybką synchronizację układu) lub z ustalonym prawdopodobieństwem. Warstwy A i B łączy się według reguły polegającej na tym, że ostatnia podwarstwa warstwy A jest połączona z pierwszą podwarstwą warstwy B. Podwarstwy mają wymiary $8 \times 8 \times 1$ [20].

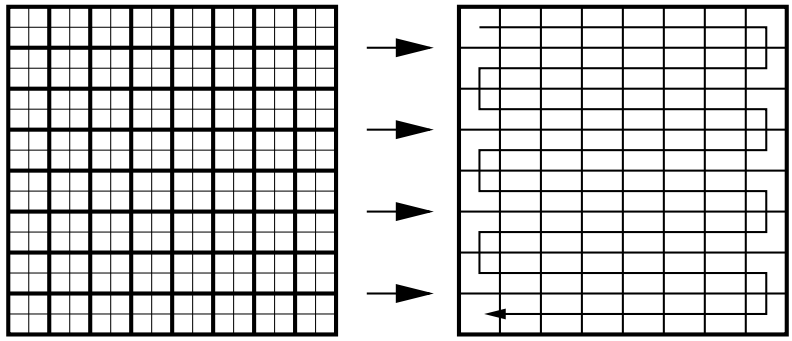
W zgodzie z wiedzą zaczerpniętą ze źródeł neurofizjologicznych oraz z teorią Maassa w obrębie każdej kolumny ustalono 80% połączeń wzbudzających i 20% połączeń hamujących. Neurony tworzące całą strukturę były stosunkowo prostymi komórkami typu HH zawierającymi ciało i dwa dendryty każda [20].

W zależności od konkretnego eksperymentu komputerowego kolumny HHLSM mogły być połączone ze sobą lub też nie, a twórca modelu przewidział możliwość dodawania połączeń między komórkami w szybki i prosty sposób [20].



Rysunek A.3. Schemat kolumny HHLSM z zaznaczoną strukturą połączeń [20]

Tak modelowany układ wzrokowy może zostać następnie poddany ekspertyzom rozmaitych warstw odczytujących. Co więcej, struktura modelu gwarantuje dobrą równoleglizację oraz bezproblemowe zwiększenie liczby symulowanych komórek do około 256 tysięcy (model 256k) przy dokonaniu podziału siatkówki na pola recepcyjne zawierające tylko jedną komórkę. Przykładowo na rys. A.4 przedstawiono schemat modelu 64k, w którym siatkówkę stanowią 64 pola recepcyjne o wymiarach 2×2, rozmieszczone na siatce 8×8 i połączone z 64 kolumnami HHSLM [20].



Rysunek A.4. Schemat modelu 64k. Zaznaczono przykładową strukturę połączeń między kolumnami [20]

Dodatek B

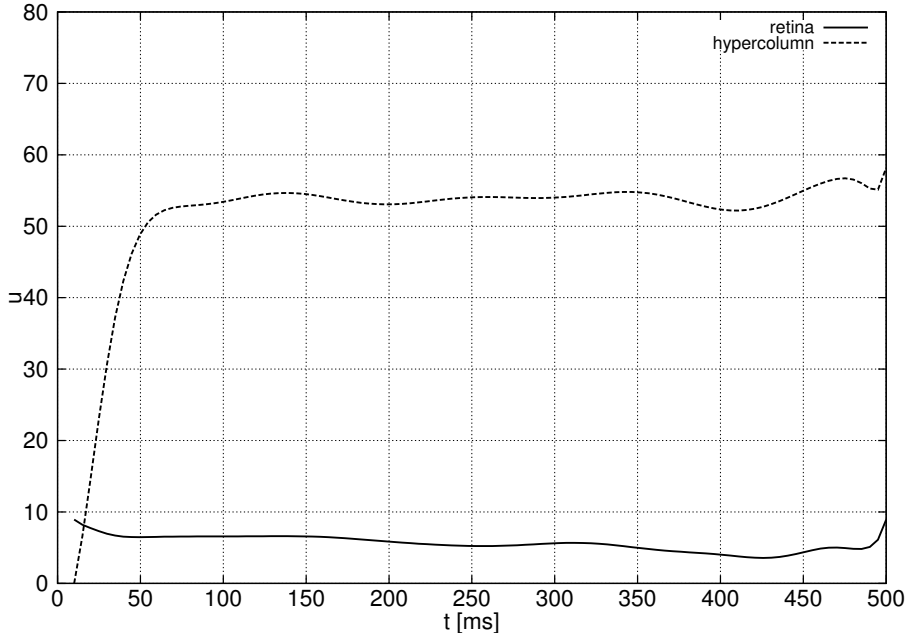
Rezultaty i publikacje autora

G. M. Wojcik and W. A. Kaminski, "Liquid computations and large simulations of the mammalian visual cortex," in Computational Science – ICCS 2006, vol. 3992 of Lecture Notes in Computer Science, pp. 94–101, Springer, 2006. 127

G. M. Wojcik and W. A. Kaminski, "Large scalable simulations of mammalian visual cortex," in Parallel Processing and Applied Mathematics, vol. 3911 of Lecture Notes in Computer Science, pp. 399– 405, Springer, 2005. 127

W tym dodatku przedstawiono wybrane wykresy pochodzące z badań opublikowanych przez autora w latach 2005–2012. W celu szczegółowego zapoznania się ze znaczeniem uzyskanych wyników odsyłamy do artykułów. W drugiej części dodatku przedstawiono angielskie streszczenia najważniejszych z nich.

B.1. Ważniejsze rezultaty

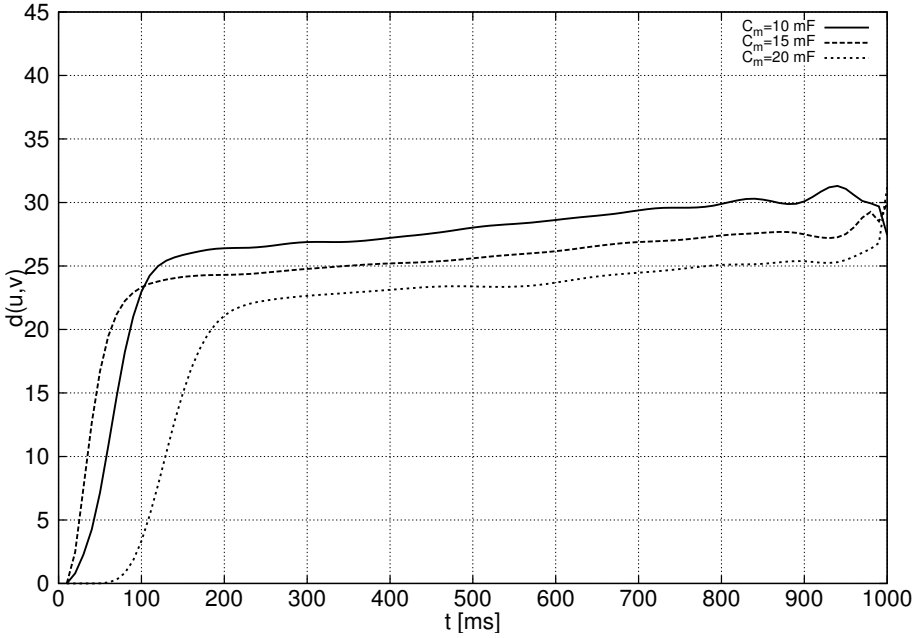


Rysunek B.1. Typowe wzmocnienie odległości stanów w maszynie LSM ma przykładzie zdolności separacji siatkówki oraz kolumny kory (za: [20])

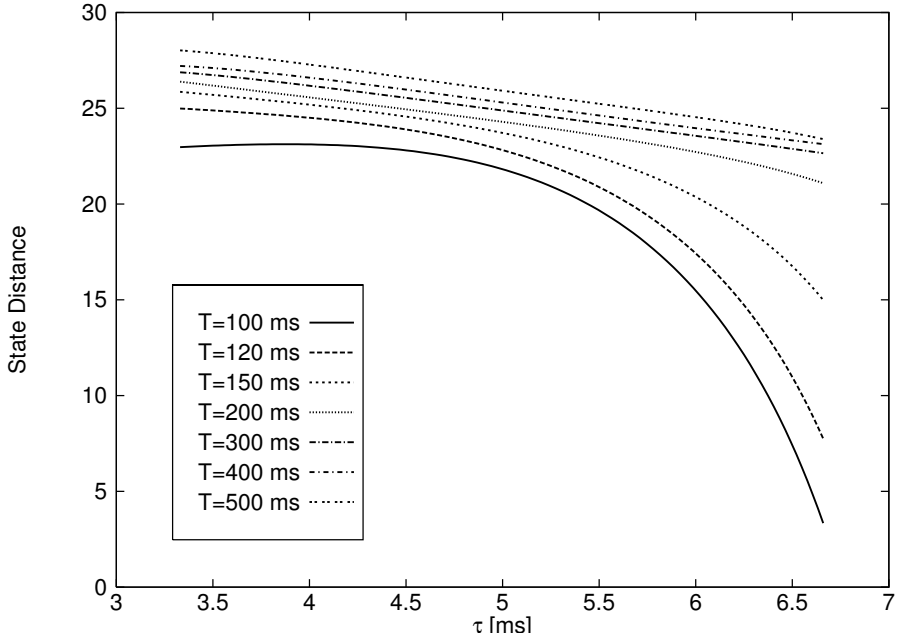
B.2. Streszczenia artykułów

G. M. Wojcik, "Self-organising criticality in the simulated models of the rat cortical microcircuits," Neurocomputing, no. 79, pp. 61–67, 2012.

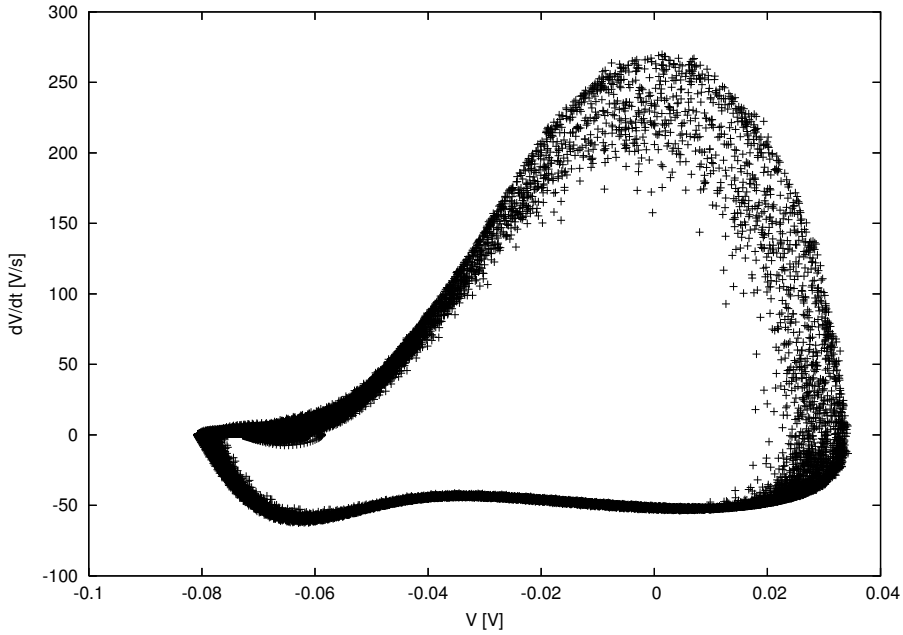
Two and three dimensional models of rat barrel and somatosensory cortex were simulated. Hoddgkin–Huxley and Leaky-Integrate-and-Fire neurons were used to the construction of the networks in GENESIS and PCSIM environments. The dynamics of both models was analysed. Self-organising



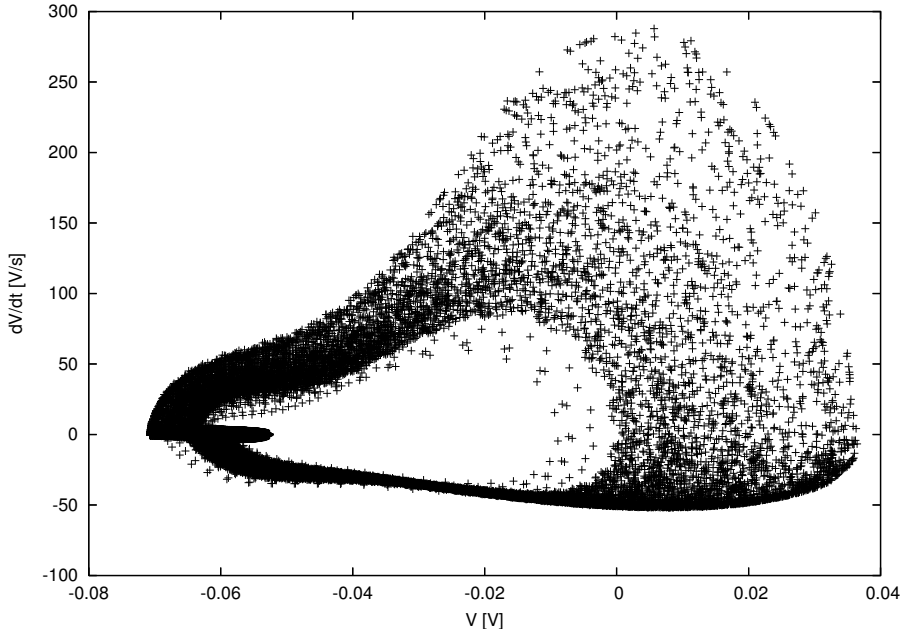
Rysunek B.2. Zdolność separacji maszyny LSM w zależności o pojemności błony komórkowej (za: [20])



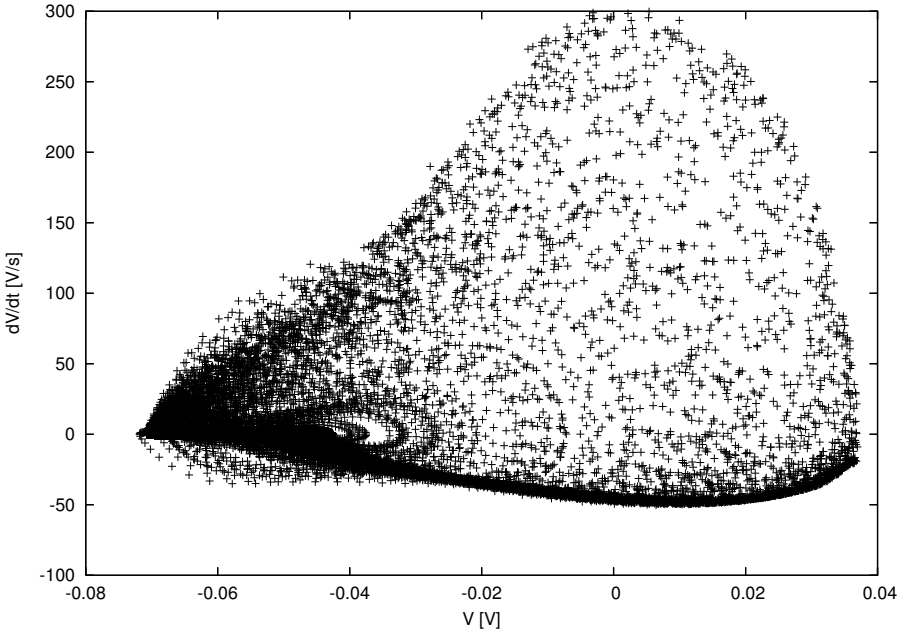
Rysunek B.3. Zdolność separacji maszyny LSM w zależności od stałej czasowej błony komórkowej (za: [20])



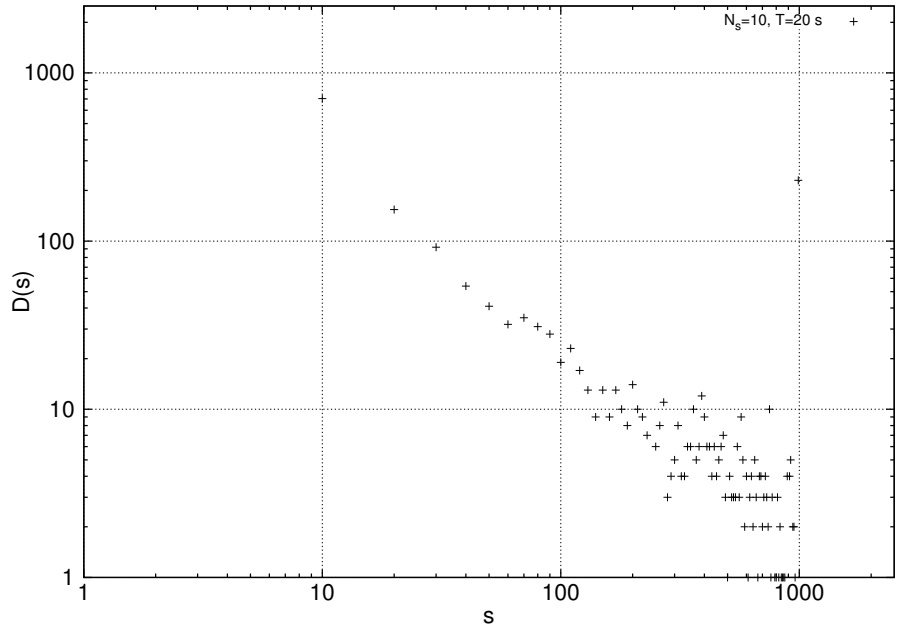
Rysunek B.4. Przykładowe odwzorowanie dynamiki neuronu w przestrzeni fazowej potencjału – przewidywalna trajektoria (za: [20])



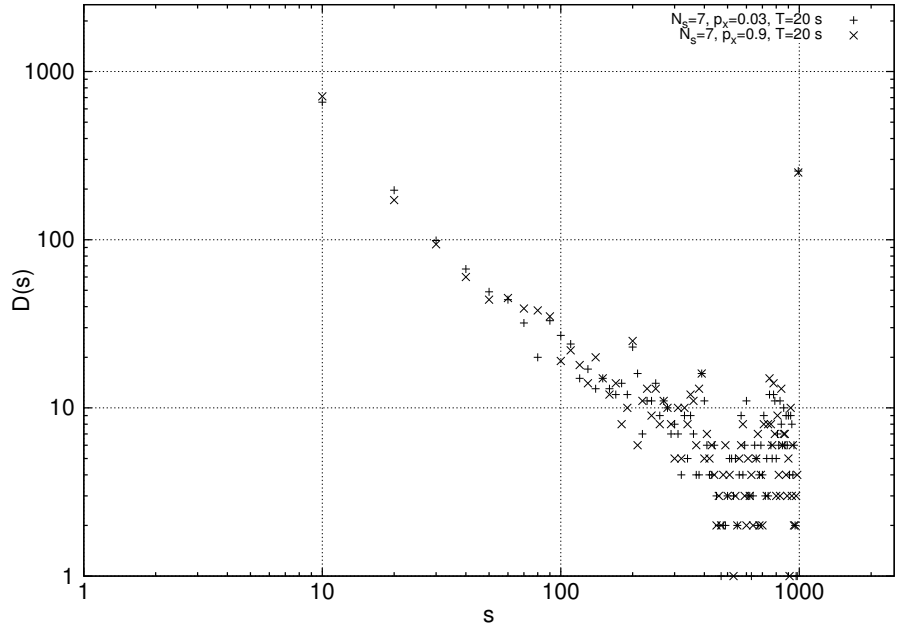
Rysunek B.5. Przykładowe odwzorowanie dynamiki neuronu w przestrzeni fazowej potencjału – trajektoria bardziej rozmyta niż na rys. B.4 (za: [20])



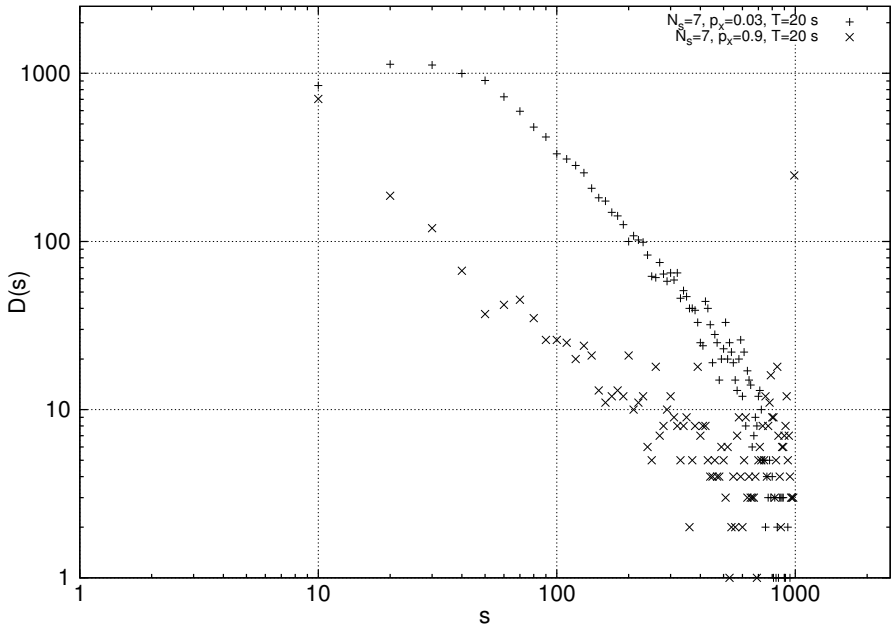
Rysunek B.6. Przykładowe odwzorowanie dynamiki neuronu w przestrzeni fazowej potencjału – trajektoria świadcząca o nieliniowej dynamice wybranego neuronu (za: [20])



Rysunek B.7. Typowy wykres przedstawiający występowanie zjawiska samoorganizującej się krytyczności w korze baryłkowej szczura (za: [20])



Rysunek B.8. Skala samoorganizującej się krytyczności w zależności od prawdopodobieństwa egzocytozy w sieci typu 2.5k (za: [20])



Rysunek B.9. Zjawiska SOC dla ustalonej liczby połączeń i zmieniającego się prawdopodobieństwa egzocytozy i zmiennej liczby połączeń między strefami modelu 2.5k (za: [20])

criticality phenomena were found. Profound investigations of this behaviour showed its dependence not only on the number of connections, but also on the simulated network architecture originating, e.g., from varying probability of exocytosis or synapse creation in the selected areas of the network. For 2-D model the results were compared to that obtained for smaller models and analysis of this comparison is presented to some extent. The three dimensional model of the rat primary somatosensory cortex based on the ensemble of Liquid State Machines. The results obtained from that model are in good agreement with the dynamics recorded in neurophysiological experiments on real brain [28].

G. M. Wojcik, "Electrical parameters influence on the dynamics of the hodgkin-huxley liquid state machine," Neurocomputing, no. 79, pp. 68– 78, 2012.

The model of mammalian cortical hypercolumn was simulated using liquid state machines built of simple Hodgkin–Huxley neurons. The influence of cell electrical parameters on the system dynamics was investigated. The systematic analysis of the hypercolumn separation ability in the function of time constants, cell membrane capacitance, resistance, Hodgkin–Huxley equilibrium potential and sodium and potassium channel conductances was performed. Optimal ranges of time constants for the most effective computational abilities of the model were estimated [29].

G. M. Wojcik and W. A. Kaminski, "Self-organised criticality as a function of connections' number in the model of the rat somatosensory cortex," in Computational Science – ICCS 2008, vol. 5101 of Lecture Notes in Computer Science, pp. 620–629, Springer, 2008.

The model of the part of the rat somatosensory cortex was examined. Large network of Hodgkin-Huxley neurons was simulated and the modular architecture of this structure divided into layers and sub-regions was implemented. High degree of complexity required effective parallelisation of the simulation. In this article the results of the parallel neural computations are presented. An occurrence of the self-organised criticality was observed and its characteristics as a function of the connections number was investigated. It was proved that frequency of the so-called spike potential avalanches depends on the density of inter-neuron connections. In addition some benchmarking runs were conducted and parallelisation effectiveness is presented to some extent [30].

G. M. Wojcik and W. A. Kaminski, "Nonlinear behaviour in mpi- parallelised model of the rat somatosensory cortex," Informatica, vol. 19, no. 3, pp. 461–470, 2008.

Mammalian brains consisting of up to 10^{11} neurons belong to group of the most complex systems in the Universe. For years they have been one of the hardest objects of simulation. There are many different approaches to modelling of neurons, but one of the most biologically correct is Hodgkin-Huxley (HH) model. Simulations that require solving a large number of nonlinear differential equations (fundamental in HH model) are always time and power consuming. The structures discussed in this article simulate a part of the rat somatosensory cortex. We use a modular architecture of the network divided into layers and sub-regions. Because of a high degree of complexity effective parallelisation of algorithms is required. We propose method of parallelisation for the network and the results of simulations using GENESIS parallelised for MPI environment are presented. An occurrence of nonlinear behaviour is demonstrated. Most notably, in large biological neural networks consisting of the HH neurons, nonlinearity is shown to manifest itself in the Poincaré sections generated for the varying value of neural membrane's potential [31].

G. M. Wojcik, W. A. Kaminski, and P. Matejanka, "Self-organised criticality in a model of the rat somatosensory cortex," in Parallel Computing Technologies, vol. 4671 of Lecture Notes in Computer Science, pp. 468–475, Springer, 2007.

Large Hodgkin-Huxley (HH) neural networks were examined and the structures discussed in this article simulated a part of the rat somatosensory cortex. We used a modular architecture of the network divided into layers and sub-regions. Because of a high degree of complexity effective parallelisation of algorithms was required. The results of parallel simulations were presented. An occurrence of the self-organised criticality (SOC) was demonstrated. Most notably, in large biological neural networks consisting of artificial HH neurons, the SOC was shown to manifest itself in the frequency of its appearance as a function of the size of spike potential avalanches generated within such nets. These two parameters followed the power law characteristic of other systems exhibiting the SOC behaviour [32].

G. M. Wojcik and W. A. Kaminski, "Liquid state machine and its separation ability as function of electrical parameters of cell," Neurocomputing, vol. 70, no. 13–15, pp. 2593–2697, 2007.

Large artificial Hodgkin-Huxley neural networks are examined. The structures discussed in this article simulate a part of the cortex of the mammalian visual system. We use a modular architecture of the cortex divided into sub-regions. Results of parallel simulations based on the Liquid Computing theory are presented in some detail. Separation ability of groups of neural microcircuits is observed. We check if such property depends on electrical parameters of particular cells. Properties Liquid Sate Machine's are derived from the types of neurons used for simulations. They are compared and discussed to some extent [33].

G. M. Wojcik and W. A. Kaminski, "Liquid computations and large simulations of the mammalian visual cortex," in Computational Science – ICCS 2006, vol. 3992 of Lecture Notes in Computer Science, pp. 94–101, Springer, 2006.

Large artificial Hodgkin-Huxley neural networks are examined. The structures discussed in this article simulate the cortex of the mammalian visual system. We use a modular architecture of the cortex divided into sub-regions. Results of parallel simulations based on the liquid computing theory are presented in some detail. Separation ability of groups of neural microcircuits is observed. We show that such property may be useful for explaining some edge or contrast detection phenomena [34].

G. M. Wojcik and W. A. Kaminski, "Large scalable simulations of mammalian visual cortex," in Parallel Processing and Applied Mathematics, vol. 3911 of Lecture Notes in Computer Science, pp. 399– 405, Springer, 2005.

Large artificial neural networks are examined. Structures discussed in this article simulate the cortex of mammalian visual system and its dynamics. Simulations of thousands of Hodgkin-Huxley neurons always require high computational power. Discussion of such networks parallelisation is presented in some detail. Analysis of simulation time, algorithm's speedup as a function of processors' number and density of connections is discussed as well [35].

Dodatek C

Zdjęcia pamiątkowe

Rysunek C.1. Klaster Clusterix i autor stojący obok klastra. Zbudowany z dwunastu dwuprocesorowych komputerów wyposażonych każdy w dwa procesory Itanium 2 1,4 GHz, 4 GB pamięci operacyjnej i 74 GB pamięci dyskowej. Maszyna stanowiła lubelski moduł Krajowego Klastra Linuksowego [36] wdrażanego w latach 2004–2005. Autor w ramach projektu Clusterix przeprowadzał symulacje wielkiej skali kory mózgowych ssaków wykorzytując pakiet GENESIS w wersji równoległej. (Fot. Stefan Ciechan)

Rysunek C.2. Autor skryptu z twórcą GENESIS Davidem Beemanem oraz jego żoną w sierpniu 2001 podczas szkoły EU Advanced Course in Computational Neuroscience w Obidos w Portugalii

Spis rysunków

Bibliografia

[1] R. Tadeusiewicz, *Theoretical Neurocybernetics.* Warsaw: Warsaw University Publishers, 2009.

[2] R. Tadeusiewicz, *Theoretical Neurocybernetics [In Polish: Neurocybernetyka Teoretyczna]*, ch. Models of the nervous system elements as artificial neural networks [In Polish: Modele elementów układu nerwowego w postaci sztucznych sieci neuronowych], pp. 109–127. Warsaw University Publishers, 2009.

[3] E. Mikołajewska and D. Mikołajewski, "Wybrane zastosowania modeli komputerowych w medycynie," *Annales Academiae Medicae Silesiensis*, vol. 1, no. 2, pp. 70–87, 2011.

[4] R. Tadeusiewicz, *Problems of Biocybernetics.* Warsaw: Polish Scientific Publishers (PWN), 1994.

[5] R. Tadeusiewicz, *Informatyka i psychologia w społeczeństwie informacyjnym*, ch. Neural Networks as Computational Tool with Interesting Psychological Applications [In Polish: Sieci neuronowe i inne systemy neurocybernetyczne jako narzędzia informatyczne o ciekawych zastosowaniach na gruncie psychologii], pp. 49–101. AGH Publishers, Kraków 2011.

[6] R. Tadeusiewicz, *The Industrial Electronics Handbook – Intelligent Systems by B.M. Wilamowski, J.D. Irvin (eds.)*, ch. Introduction to Intelligent Systems, pp. 1.1–1.12. CRC Press, Boca Raton, 2011.

[7] R. Tadeusiewicz, "New trends in neurocybernetics," *Computer Methods in Materials Science*, vol. 10, no. 1, pp. 1–7, 2010.

[8] R. Tadeusiewicz, *Lecture Notes in Artificial Intelligence 6114 by L. Rutkowski (et al., eds.)*, ch. Artificial Intelligence and Soft Computing, pp. 104–123. Springer-Verlag, Berlin – Heidelberg – New York, 2010.

[9] R. Tadeusiewicz, *On the paths of Neuroscience by Piotr Fracuz (ed.)*, ch. Modelling of the neural system elements with the neural networks use, pp. 13–34. KUL Publishers, 2010.

[10] R. Tadeusiewicz, "Using neural networks for simplified discovery of some psychological phenomena," in *Artificial Intelligence and Soft Computing* (e. L. Rutkowski et al., ed.), vol. 6114 of *Lecture Notes in Artificial Intelligence*, pp. 104–123, Springer-Verlag, Berlin – Heidelberg – New York, New York 2010.

[11] R. Tadeusiewicz, "Computers in psychology and psychology in computer science," in *Proceedings of the 2010 International Conference on Computer Information Systems and Industrial Management Applications (CISIM) With Applications to Ambient Intelligence and Ubiquitous Systems* (V. S. e. A. Abraham, K. Saeed, ed.), pp. 34–38, IEEE, 2010.

[12] R. Tadeusiewicz, *Prace Komisji Nauk Technicznych PAN, Tom III*, ch. Cybernetic modeling and computer simulation of biological systems [In Polish: Modelowanie cybernetyczne i symulacja komputerowa systemów biologicznych], pp. 109–118. Polish Academy of Sciences Publishers, 2009.

[13] R. Tadeusiewicz, *Inżynieria Biomedyczna*, ch. Models of biological systems and their application [In Polish: Modele systemów biologicznych i ich zastosowania], pp. 191–193. Kraków UWND, 2008.

[14] R. Tadeusiewicz, *Problemy Biocybernetyki i Inżynierii Biomedycznej, Tom 5*, ch. Cybernetic modeling of the parts of neural system involved in moving control process [In Polish: Cybernetyczne modelowanie fragmentów systemu nerwowego zaangażowanych w procesie sterowania ruchem], pp. 125–136. WKiŁ, Warszawa 1990.

[15] E. Mikołajewska and D. Mikołajewski, "Implikacje neuroplastyczności mózgu na modelowanie procesów umysłowych człowieka," *Kognitywistyka i Media w Edukacji*, no. 2, pp. 199–207, 2010.

[16] E. Mikołajewska and D. Mikołajewski, "Role of brainstem within human body systems – computational approach," *Journal of Health Sciences*, vol. 1, no. 2, pp. 95–106, 2012.

[17] E. Mikołajewska and D. Mikołajewski, "Consciousness disorders as the possible effect of brainstem activity failure – computational approach," *Journal of Health Sciences*, vol. 2, no. 2, pp. 7–18, 2012.

[18] W. Duch, W. Nowak, J. Meller, G. Osinski, K. Dobosz, D. Mikolajewski, and G. M. Wójcik, "Consciousness and attention in autism spectrum disorders," in *Proceedings of Cracow Grid Workshop 2010*, pp. 202–211, 2011.

[19] J. Bower and D. Beeman, *The Book of Genesis – Exploring Realistic Neural Models with GEneral NEural SImulation System*. Springer, 1998.

[20] G. M. Wójcik, *Obliczenia płynowe w modelowaniu mózgu*. Warszawa: Akademicka Oficyna Wydawnicza Exit, 2011.

[21] S. Greenfield, *Tajemnice mózgu*. Warszawa: Diogenes, 1998.

[22] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *Journal of Physiology*, vol. 117, pp. 500–544, 1952.

[23] W. A. Kamiński, *Neurocybernetyka teoretyczna pod redakcją naukową Ryszarda Tadeusieiwcza*, ch. Modelowanie pojedynczych komórek nerwowych, pp. 75–86. Wydawnictwo Uniwersytetu Warszawskiego, 2010.

[24] R. Tadeusiewicz, *Sieci neuronowe*. Warszawa: Akademicka Oficyna Wydawnicza RM, 1993.

[25] G. M. Wójcik and S. Kotyra, *Środowisko programisty*. Lublin: Instytut Informatyki UMCS, 2011.

[26] "Mpich 2." http://www.mcs.anl.gov/research/projects/mpich2/, grudzień 2010.

[27] "General neural simulation system." http://genesis-sim.org/, December 2010.

[28] G. M. Wojcik, "Self-organising criticality in the simulated models of the rat cortical microcircuits," *Neurocomputing*, no. 79, pp. 61–67, 2012.

[29] G. M. Wojcik, "Electrical parameters influence on the dynamics of the hodgkin-huxley liquid state machine," *Neurocomputing*, no. 79, pp. 68–78, 2012.

[30] G. M. Wojcik and W. A. Kaminski, "Self-organised criticality as a function of connections' number in the model of the rat somatosensory cortex," in *Computational Science – ICCS 2008*, vol. 5101 of *Lecture Notes in Computer Science*, pp. 620–629, Springer, 2008.

[31] G. M. Wojcik and W. A. Kaminski, "Nonlinear behaviour in mpi-parallelised model of the rat somatosensory cortex," *Informatica*, vol. 19, no. 3, pp. 461–470, 2008.

[32] G. M. Wojcik, W. A. Kaminski, and P. Matejanka, "Self-organised criticality in a model of the rat somatosensory cortex," in *Parallel Computing Technologies*, vol. 4671 of *Lecture Notes in Computer Science*, pp. 468–475, Springer, 2007.

[33] G. M. Wojcik and W. A. Kaminski, "Liquid state machine and its separation ability as function of electrical parameters of cell," *Neurocomputing*, vol. 70, no. 13–15, pp. 2593–2697, 2007.

[34] G. M. Wojcik and W. A. Kaminski, "Liquid computations and large simulations of the mammalian visual cortex," in *Computational Science – ICCS 2006*, vol. 3992 of *Lecture Notes in Computer Science*, pp. 94–101, Springer, 2006.

[35] G. M. Wojcik and W. A. Kaminski, "Large scalable simulations of mammalian visual cortex," in *Parallel Processing and Applied Mathematics*, vol. 3911 of *Lecture Notes in Computer Science*, pp. 399–405, Springer, 2005.

[36] "Clusterix – krajowy klaster linuksowy." http://www.clusterix.pcz.pl/, grudzień 2010.