Нижегородский государственный университет им. Н.И. Лобачевского
Национальный исследовательский университет
Программа повышение конкурентоспособности ННГУ им. Н.И. Лобачевского
Стратегическая инициатива 7 «Достижение лидирующих позиций в области суперкомпьютерных технологий и высокопроизводительных вычислений»

Основная образовательная программа
Нелинейные колебания и волны
Направление подготовки - 011800 Радиофизика
Учебно-методическая разработка по дисциплине
Моделирование сигнальных процессов в нейронных сетях мозга

Симонов А.Ю.

ИСПОЛЬЗОВАНИЕ СИМУЛЯТОРА НЕЙРОННЫХ СЕТЕЙ NEST В ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ВЫЧИСЛЕНИЯХ

Нижний Новгород
2014

УДК 519.876.5, 004.428.4, 004.942, 51-76, 591.181
ББК Е903.9
С37

С37 ИСПОЛЬЗОВАНИЕ СИМУЛЯТОРА НЕЙРОННЫХ СЕТЕЙ NEST В ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ВЫЧИСЛЕНИЯХ. Автор: Симонов А.Ю.: учебно-методическое пособие / Нижний Новгород: Нижегородский госуниверситет, 2014. – 85 с.

Рецензент: д.ф.-м.н., **В.Б. Казанцев**

Методическое руководство содержит краткие теоретические сведения о программном продукте NEST с открытым исходным кодом для расчёта реализаций моделей спайковых нейронных сетей, а также практические рекомендации по использованию симулятора на уровне продвинутого пользователя. В руководстве приведена информация о необходимости использования стандартных симуляторов в задачах моделирования, приведено описание симулятора NEST, его структура, основы использования, приведена информация об исходных кодах симулятора, описаны принципы организации параллельных вычислений, приведены руководства по установке и разработке пользовательских модулей.

Пособие предназначено для студентов 1 курса магистратуры, обучающихся по направлению подготовки "011800 Радиофизика", по профилю "Нелинейные колебания и волны" в рамках проведения практических работ по курсу "Моделирование сигнальных процессов в нейронных сетях мозга". Также пособие может быть использовано студентами старших курсов, обучающихся по направлению подготовки 010300 «Фундаментальная информатика и информационные технологии» и занимающимися математическим моделированием динамических процессов в нейронных системах мозга.

Ответственный за выпуск:
председатель методической комиссии _____ факультета ННГУ, _____

УДК 519.876.5, 004.428.4, 004.942, 51-76, 591.181
ББК Е903.9

© Нижегородский государственный университет им. Н.И. Лобачевского, 2014

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Современный этап развития науки и техники сопровождается интенсивным внедрением новых информационных технологий во множество сфер человеческой деятельности, и на сегодняшний день исследования в различных областях науки требуют использования передовых компьютерных систем. Благодаря непрерывному процессу развития и совершенствования ресурсы современной вычислительной техники многократно возросли и способны обеспечить решения задач такой сложности, которая в недавнем прошлом казалась недостижимой из-за недостатка вычислительной мощности [1]. Примером таких задач может служить компьютерное моделирование больших сложных систем. Математическое и компьютерное моделирование является одним из эффективных методов изучения. Задача моделирования состоит в разработке и анализе моделей реальных систем и процессов, протекающих в рассматриваемых системах, с целью получения объяснений их механизмов, предсказания феноменов [2]. Модельный подход в науке позволяет преодолеть ограничения и трудности, возникающие при постановке эксперимента в лабораторных или полевых условиях. Это достигается благодаря возможности проведения численных экспериментов, позволяющих исследовать отклик изучаемой системы на изменения ее параметров и начальных условий. Кроме того в рамках модельного подхода возможно выбирать значения параметров и начальных условий далеко за пределами наблюдаемых экспериментально. Это важно для выявления причин и механизмов наблюдаемых процессов, для поиска границ наблюдаемых динамических режимов. Компьютерное моделирование, в связи с этим, широко применяется во всех естественнонаучных направлениях современных исследований. Не является исключением и нейробиология, задача которой состоит в изучении функционирования головного мозга и нервной системы [3,4]. В данном контексте моделирование нейробиологических процессов сочетается с применением методов и подходов нелинейной динамики и радиофизики [5–7].

Мозг представляет собой сложнейший объект исследования, состоящий из миллиардов клеток [8–10]. Одним из типов клеток мозга являются нейроны.

Нейроны генерируют и передают электрические импульсы и способны образовывать сети посредством контактов, именуемых синапсами. Существуют и другие типы клеток в мозге, – например, глиальные клетки, которые выполняют такие важные дополнительные функции, как питание нейронов, поддержка гомеостаза, модулирование процессов передачи сигнала и др. Существующие экспериментальные методы исследования активности нейронных сетей мозга, направленные на изучение принципиально сетевых эффектов, во-первых, обладают рядом технических ограничений, связанных со сложностью объекта исследования, а во-вторых, являются крайне дорогостоящими. Для преодоления данного препятствия широко применяется подход, связанный с математическим и компьютерным моделированием изучаемых процессов. Моделирование подобных систем, сложных по топологии внутренних связей и состоящих из большого количества элементов, с помощью современных персональных компьютеров представляет серьёзную проблему, ввиду большой вычислительной нагрузки, требуемой для расчёта получаемых моделей. Однако использование суперкомпьютерных технологий и параллельных вычислений устраняет эту преграду и позволяет применять большее многообразие методов моделирования.

В связи с ростом доступности вычислительных ресурсов, исследования с применением компьютерного моделирования становятся все более популярными. Следует отметить, что в настоящее время исследовательские центры вычислительной нейронауки привлекают существенные инвестиции со стороны правительства и частных фондов, особенно в США (Центр Слоан-Шварц вычислительной нейронауки в Гарварде, Принстоне, Колумбии, Университете Нью-Йорка и других ведущих университетов) и Германии (Центр Бернштейна вычислительной нейронауки, располагающийся в Берлине, Мюнхене, Геттингеме и Гейдельберге). Следует также указать ведущие мировые проекты по крупномасштабному моделированию структур и функций нейронных систем мозга, такие как The Human Brain Project, The Brain Corporation, SyNAPSE, The BRAIN Initiative. Например, в проекте The Blue Brain Project (Лозанна, Швейцария, http://bluebrain.epfl.ch/ – сайт проекта), направленном на создание детализированной модели области коры головного мозга на уровне сети

биофизических нейронов с реалистичной топологической структурой, расчёт модельных сетей производился с применением суперкомпьютерных технологий. Проект, финансировался швейцарским правительством и несколькими частными фондами, и получил звание флагманского проекта Европейской Комиссии “Будущие и Развивающиеся Технологии” (Future and Emerging Technologies) с финансированием размером в 0.5 миллиарда евро на 10 лет. После одобрения проект носит название The Human Brain Project (сайт проекта https://www.humanbrainproject.eu/en_GB). Другой проект носит название Brain Corporation (Сан-Диего, США, http://braincorporation.com/ – сайт проекта). Проект также связан с имитационным моделированием и направлен на создание микроэлектронного чипа, реализующего принципы распознавания визуальной информации, найденные в коре головного мозга млекопитающих.

Использование модельного подхода действительно выглядит многообещающим, так как в современной нейронауке существует круг вопросов, решение которых возможно только с использованием моделирования. Однако существует ряд проблем, связанных с тем, что результаты подобных исследований трудно проверить и воспроизвести. Для проведения компьютерных симуляций разработанной модели исследователю необходимо разрабатывать собственные средства численного интегрирования. Уровень навыков программиста у разных исследователей различен, исследователи могут применять разные численные методы для одних и тех же моделей, причём делая это не всегда корректно [11]. Эти и другие причины зачастую приводят к неправильным результатам и/или невозможности воспроизвести чужие результаты. С целью преодолеть эти проблемы научным сообществом разрабатываются тщательно проверенные специализированные программные продукты для симуляции нейронных сетей. Они являются хорошо документированными и адресованы широкому кругу пользователей. Данные программные средства позволяют стандартизировать код, что в свою очередь существенно упрощает взаимодействие исследовательских групп [12]. Многие из таких нейросимуляторов также содержат встроенные средства для параллельного программирования, обеспечивающие использование арсенала современной информационно-вычислительной техники для задач

моделирования активности нейронных сетей мозга [13–17]. Многие из этих нейросимуляторов нашли широкое применение в построении крупномасштабных моделей нейронных сетей. Наряду с программными средствами моделирования широкое применение в нейронауке получили симуляторы с аппаратно реализованными нейронными сетями [18–20]. Примером подобных симуляторов служит проект FACETS – платформа моделирующая работу примерно 10^6 нейронов. Нейроны расположены на нескольких связанных платах, на каждой их которых размещаются аналоговые сетевые ядра (ANCs). Эти ядра и являются главными элементами архитектуры FACETS и состоят из нейронов и синаптических связей (в среднем, каждый нейрон этой системы формирует до тысячи контактов с другими нейронами моделируемой сети).

К наиболее распространённым программным пакетам можно отнести NEST (NEuron Simulation Tool) [21] и NEURON [22,23]. Оба нейросимулятора являются масштабными проектами, вовлекающими большое количество специалистов, работающих в различных ведущих научных центрах, а их разработка имеет длительную историю.

В данном пособии приведены подробные инструкции использования нейросимулятора NEST для задач моделирования и расчёта динамики спайковых нейронных сетей. Приведён обзор устройства нейросимулятора, рассмотрены исходные коды некоторых модулей. Приложения дают подробную инструкцию об установке нейросимулятора NEST, разработке новых модулей, и добавления этих модулей в состав NEST.

ГЛАВА 1. ОПИСАНИЕ СИМУЛЯТОРА NEST.

1.1. Архитектура NEST

NEST – программный симулятор с открытым исходным кодом, используемый для моделирования сетей, биологически реалистичных и феноменологических элементов и связей. Нейросимулятор оптимизирован для расчёта больших нейрональных сетей, состоящих из сравнительно простых однокомпартментных нейронов. В настоящее время NEST способен симулировать модель, состоящую из 1.73×10^9 элементов (нейронов) и приблизительно 10^{13} связей (синапсов) между ними [24]. В нейросимуляторе NEST реализован нисходящий (сверху-вниз) подход к описанию модели нейронной сети. В соответствии с принципом иерархичности, моделируемые нейронные сети рассматриваются как структуры с многоуровневой организацией, которые могут быть удобно представлены в виде графов.

Принципы организации приложения NEST способствуют эффективному использованию вычислительных ресурсов компьютеров с многоядерным процессором, многопроцессорных компьютеров и кластеров. При моделировании на вычислительном кластере или многопроцессорных компьютерах, каждый компьютер или процессор воссоздает часть сети и хранит информацию о синаптических контактах только своих нейронов. Для распределения задач на все компьютеры (процессоры) NEST использует интерфейс мгновенных сообщений (MPI) и потоков POSIX (pthreads). О принципах организации и использовании параллельных вычислений описано в разделе «ГЛАВА 2. Параллельные вычисления с применением NEST». Построение модели сети в нейросимуляторе NEST осуществляется с использованием узлов и связей. В качестве узлов могут выступать нейроны, устройства и подсети, которые способны обмениваться (принимать и отправлять) событиями различного типа, к примеру, спайками или токами. Более подробная информация о событиях приведена при рассмотрении исходных кодов NEST в разделе «ГЛАВА 3. Обзор исходного кода симулятора NEST». В программном пакете NEST содержатся встроенные модели нейронов,

устройств, синапсов, а использование модульного принципа организации приложения, дает пользователю возможность создания собственных моделей. Пример разработки новой модели приведён в разделе «Приложение 2. Написание нового модуля», а установка модуля – в разделе «Приложение 3. Установка с добавленным модулем». Большинство реализованных моделей нейрона состоит из небольшого количества компартментов, что снижает степень детализации описываемых биофизических процессов и морфологических особенностей отдельной клетки. Нейроны образуют сеть посредством так называемых синаптических контактов, для каждого из которых может быть определена собственная динамика. Простейшая модель связи между узлами характеризуются весом (определяет силу взаимодействия между узлами) и временем задержки (время, требуемое на переход сигнала от одного узла к другому). В симуляторе встроены модели синапсов с реализованными механизмами пластичности, синаптической депрессии и восстановления, а также предусмотрены функции для создания различных топологических схем связей и структурирования больших сетей. Общая схема организации приложения представлена на рисунке 1.

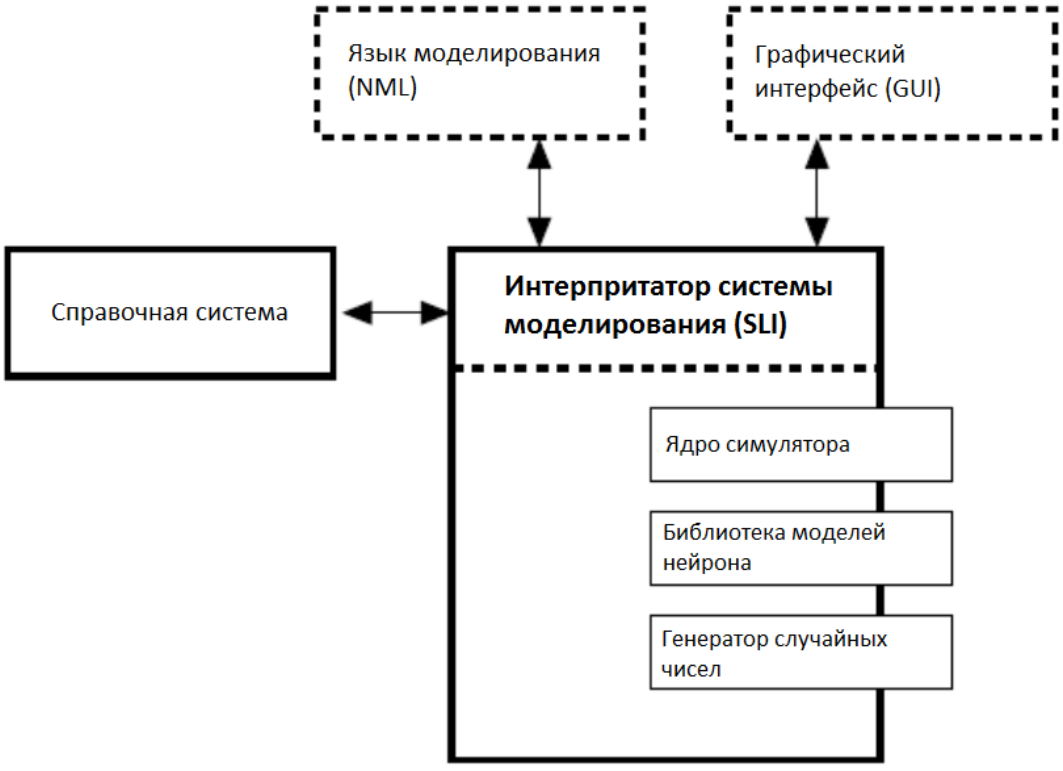

Рисунок 1. Архитектура нейросимулятора NEST.

Главные составляющие элементы системы: ядро и интерпретатор системы моделирования. Интерпретатор осуществляет взаимодействие графического интерфейса и языка моделирования с ядром системы.

Использование NEST напоминает проведение реального нейробиологического эксперимента. Симулируемая нейронная сеть генерирует сигналы, которые могут быть записаны с помощью специальных регистрирующих устройств, способных чтобы вывести результаты измерения на экран, в файл или память компьютера.

1.2. Основы использования NEST

NEST может быть использован как самостоятельное приложение, запускаемое нативно из операционной системы семейства Linux. При этом команды нейросимулятора вводятся либо в режиме командной строки, либо в виде последовательности команд, записанных в файл. При этом нейросимулятор использует свой собственный скриптовый язык, который носи название SLI. Другой возможностью использования NEST является вызов команд нейросимулятора из-под интерпретатора языка программирования python. Для этих целей разработчиками был добавлен специальный модуль PyNEST [25,26] Приведём здесь некоторые команды из основного набора синтаксиса PyNEST.

Для использования модуля NEST, обеспечивающего вызов его функций из-под оболочки интерпретатора python, необходимо запустить сам интерпретатор (например ipython) и выполнить команду

```
import nest
```

Если нейросимулятор установлен корректно и переменная окружения задана правильно (см. Приложение 1. Установка симулятора NEST на ОС семейства Linux), должно появиться следующее сообщение.

```
In [1]: import nest

        -- N E S T --

 Copyright (C) 2004 The NEST Initiative
 Version 2.4.2 Nov 15 2014 00:38:07

This program is provided AS IS and comes with
NO WARRANTY. See the file LICENSE for details.

Problems or suggestions?
 Website     : http://www.nest-initiative.org
 Mailing list: nest_user@nest-initiative.org

Type 'nest.help()' to find out more about NEST.
```

Ниже приводится список команд нейросимулятора, доступных из-под интерпретатора языка python. Полный список можно найти на интернет странице http://nest-initiative.org/index.php/PyNEST. Кроме того, описание любой функции доступно из интерпретатора вызовом

```
print nest.<function_name>.func_doc
```

где `<function_name>` – имя любой NEST команды (без треугольных скобок)

```
nest.help('object')
```

Выводит страницу помощи для объекта. Например, команда:

```
nest.help('iaf_neuron').
```

выведет справочную информацию о модели нейрона, которая называется `‘iaf_neuron’`.

```
Name: iaf_neuron - Leaky integrate-and-fire neuron model.

Description:
```

iaf_neuron is an implementation of a leaky integrate-and-fire model
with alpha-function shaped synaptic currents. Thus, synaptic currents
and the resulting post-synaptic potentials have a finite rise time.
The threshold crossing is followed by an absolute refractory period
during which the membrane potential is clamped to the resting potential.

The subthreshold membrane potential dynamics are given by [3]

dV_m/dt = - (V_m - E_L) / tau_m + I_syn(t) / C_m + I_e / C_m

where I_syn(t) is the sum of alpha-shaped synaptic currents

I_syn(t) = Sum[w_j alpha(t-t_j) for t_j in input spike times]

w_j is the synaptic weight of the connection through which the spike
at time t_j arrived. Each individual alpha-current is given by

alpha(t) = e * t/tau_s * e^{-t/tau_s} * Heaviside(t)

alpha(t=tau_s) == 1 is the maximum of the alpha-current.

The linear subthresold dynamics is integrated by the Exact
Integration scheme [1]. The neuron dynamics is solved on the time
grid given by the computation step size. Incoming as well as emitted
spikes are forced to that grid.

An additional state variable and the corresponding differential
equation represents a piecewise constant external current.

The general framework for the consistent formulation of systems with
neuron like dynamics interacting by point events is described in
[1]. A flow chart can be found in [2].

Critical tests for the formulation of the neuron model are the
comparisons of simulation results for different computation step

sizes. sli/testsuite/nest contains a number of such tests.

The iaf_neuron is the standard model used to check the consistency
of the nest simulation kernel because it is at the same time complex
enough to exhibit non-trivial dynamics and simple enough to compute
relevant measures analytically.

Parameters:

The following parameters can be set in the status dictionary.

V_m double - Membrane potential in mV
E_L double - Resting membrane potential in mV.
C_m double - Capacity of the membrane in pF
tau_m double - Membrane time constant in ms.
t_ref double - Duration of refractory period in ms.
V_th double - Spike threshold in mV.
V_reset double - Reset potential of the membrane in mV.
tau_syn double - Rise time of the excitatory synaptic alpha function in ms.
I_e double - Constant external input current in pA.

Note:
tau_m != tau_syn is required by the current implementation to avoid a
degenerate case of the ODE describing the model [1]. For very similar values,
numerics will be unstable.

References:
[1] Rotter S & Diesmann M (1999) Exact simulation of time-invariant linear
systems with applications to neuronal modeling. Biologial Cybernetics
81:381-402.
[2] Diesmann M, Gewaltig M-O, Rotter S, & Aertsen A (2001) State space
analysis of synchronous spiking in cortical neural networks.
Neurocomputing 38-40:565-571.
[3] Morrison A, Straube S, Plesser H E, & Diesmann M (2007) Exact subthreshold

integration with continuous spike times in discrete time neural network simulations. Neural Computation 19:47-79.

Sends: SpikeEvent

Receives: SpikeEvent, CurrentEvent, DataLoggingRequest

Author: September 1999, Diesmann, Gewaltig
SeeAlso: iaf_psc_alpha, testsuite::test_iaf

```
Models(mtype = 'all', sel = None)
```

Возвращает список всех доступных моделей нейросимулятора NEST (нейронов, устройств, синапсов). Если указать значение аргумента `mtype = 'nodes'` для отображения только моделей узлов (нейронов и устройств) или `mtype = 'synapses'` для отображения списка моделей одних только синапсов. Аргумент `sel` может быть строкой, используемой для фильтрации результата, возвращая только те модели, имя которых содержит эту строку.

```
CopyModel(existing, new, params = None)
```

Создаёт новую модель копированием уже существующей. `existing` и `new` – имена существующей и новой моделей соответственно. Заметим, что все имена моделей в PyNEST являются строками, следовательно, передавать эти имена в функцию копирования модели, равно как и в любую другую функцию, нужно в виде строки, то есть, заключив имя в одинарные или двойные кавычки. Параметры по умолчанию для новой модели задаются в виде словаря или копируются из старой модели, в случае отсутствия данного параметра в предоставленном словаре.

```
GetDefaults(model)
```

Возвращает словарь с параметрами по умолчанию для указанной модели.

```
SetDefaults(model, params)
```

Задаёт для данной модели набор новых параметров по умолчанию, указанных в виде словаря параметров.

```
Create(model, n = 1, params = None)
```

Создаёт `n` экземпляров модели указанного типа. Параметры новых узлов могут быть указаны в виде одного словаря или списка словарей длины `n`. При отсутствии словаря параметров используются параметры по умолчанию.

```
GetStatus(nodes, keys = None)
```

Возвращает словари параметров для данного списка узлов. Если заданы ключи `keys`, вместо словарей возвращается список одних только значений. Ключи могут быть также заданы в виде списка. В этом случае возвращаемый список будет содержать списки значений.

```
SetStatus(nodes, params, val = None)
```

Задаёт параметры `params` для узлов `nodes`. Параметры могут быть указаны в виде одного словаря или в виде списка словарей. Если значение `val` задано, `params` должен быть задан в виде строки, обозначающей имя того атрибута, значение которого изменяется на значение `val`. val может быть одним значением или списком такой же длины, как и длина списка узлов `nodes`.

```
Connect(pre, post)
Connect(pre, post, conn_spec)
Connect(pre, post, conn_spec, syn_spec)
```

Создаёт связи с выбором модели синапса (по умолчанию `'static_synapse'`) между узлами из списка `pre` и узлами из списка `post`. Списки узлов должны быть одинаковой длины. `conn_spec` может быть либо строчкой, содержащей названия правила формирования связей (по умолчанию `'one_to_one'`), либо словарём, задающим правило и специфичные для этого правила параметры (например, `'indegree'`), которые обязательно должны быть указаны в этом случае. Кроме

того, флаги, позволяющие или запрещающие связи нейрона самого с собой (`'autapses'`, по умолчанию: `True`) и множественные связи между одной и той же парой нейронов (`'multapses'`, по умолчанию: `True`), также могут быть частью этого словаря.

```
GetConnections(source, target = None, synapse_type = None)
```

Возвращает массив идентификаторов для тех связей, которые удовлетворяют заданным параметрам. Если заданы `target` и/или `synapse_type`, они должны быть одними значениями, списками длины 1 или такой же длины как `source`. Для получения/изменения найденных связей используется `GetStatus()/SetStatus()`.

```
Simulate(t)
```

Запускает симуляцию сети на `t` миллисекунд.

```
ResetKernel()
```

Сбрасывает ядро симулятора. Данная операция разрушает всю сеть, а также все модели, созданные с помощью `CopyModel()`. Вызов данной функции эквивалентен перезапуску нейросимулятора.

```
ResetNetwork()
```

Сбрасывает все узлы и связи к их изначальному состоянию.

```
GetKernelStatus()
```

Возвращает словарь с параметрами ядра симулятора.

```
SetKernelStatus(params)
```

Задаёт параметры для ядра симулятора. `params` – словарь с параметрами. Чтобы посмотреть вид этого словаря (полный список параметров ядра) можно воспользоваться предыдущей командой.

```
computername@username:~/ $ ipython
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
Type "copyright", "credits" or "license" for more information.

IPython 1.2.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import nest

               -- N E S T --

  Copyright (C) 2004 The NEST Initiative
  Version 2.4.2 Nov 15 2014 00:38:07

This program is provided AS IS and comes with
NO WARRANTY. See the file LICENSE for details.

Problems or suggestions?
  Website     : http://www.nest-initiative.org
  Mailing list: nest_user@nest-initiative.org

Type 'nest.help()' to find out more about NEST.

In [2]: nest.Models()
Out[2]:
(u'ac_generator',
 u'aeif_cond_alpha',
 u'aeif_cond_alpha_RK5',
 u'aeif_cond_alpha_multisynapse',
 u'aeif_cond_exp',
 u'cont_delay_synapse',
 u'correlation_detector',
 u'correlomatrix_detector',
 u'dc_generator',
 u'gamma_sup_generator',
 u'ginzburg_neuron',
 u'hh_cond_exp_traub',
 u'hh_psc_alpha',
 u'ht_neuron',
```

```
 u'ht_synapse',
 u'iaf_chs_2007',
 u'iaf_chxk_2008',
 u'iaf_cond_alpha',
 u'iaf_cond_alpha_mc',
 u'iaf_cond_exp',
 u'iaf_cond_exp_sfa_rr',
 u'iaf_neuron',
 u'iaf_psc_alpha',
 u'iaf_psc_alpha_canon',
 u'iaf_psc_alpha_multisynapse',
 u'iaf_psc_alpha_presc',
 u'iaf_psc_delta',
 u'iaf_psc_delta_canon',
 u'iaf_psc_exp',
 u'iaf_psc_exp_multisynapse',
 u'iaf_psc_exp_ps',
 u'iaf_tum_2000',
 u'izhik_cond_exp',
 u'izhikevich',
 u'mat2_psc_exp',
 u'mcculloch_pitts_neuron',
 u'mip_generator',
 u'multimeter',
 u'noise_generator',
 u'parrot_neuron',
 u'parrot_neuron_ps',
 u'poisson_generator',
 u'poisson_generator_ps',
 u'pp_pop_psc_delta',
 u'pp_psc_delta',
 u'ppd_sup_generator',
 u'pulsepacket_generator',
 u'quantal_stp_synapse',
 u'sinusoidal_gamma_generator',
 u'sinusoidal_poisson_generator',
 u'sli_neuron',
 u'spike_detector',
 u'spike_generator',
 u'spin_detector',
 u'static_synapse',
 u'static_synapse_hom_wd',
 u'stdp_dopamine_synapse',
```

```
 u'stdp_facetshw_synapse_hom',
 u'stdp_pl_synapse_hom',
 u'stdp_synapse',
 u'stdp_synapse_hom',
 u'step_current_generator',
 u'subnet',
 u'topology_layer_free',
 u'topology_layer_free_3d',
 u'topology_layer_grid',
 u'topology_layer_grid_3d',
 u'tsodyks2_synapse',
 u'tsodyks_synapse',
 u'voltmeter',
 u'volume_transmitter')
```

1.3. Обзор синапсов в NEST

Различают два типа синапсов в NEST: первый – это связующий объект, который управляет весом и задержкой синапса, второй – код, реализующий синаптические токи и проводимости, появляющиеся в результате спайков. Последний кодируется всегда в классе модели нейрона.

Все целевые нейроны для заданного нейрона сохранены в целевом списке, содержащем один связующий объект на цель. Когда спайк отправлен, NEST переходит в целевой список и посылает информацию о спайке, то есть весе, задержке и времени спайка, каждому принимающему нейрону. Некоторые синапсы реализуют пластичность, зависящую от времени спайка (STDP), иными словами они изменяют синаптический вес, содержащийся в связующем объекте, основываясь на спайковой истории принимающего нейрона. Такие синапсы обращаются к спайковой истории через специальные функции класса принимающих нейронов, которые должны быть получены из узла архивирования (`ArchivingNode`). Когда спайк достигает нейрона посредством обращения к его дескриптору (`SpikeEvent`), `SpikeEvent` предоставляет информацию о синаптическом весе, задержке и времени спайка, так что фактическое время достижения спайком нейрона может быть легко вычислено. В частности, вся

динамика синаптических токов и проводимостей может быть реализована в нейрональной модели. Для более глубокого изучения этого вопроса предлагается самостоятельно рассмотреть `iaf_neuron` и `iaf_cond_alpha`. На сегодняшний момент не имеется никаких общедоступных моделей с синапсами, зависящими от напряжения, такими как NMDA-каналы. Простым способом внести такую зависимость от напряжения в модель, например, в `iaf_cond_alpha` модель, будет умножить синаптическую проводимость *g(t)* на сигмоидную функцию *m(V)* в выражении, вычисляющем правую часть уравнения мембранного потенциала. Немного неудачно, что нужно включать код для выражения синаптических токов/проводимостей каждый раз заново в модели каждого нового нейрона, но для многих более простых моделей нейронов это означает более эффективный код, чем можно было бы получить, если бы текущая синаптическая динамика была представлена отдельными объектами. Говоря об NMDA-каналах, может возникнуть вопрос о том, как дифференцировать входящий сигнал от различных типов каналов на нейроне, например, AMPA, GABA_A, GABA_B и NMDA-каналов. Большинство моделей NEST в этом плане очень упрощены: они различают в основном возбуждающие и тормозящие синапсы, входящий импульс с положительным весом обрабатывается возбуждающим, а с отрицательным весом – тормозящим синапсом. Большинство каналов может быть обработано с использованием механизма порта тип рецептора/получатель в NEST.

Время имеет большое значение при прохождении спайка. В особенности, когда спайковый эффект на нейроне зависит от истории или состояния нейрона, крайне важно учитывать эти факторы. Это является причиной того, что информация об истории нейрона доступна только при помощи интерфейса `ArchivingNode`, в то время как данные о состоянии нейрона выводятся только через `UniversalDataLogger` интерфейс.

NEST передает спайки пачками. Моделирование продолжается до времени минимальной задержки (`min_delay`). На протяжении этого времени любые сгенерированные спайки сохраняются в центральную очередь. После каждого интервала `min_delay` спайки доставляются из центральной очереди с помощью

`ConnectionManager` и `ConnectionObjects` к отдельным нейронам. Все задержки обрабатываются в нейроне, как описано выше. Это означает, что, когда спайк проходит через `ConnectionObject`, фактическое биологическое время прибытия спайка (время, когда спайк ушел от отправителя плюс задержка) не может превышать время максимальной задержки (`max_delay`). Что, в частности, в свою очередь может приводить к тому, что `ConnectionObjects` не могут выполнить вычисления, зависящие от состояния нейрона во время «биологического» прибытия спайка; они могут пользоваться только прошлыми данными. Другой важный момент, который необходимо отметить, то, что спайки проходят через `ConnectionObject` не в правильном порядке «биологического» времени поступления – они не упорядочены по времени.

ГЛАВА 2. ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ С ПРИМЕНЕНИЕМ NEST

2.1. Принципы организации параллельных вычислений

Увеличивающийся поток новых результатов в области нейронаук, получающих всё более широкую огласку из-за своей значимости для человечества, указывает на то, что за последние несколько лет науки о мозге претерпели существенный скачок, продвинувшись как в выявлении фундаментальных механизмов функционирования нейронных систем, так и в понимании причин развития нейродегенеративных заболеваний, разработке стратегий их профилактики, диагностики и лечения. Стали появляться специализированные научные институты и лаборатории, а крупнейшие конференции, собирающие специалистов в области наук о мозге, стали насчитывать десятки тысяч участников. Не за горами видится и прикладное применение заимствованных у живых систем параллельных алгоритмов обработки информации, обучения, формирования целеполагающего поведения и способности эффективно действовать в условиях непредсказуемо меняющейся внешней среды. Одной из важных причин такого прорыва является развитие экспериментальных методов. В настоящее время все нейробиологические эксперименты носят принципиально междисциплинарный характер и вовлекают для проведения специалистов всё новых передовых областей современной науки. С развитием экспериментальных методов в современной нейробиологии увеличивается масштаб и количество полученных данных, иллюстрирующих всё новые закономерности. Но в это же время растёт специализация и детализация накапливаемых знаний, что приводит к формированию всё более разрозненной целостной картины. На путь решения этой проблемы встаёт математическое компьютерное моделирование.

В течение последних десятилетий технологии компьютерного моделирования достаточно сильно продвинулись и усовершенствовались. Суперкомпьютерные технологии стали более доступными, существенно выросла вычислительная

мощность современных кластеров и суперкомпьютеров, разработаны более новые и эффективные подходы к параллельным вычислениям. Методы и алгоритмы компьютерной симуляции спайковых нейронных сетей также претерпели большие изменения, позволяющие, например, проведение вычислительных экспериментов на масштабе локальных кортикальных структур мозга (сети, состоящие из 10^5 нейронов и 10^9 синапсов). При этом такие модели учитывают такие принципиально важные нейрофизиологические явления, как синаптическая пластичность. Однако подобные модели всё ещё имеют существенные ограничения. Например, известно, что входящие синаптические контакты нейрона коры головного мозга, формируемые от нейронов, расположенных в той же самой области, в среднем составляют около половины всех синапсов данного нейрона. Другая половина синапсов принимает сигнал от удалённых структур. В ограниченных моделях локальных сетей эта вторая половина обычно моделируется как случайный или постоянный входящий сигнал. Для достижения реалистичности входящих сигналов требуется разработка крупномасштабных моделей.

Начиная с версии 2.0, NEST способен осуществлять симуляцию на многоядерных, многопроцессорных компьютерах и компьютерных кластерах, используя два пути распараллеливания: симуляцию в параллельных потоках и распределенную симуляцию. Первый реализуется с помощью POSIX потоков, в то время как второй реализуется преимущественно через интерфейс мгновенных сообщений (MPI). Использование потоков позволяет воспользоваться преимуществом многоядерных и многопроцессорных компьютеров без привлечения дополнительного программного обеспечения. Использование распределенных вычислений имеет дополнительную выгоду в том, что позволяет проводить более масштабное моделирование, чем могло бы позволить использование памяти одного компьютера. Кроме того, общее время соединения и время выполнения моделирования могут быть уменьшены, поскольку сеть основана на множестве компьютеров, соединенных параллельно. Следующие параграфы описывают удобства, возможности и преимущества для параллельного и распределенного вычисления в нейросимуляторе NEST более подробно.

Чтобы упростить обработку распределения узлов потоков и процесса, основанного на распараллеливании, используется концепция локальных и удаленных потоков, называемых виртуальными процессами. Виртуальный процесс (VP) – это поток, находящийся в одном из процессов MPI. Виртуальные процессы распределены согласно модульно-ориентированному алгоритму в MPI процессах и непрерывно подсчитываются в течение всех процессов. Общее представление представлено на рисунке 2.

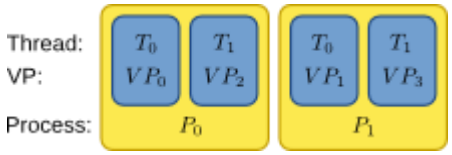

Рисунок 2. Общая схема подсчета потоков (T), виртуальных процессов (VP) и процессов MPI (P) в NEST

Алгоритм организации параллельных вычислений в NEST иллюстрирует следующий пример на псевдокоде:

```
t <- 0
  WHILE t < T_stop
    PARALLEL on all virtual processes
      deliver all events due
      call U(S_t) forall nodes
    END
    exchange events between all virtual processes
    increment time t <- t+Delta
  END
```

Время симуляции разбивается на дискретную сетку с определённым шагом $t_i := i \cdot \Delta$. Запускается цикл, пробегающий все шаги по этой временной сетке. Внутри цикла происходит распараллеливание вычислений между всеми виртуальными процессами.

Статус справочника каждого узла содержит три входа, которые связаны с параллельным вычислением:

- `local`: логический, который показывает, существует ли узел на локальном процессе или нет

- `thread`: идентификатор локального потока, к которому привязан узел

- `vp`: идентификатор виртуального процесса, к которому привязан узел.

Распределение узлов зависит от их типа. Нейроны привязываются автоматически к одному из виртуальных процессов также по модульно-заданному алгоритму. На всех других виртуальных процессах выделяется прокси узел. Присвоение происходит через $id_{VP}=id_{Node} \ mod \ N_{VP}$, где id_{VP} идентификатор виртуального процесса, к которому привязан нейрон, id_{Node} глобальный идентификатор узла, а N_{VP} общее число виртуальных процессов.

Устройства для моделирования и наблюдения за сетью повторяются в каждом потоке таким образом, что позволяет уравновесить нагрузку на разные потоки и минимизировать взаимодействие между различными потоками. Таким образом, устройства не имеют прокси на удаленных виртуальных процессах. Устройства сконфигурированы так, чтоб записывать данные в файл, что означает, что каждый поток будет записывать свой собственный файл с данными измерений нейронов. Имена файлов формируются согласно следующей схеме:

```
[model|label]-gid-vp.[dat|gdf]
```

Первая часть – это имя модели (например, voltmeter или spike_detector), или, если задано, ярлык записывающего устройства. Вторая часть – это глобальный идентификатор (GID) записывающего устройства. Третья часть – это идентификатор виртуального процесса, к которому привязано записывающее устройство, нумеруемое с нуля. Расширение gtf применяется для спайковых файлов, а dat для аналоговых записей. Ярлык и расширение файла для записывающего устройства могут быть установлены, как любые другие параметры узла, с использованием `SetStatus`.

Привязывание узлов к виртуальным процессам по умолчанию – удобный выбор в плане соблюдения баланса загрузки процессов, если это соответствует случайному распределению для сетей со случайными связями. Однако, для сетей с известной архитектурой и определённым направлением потока передачи сигналов, всегда существуют лучшие распределения. Для осуществления более мелкомодульного распределения, модель `subnet` имеет логический параметр `children_on_same_vp`, который вызывает/активирует/принуждает все узлы в подсети быть привязанными к тому же виртуальному процессу.

Узловое распределение для малой сети состоит из `spike_generator`, четырех `iaf_neurons`, и `spike_detector` в примере с двумя процессами с двумя потоками, каждый показан в следующем рисунке 3.

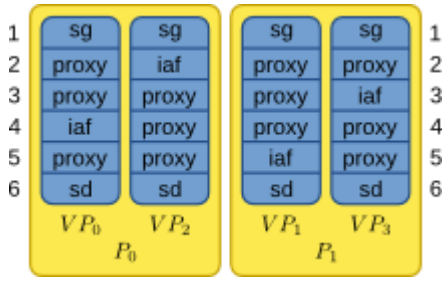

Рисунок 3. Иллюстрация узлового распределения, где sg - spike_generator, iaf – iaf_neuron, sd – spike_detector. Числа слева и справа демонстрируют глобальные идентификаторы

На рисунке 4 представлены графики, показывающие зависимость производительности NEST о количества задействованных процессоров. Рассчитывается сеть из 12500 элементов, описываемая моделью integrate-and-fire (80 % возбуждающих, 20 % тормозных нейронов), на вход каждого из которых поступает сигнал от 10% всех нейронов. Общее количество синапсов в модели равняется $1{,}56\times10^{7}$. Нейроны инициализируются случайными мембранными потенциалами и в дальнейшем стимулируются нерегулярным воздействием.

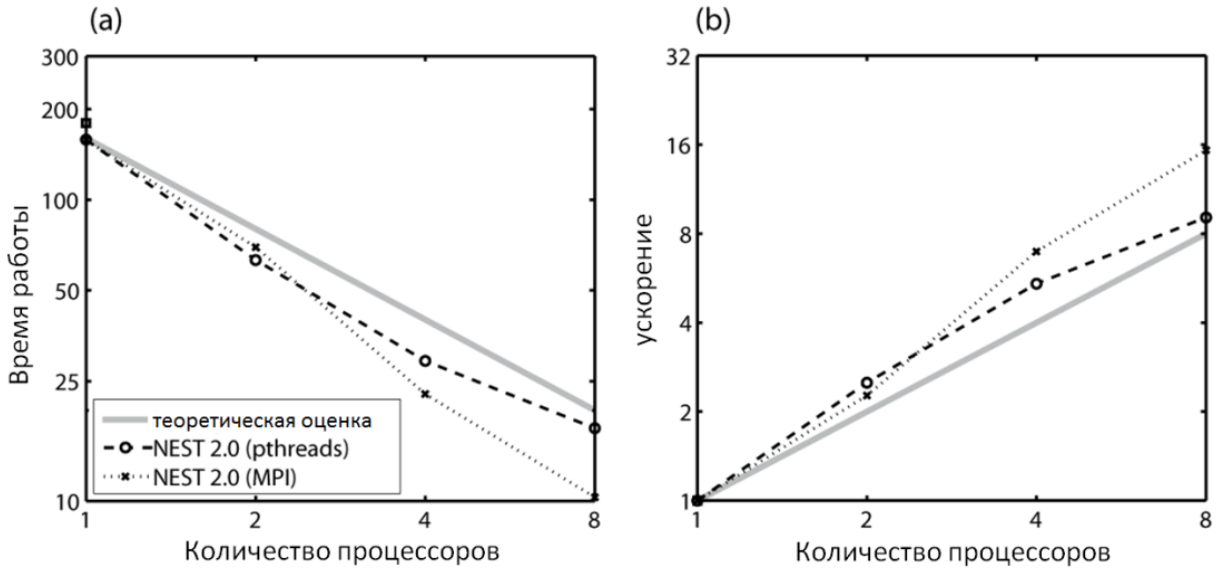

Рисунок 4. Моделирование нейронной сети в NEST. (a) абсолютное время работы и (b) ускорение как функция от количества задействованных процессоров.

2.2. Использование параллельных вычислений.

Распараллеливание потоков скомпилировано в NEST по умолчанию и должно работать на всех компьютерах с MacOS и Linux без дополнительных требований. Для использования параллельных вычислений для моделирования, необходимое количество потоков должно быть установлено до начала создания любых узлов и соединений. SLI команда для этого следующая:

```
0 << /local_num_threads T >> SetStatus
```

В PyNEST количество потоков задается следующим образом:

```
SetKernelStatus({"local_num_threads": T})
```

Обычно для `T` задается значение, равное количеству процессов, доступных на компьютере. Тем не менее, последние наблюдения показали, что может быть выгодно превысить намеченную сумму ядер максимум на два потока на ядро, что приводит к улучшению скорости моделирования на 20-30% (обширные

исследования для определения верней границы, где такое превышение все еще выгодно, пока еще не представлены). Наиболее вероятная причина этого в мощной оптимизации, встроенной в большинство текущих процессов (например, одновременная многопоточность, в чипах Intel названная hyper-threading) и различные ядерные планировщики.

Для компилирования в NEST распределенных вычислений необходима библиотека, подключенная к MPI-интерфейсу. В настоящее время поддерживается MPICH, OpenMPI и ScaliMPI. Несколько других реализаций также могут быть задействованы в работе, но возможно потребуют небольшой доработки. Большинство программ под Linux укомплектованы версиями MPICH и OpenMPI. В случае заранее укомплектованной MPI-библиотеки необходимо наличие и библиотеки и установочных пакетов.

Если MPI-библиотека и заглавные файлы установлены в стандартные директории системы, скорее всего команда

```
$NEST_SOURCE_DIR/configure --with-mpi
```

найдет их (`$NEST_SOURCE_DIR` – это директория, содержащая исходные коды NEST). Если установка MPI производится из другой локации, можно использовать следующую команду

```
$NEST_SOURCE_DIR/configure --with-mpi=/path/to/mpi
```

для указания основной директории, где находятся каталоги MPI `bin`, `include`, и `lib`. Этот вариант более гибкий и может быть использован, если установка MPI производится из других источников. В некоторых случаях необходимо задать компилятор MPI в явном виде, например

```
$NEST_SOURCE_DIR/configure CC=mpicc CXX=mpicxx --with-mpi
```

Распределенные вычисления не могут быть выполнены интерактивно, что означает, что моделирование должно быть обеспечено сценарием выполнения, который не может быть изменен по сравнению со сценарием выполнения

серийных симуляций: межпроцессорные коммуникации и узлы распределения заданы очевидно внутри NEST.

Для запуска распределенных вычислений на 128 процессорах необходимо выполнить команду

```
mpirun -np 128 nest script.sli
```

В случае использования PyNEST команда выглядит следующим образом

```
mpirun -np 128 python script.py
```

Надо отметить, что PyNEST будет корректно работать вместе с OpenMPI. Для деталей использования `mpirun` нужно обратиться к документации, содержащейся в библиотеке.

В интерфейсе MPI каждый процесс имеет свой собственный уникальный идентификатор. Обычно, нет необходимости отслеживать/следить/заботиться об этом. Однако, в особых случаях может потребоваться использовать его (например, открытие файлов с разными именами для каждого процесса). Категория процесса доступна по следующей команде

```
Rank
```

Количество MPI процессов можно узнать по команде

```
NumProcesses
```

Узнать имя каждой машины можно, используя команду

```
MPIProcessorName
```

Для достижения сходимости результатов симуляций и воспроизводимости, даже при разных стратегиях распараллеливания, необходимо сохранять количество виртуальных процессов постоянным. Моделирование с определенным количеством виртуальных симуляторов всегда будет показывать те же результаты, не важно как они распределены между потоками и процессорами, учитывая сиды

для генераторов случайных чисел на различных виртуальных процессах. Чтобы держать количество виртуальных процессов постоянным, в NEST есть `total_virtual_procs`.

Нижеследующий листинг содержит завершенный процесс симуляции в SLI с четырьмя нейронами, соединенными в цепь. Первый нейрон получает случайный сигнал от poisson_generator и спайки от всех четырех нейронов записываются в файлы.

```
0 << /total_num_virtual_procs 4 >> SetStatus
/poisson_generator << /rate 50000.0 >> Create /pg Set
/iaf_neuron 4 Create ;
/spike_detector << /to_file true >> Create /sd Set
pg 2 1000.0 1.0 Connect
2 3 1000.0 1.0 Connect
3 4 1000.0 1.0 Connect
4 5  1000.0 1.0 Connect
[ 2 3 4 5 ] sd ConvergentConnect
100.0 Simulate
```

Предположим, что скрипт сохранился как `simulation.sli`, так что можно запустить процесс моделирования, используя `mpirun` и различное количество процессов

```
mkdir 4vp_1p 4vp_2p 4vp_4p
cd 4vp_1p
mpirun -np 1 nest ../simulation.sli
cd ../4vp_2p
mpirun -np 2 nest ../simulation.sli
cd ../4vp_4p
mpirun -np 4 nest ../simulation.sli
```

```
cd ..
diff 4vp_1p 4vp_2p
diff 4vp_1p 4vp_4p
```

Результатом всех вариантов эксперимента будут четыре файла с данными (`spike_detector-6-0.gdf, spike_detector-6-1.gdf, spike-detector-6-2.gdf, spike_detector-6-3.gdf`). Используя утилиту `diff` на трех директориях, показано, что все они содержат одинаковые спайки, что означает, что результаты симуляции одинаково независимы от деталей алгоритма распараллеливания.

При использовании PyNEST вместо SLI вышепредставленный код будет работать в точности так же. Единственная разница заключается в замене `simulation.sli` на `simulation.py`. Код PyNEST показан ниже:

```
from nest import *
SetKernelStatus({"total_num_virtual_procs": 4})
pg = Create("poisson_generator", params={"rate": 50000.0})
n = Create("iaf_neuron", 4)
sd = Create("spike_detector", params={"to_file": True})
Connect(pg, [n[0]], 1000.0, 1.0)
Connect([n[0]], [n[1]], 1000.0, 1.0)
Connect([n[1]], [n[2]], 1000.0, 1.0)
Connect([n[2]], [n[3]], 1000.0, 1.0)
ConvergentConnect(n, sd)
Simulate(100.0)
```

Ход эксперимента показан в следующем листинге:

```
mkdir 4vp_1p 4vp_2p 4vp_4p
cd 4vp_1p
```

```
mpirun -np 1 python ../simulation.py
cd ../4vp_2p
mpirun -np 2 python ../simulation.py
cd ../4vp_4p
mpirun -np 4 python ../simulation.py
cd ..
diff 4vp_1p 4vp_2p
diff 4vp_1p 4vp_4p
```

ГЛАВА 3. ОБЗОР ИСХОДНОГО КОДА СИМУЛЯТОРА NEST

3.1. Каталоги и файлы исходных кодов NEST

В данной главе мы рассмотрим файлы исходного кода нейросимулятора NEST. Для начала необходимо скачать архив, содержащий исходные коды. Сделать это можно с официального сайта сообщества по ссылке http://www.nest-simulator.org/download/gplreleases/nest-2.4.2.tar.gz. Распаковав архив в удобную директорию, посмотрим список файлов и каталогов, находящихся внутри.

Как видно, NEST представляет собой достаточно сложную структуру, состоящую из множества модулей и компонентов, исходные коды которых расположены в 16 каталогах. В рамках данного пособия мы рассмотрим лишь некоторые из них. Читатель может обратиться к документации самостоятельно, а также самостоятельно просматривать исходные коды для более детального изучения NEST.

Директория `nest/conngen` содержит модуль, реализующий генератор связей *ConnectionGenerator*. ConnectionGenerator API представляет собой стандартный интерфейс поддержки эффективной генерации сетевой структуры в процессе настройки сетевой модели в симуляторах нейронных сетей. Он используется как абстракция, изолирующая обе стороны API: любой нейросимулятор может использовать данный генератор связей, а также использовать любую библиотеку, предоставляющую интерфейс ConnectionGenerator.

Каталог `nest/doc` содержит документацию к нейросимулятору.

Рассмотрим далее содержание каталога `nest/examples`. Внутри в каталоге `nest/examples/nest` находятся скрипты, написанные на внутреннем языке нейросимулятора NEST SLI. Данные скрипты реализуют примеры использования NEST. В качестве примеров выбраны результаты из области компьютерного моделирования активности нейронных сетей мозга, полученные за последние годы, публикация которых в научных журналах вызвала наиболее сильный резонанс в научном сообществе. Также в каталоге с примерами содержатся

простые примеры использования скриптового языка SLI для моделирования активности нейронных сетей при помощи нейросимулятора NEST. Читателям предлагается самостоятельно разобрать предложенные примеры для более глубокого ознакомления с использованием SLI в нейросимуляторе. Кроме примеров на языке SLI в директории `nest/examples` находится пользовательский модуль (каталог `nest/examples/MyModule`). Данный модуль представляет собой интерфейс для добавления новых моделей в NEST. Способ применения интерфейса добавления пользовательского модуля подробно описан на официальном сайте сообщества. Читатель может самостоятельно ознакомиться с ним по ссылке http://nest-initiative.org/Writing_an_Extension_Module. Такой способ позволяет наиболее быстро расширить набор существующих стандартных моделей нейросимулятора, не изменяя его исходный код. Однако при применении этого способа всё равно требуется повторная компиляция исходного кода нейросимулятора вместе с добавленным пользовательским модулем. После чего необходимо сообщить ядру о добавленном новом модуле. В данном пособии мы рассмотрим подробно добавление пользовательского модуля другим способом (смотри Приложение 3. Установка с добавленным модулем).

Директория `extras` содержит несколько вспомогательных скриптов для запуска NEST на распределённых вычислительных системах (nest_serial и nest_indirect), некоторые патчи, применяемые в ходе процесса установки на различные архитектуры, а также режим SLI для emacs.

Каталог `include` включает файлы, генерируемые в ходе процессов конфигурации и сборки. Эти файлы содержат информацию об архитектуре и системе, для которых NEST был скомпилирован.

Файлы библиотек для SLI и NEST содержатся в каталоге `lib`. Они включают файлы запуска и библиотеки обёртки SLI для встроенных функций.

В каталоге `libnestutil` находится дополнительная библиотека общего назначения, используемая SLI и NEST. Эта библиотека содержит функции выделения памяти, вспомогательные функции для управления потоками,

дебаггинга, численных и ссылочных указателей. В целом, этот каталог содержит все файлы, которые не подходят ни под одну из остальных категорий.

В директории `librandom` находится дополнительная библиотека для генераторов случайных чисел.

Папка `models` содержит исходные коды отдельных моделей NEST – синапсы, устройства, нейроны.

Директория nest содержит код, который после компиляции становится исполняемым файлом для запуска NEST как отдельной программы. В основном, `main.cpp` просто собирает все необходимые модули и линкует их в связке с требуемыми библиотеками.

Внутри каталога `nest/nestkernel` находятся исходные коды ядра симулятора.

В NEST также реализована технология симуляции с точностью большей, чем размер шага интегрирования. Файлы исходных кодов, реализующие эту технологию для моделей нейронов и синапсов и для соответствующих устройств, располагаются в каталоге `precise`.

Каталог `pynest` содержит исходные коды на языках C и Python для интерфейса PyNEST.

Директория `sli` включает исходные коды интерпретатора скриптового языка SLI (simulation language interpreter).

3.2. Описание модуля модели нейрона

В данном разделе рассмотрим структуру и устройство одного из модулей нейросимулятора NEST на примере модели Ходжкина-Хаксли. Все модели нейросимулятора находятся в каталоге `nest/models`. Исходные коды каждого модуля состоят из двух файлов с именами `<modulename>.h` и `<modulename>.cpp`, содержащими заголовки и сами исходные коды соответственно. В списке стандартных моделей нейросимулятора NEST

содержится две модели Ходжкина-Хаксли. Одна из них называется `‘hh_cond_exp_traub’`, а другая `‘hh_psc_alpha’`, а их соответствующие файлы с исходными кодами называются `hh_cond_exp_traub.h`, `hh_cond_exp_traub.cpp` и `hh_psc_alpha.h`, `hh_psc_alpha.cpp`. Откроем эти файлы и рассмотрим структуру кода внутри.

hh_psc_alpha.h, hh_psc_alpha.cpp

```
1  /*
2   *  hh_psc_alpha.***
3   *
4   *  This file is part of NEST.
…   …


…   …
20  *
21  */
```

В начале обоих файлов мы видим блок комментариев, содержащий информацию о названии файла, копирайте, лицензии и гарантии. Далее следуют привычный для разработчиков на C++ блок препроцессора языка, с включениями других файлов, содержащих заголовки используемых в данном модуле других модулей и библиотек, макроподстановками и командами условной компиляции (строчки 23 - 42, файл `hh_psc_alpha.h` и строчки 24 - 40, файл `hh_psc_alpha.cpp`).

Заметим, что исходные коды NEST достаточно хорошо документированы посредством множества подробных и понятных комментариев внутри самого кода. Например, далее, в заголовочном файле `hh_psc_alpha.h` следом за конструкцией, открывающей пространство имён `nest` (строчка 44) и оператором включения класса `vector` из пространства имён библиотеки `std` (строчка 46), следует комментарий, поясняющий необходимость декларирования функции вычисления правых частей для интегратора обыкновенных дифференциальных уравнений библиотеки GSL, используемой в данном модуле (строчки 48 - 57). За блоком комментариев следует само декларирование (строчка 59).

Далее следует другой блок комментариев, начинающийся со строчки `/* BeginDocumentation` (строчка 61). Такой блок комментариев присутствует в каждом модуле в его заголовочном файле. Он содержит специальную справочную информацию о данном модуле. После компиляции нейросимулятора эта справочная информация оказывается доступной через интерфейс командной строки. Для просмотра справки по модулю из командной строки необходимо ввести команду `nest.help('<modelname>')`, и данный блок комментариев будет выведен в консоль. Таким образом, для написания собственного модуля можно включить справочную информацию об этом модуле, добавив блок комментариев в заголовочный файл исходного кода, начинающийся со строчки `/* BeginDocumentation`.

hh_psc_alpha.h

```
61  /* BeginDocumentation
62  Name: hh_psc_alpha - Hodgkin Huxley neuron model.
63
64  Description:
65
66    hh_psc_alpha is an implementation of a spiking neuron using the
    Hodkin-Huxley formalism.
…   …


…   …
118
119 Receives: SpikeEvent, CurrentEvent, DataLoggingRequest
120
121 Authors: Schrader
122 SeeAlso: hh_cond_exp_traub
123 */
```

После встроенной в исходные коды документации модуля в заголовочном файле находится объявления членов класса данной модели.

hh_psc_alpha.h

```
…   …
125 class hh_psc_alpha:
```

```
126     public Archiving_Node
127   {
128
129   public:
130
131     hh_psc_alpha();
132     hh_psc_alpha(const hh_psc_alpha&);
133     ~hh_psc_alpha();
...  ...
```

Как видно из листинга, класс рассматриваемой модели наследуется от родительского класса `Archiving_Node`. Данный класс является базовым для всех модулей узлов нейросимулятора NEST. Модули других типов наследуются от своих родительских классов. Для изучения устройства базовых классов можно также просматривать их исходные коды. Они располагаются в директории `nest/nestkernel`. Как обычно, объявление членов класса начинается с объявлений конструктора по умолчанию и конструктора копирования, а также деструктора класса (строчки 131 - 133). Тела этих методов, располагающиеся в файле `hh_psc_alpha.cpp` после завершения ознакомления с членами класса `hh_psc_alpha`.

Далее следуют операторы импорта перегруженных виртуальных функций `connect_sender` и `handle`, которым предшествует очередной комментарий, повествующий о технической необходимости такого импорта. Первая из этих функций, `connect_sender`, служит для проверки, может ли данный узел принимать событие определённого типа. Например, рассматриваемый модуль, равно как и все модели нейронов нейросимулятора NEST, посылает спайки всем узлам, к которым он подключён. С точки зрения кода эти спайки являются событием типа `SpikeEvent&`, класс которого описан в заголовочном файле `event.h`, располагающемся также в директории с исходными кодами ядра нейросимулятора `nest/nestkernel`. Кроме спайков в NEST также определены другие типы событий – ответ на запрос о спаковом событии (`DSSpikeEvent`), ток (`CurrentEvent`), ответ на запрос о токовом событии (`DSCurrentEvent`), спайковая частота (`RateEvent`), запрос данных для записи или записанных данных

для отправки (`DataLoggingRequest`), ответ на этот запрос (`DataLoggingReply`), проводимость (`ConductanceEvent`), а также события, содержащие абстрактные данные, использующиеся для специфицеских данных, которые не подходят ни под одну из категорий событий (`DataEvent` и `DoubleDataEvent`). Все эти события, равно как и базовый класс события (`Event`), определены в заголовочном файле `event.h`. Функция `handle` служит для обработки входящего спайкового события и вызывается в том случае, когда на данный узел приходит входящий спайк. Обе перегруженные функции описаны в заголовочном файле `node.h` – базовом модуле узла, – который находится в папке с исходными кодами ядра NEST.

hh_psc_alpha.h

```
…   …
144     using Node::connect_sender;
145     using Node::handle;
146
147     port check_connection(Connection&, port);
148
149     void handle(SpikeEvent &);
150     void handle(CurrentEvent &);
151     void handle(DataLoggingRequest &);
152
153     port connect_sender(SpikeEvent &, port);
154     port connect_sender(CurrentEvent &, port);
155     port connect_sender(DataLoggingRequest &, port);
…   …
```

В приведённом фрагменте объявляются по три разных варианта функций `handle` и `connect_sender`, принимающие в качестве аргументов события спайка, тока и запроса данных, соответственно. При этом функция принимает и возвращает тип данных `port`. Этот тип описывает номер порта, через который узел сети принимает события от других узлов. События разных типов приходят на разные порты. Кроме того, одни и те же события могут приходить на разные порты. Это применяется при необходимости добавления нескольких способов обработки события одного и того же типа. Примером может служить наличие нескольких рецепторов на нейроне (AMPA, NMDA, GABA и др.). Каждый из

рецепторов активируется при поступлении входного спайкового события, генерируя токи, что приводит к изменению мембранного потенциала нейрона. Однако динамика, характерные временные масштабы и знак этих токов у разных рецепторов могут быть разными. Если требуется разнести события приходящие на разные рецепторы от разных нейронов и учесть в модели различия реакции нейрона на активацию этих рецепторов, можно определить несколько портов на один тип входящего события. О функции `check_connection`, равно как и об остальных методах, объявляемых в заголовочном файле, будет сказано немного позднее.

После объявлений частных и публичных методов следуют структуры данных. Структура `Parameters_` содержит значения параметров модели нейрона. Обозначения объявляемых элементов структуры, а также комментарии, сопровождающие их объявления, поясняют их назначение и соответствие математической формулировке модели. Структура также содержит методы считывания и изменения (записи) значений элементов. Обратите внимание на тип данных всех элементов структуры, содержащих значения параметров модели нейрона. Этот тип, `double_t` определён в одном из заголовочных файлов ядра (`nest/nestkernel/nest.h`) при помощи ключевого слова `typedef`.

```
…     …
197 private:
198
199     // --------------------------------------------------------------------
200
201     //! Independent parameters
202     struct Parameters_ {
203       double_t t_ref_;    //!< refractory time in ms
204       double_t g_Na;                  //!< Sodium Conductance in nS
205       double_t g_K;                   //!< Potassium Conductance in nS
206       double_t g_L;       //!< Leak Conductance in nS
207       double_t C_m;       //!< Membrane Capacitance in pF
208       double_t E_Na;                  //!< Sodium Reversal Potential in
              mV
209       double_t E_K;                   //!< Potassium Reversal Potential
              in mV
```

```
210       double_t E_L;       //!< Leak reversal Potential (aka resting po-
              tential) in mV
211       double_t tau_synE;  //!< Synaptic Time Constant Excitatory Synapse
              in ms
212       double_t tau_synI;  //!< Synaptic Time Constant for Inhibitory
              Synapse in ms
213       double_t I_e;       //!< Constant Current in pA
214
215       Parameters_();  //!< Sets default parameter values
216
217       void get(DictionaryDatum&) const;  //!< Store current values in
              dictionary
218       void set(const DictionaryDatum&);  //!< Set values from dicitonary
219     };
...   ...
```

Как видно из листинга кода, структура параметров модели имеет три встроенных метода. `Parameters_` служит для присвоения всем параметрам значений по умолчанию, `get` предназначен для считывания набора значений параметров и записи этого набора в специальный словарь, а метод `set` используется для изменения значений параметров, путём считывания их значений из такого же словаря. Рассмотрим как выглядят исходные коды этих методов, которые можно найти в файле `hh_psc_alpha.cpp`.

hh_psc_alpha.cpp

```
...   ...
126 nest::hh_psc_alpha::Parameters_::Parameters_()
127   : t_ref_   (    2.0  ),  // ms
128     g_Na     (12000.0  ),  // nS
129     g_K      ( 3600.0  ),  // nS
130     g_L      (   30.0  ),  // nS
131     C_m      (  100.0  ),  // pF
132     E_Na     (   50.0  ),  // mV
133     E_K      (  -77.0  ),  // mV
134     E_L      (  -54.402),  // mV
135     tau_synE(  0.2     ),  // ms
136     tau_synI(  2.0     ),  // ms
137     I_e      (  0.0     )   // pA
138 {
```

```
139 }
... ...
```

Функция инициализации параметров записывает значения по умолчанию во все элементы структуры с параметрами модели. Комментарии к каждой строчке с присвоением значения по умолчанию указывают на единицы измерения величин (миллисекунды, наносименсы, пикофарады, милливольты и пикоамперы), поскольку рассматриваемая модель нейрона является размерной.

hh_psc_alpha.cpp

```
... ...
182 void nest::hh_psc_alpha::Parameters_::get(DictionaryDatum &d) const
183 {
184   def<double>(d,names::t_ref, t_ref_);
185   def<double>(d,names::g_Na, g_Na);
186   def<double>(d,names::g_K, g_K);
187   def<double>(d,names::g_L, g_L);
188   def<double>(d,names::E_Na, E_Na);
189   def<double>(d,names::E_K, E_K);
190   def<double>(d,names::E_L, E_L);
191   def<double>(d,names::C_m, C_m);
192   def<double>(d,names::tau_syn_ex, tau_synE);
193   def<double>(d,names::tau_syn_in, tau_synI);
194   def<double>(d,names::I_e, I_e);
195 }
196
197 void nest::hh_psc_alpha::Parameters_::set(const DictionaryDatum& d)
198 {
199   updateValue<double>(d,names::t_ref, t_ref_);
200   updateValue<double>(d,names::C_m, C_m);
201   updateValue<double>(d,names::g_Na,g_Na);
202   updateValue<double>(d,names::E_Na,E_Na);
203   updateValue<double>(d,names::g_K, g_K);
204   updateValue<double>(d,names::E_K, E_K);
205   updateValue<double>(d,names::g_L, g_L);
206   updateValue<double>(d,names::E_L, E_L);
207
208   updateValue<double>(d,names::tau_syn_ex,tau_synE);
209   updateValue<double>(d,names::tau_syn_in,tau_synI);
210
```

```
211   updateValue<double>(d,names::I_e, I_e);
212
213   if ( C_m <= 0 )
214     throw BadProperty("Capacitance must be strictly positive.");
215
216   if ( t_ref_ < 0 )
217     throw BadProperty("Refractory time cannot be negative.");
218
219   if ( tau_synE <= 0 || tau_synI <= 0 )
220     throw BadProperty("All time constants must be strictly positive.");
221
222   if ( g_K < 0 || g_Na < 0 || g_L < 0 )
223     throw BadProperty("All conductances must be non-negative.");
224 }
...     ...
```

Методы считывания и записи включают работу с типом данных словарь, `DictionaryDatum`, определённом в модуле интерпретатора внутреннего скриптового языка нейросимулятора SLI (`nest/sli/dictdatum.h`). Функции `get` передаётся указатель на словарь, в который происходит запись значений параметров, считываемых из элементов соответствующей структуры. Функция `set` также принимает указатель на словарь, только теперь значения из этого словаря присваиваются переменным – элементам структуры `Parameters_`. Кроме того функцией set осуществляется проверка допустимых значений некоторых типов параметров при присвоении. Так, например, значения ёмкости мембраны нейрона, характерного рефрактерного времени, характерных времён динамики постсинаптических ответов, а также значения проводимостей не могут быть отрицательным в соответствии с физическим смыслом этих параметров (смотри сообщения, выдаваемые оператором `throw`).

Переменные модели задаются структурой `State_`, которая состоит из перечисления, вектора состояния, методов инициализации, считывания и изменения элементов структуры. Как указано в комментариях, нумерация элементов перечисления соответствует порядковому номеру элементов вектора состояния массива `State::y_`, используемого для интегрирования динамики модели численно с применением библиотеки GSL.

hh_psc_alpha.h

```
…     …
229 struct State_ {
230
231       /**
232        * Enumeration identifying elements in state array State_::y_.
233        * The state vector must be passed to GSL as a C array. This enum
234        * identifies the elements of the vector. It must be public to be
235        * accessible from the iteration function.
236        */
237       enum StateVecElems
238         {
239           V_M    = 0,
240           HH_M      ,  // 1
241           HH_H      ,  // 2
242           HH_N      ,  // 3
243           DI_EXC    ,  // 4
244           I_EXC     ,  // 5
245           DI_INH    ,  // 6
246           I_INH     ,  // 7
247           STATE_VEC_SIZE
248         };
249
250
251       double_t y_[STATE_VEC_SIZE];  //!< neuron state, must be C-array
    for GSL solver
252       int_t    r_;           //!< number of refractory steps remaining
253
254       State_(const Parameters_&);  //!< Default initialization
255       State_(const State_&);
256       State_& operator=(const State_&);
257
258       void get(DictionaryDatum&) const;
259       void set(const DictionaryDatum&);
260     };
…     …
```

Структура состояния переменных модели нейрона также содержит методы инициализации, копирования, а также методы записи и считывания значений через структуру данных `DictionaryDatum`. Подобно рассмотренной выше структуре

параметров эти методы реализованы в основном файле исходных кодов модуля `hh_psc_alpha.cpp`. Рассмотрим сначала методы инициализации и копирования. При инициализации переменных, описывающих состояние нейрона, требуется значение параметров. Поэтому в качестве аргумента конструктору структуры состояния передаётся указатель на структуру с параметрами. Далее происходит вычисление начальных значений переменных состояния по формулам, определяющим динамику модели. Вычисляется значение равновесного мембранного потенциала при текущих параметрах, вычисляются значения равновесных функций активации и деактивации в модели Ходжкина-Хаксли, полученные значения записываются в вектор переменных состояния модели. Конструкторы копирования циклом осуществляют копирование всех значений вектора переменных состояния модели.

hh_psc_alpha.cpp

```
…     …
141 nest::hh_psc_alpha::State_::State_(const Parameters_&)
142   : r_(0)
143 {
144   y_[0] = -65;//p.E_L;
145   for ( size_t i = 1 ; i < STATE_VEC_SIZE ; ++i )
146     y_[i] = 0;
147
148   // equilibrium values for (in)activation variables
149   const double_t alpha_n = (0.01 * (y_[0]+55.)) / (1. -std::exp( -
    (y_[0]+55.)/10.));
150   const double_t beta_n  = 0.125 * std::exp( -(y_[0]+65.)/80.);
151   const double_t alpha_m = (0.1 * (y_[0]+40.)) / (1. - std::exp( -
    (y_[0]+40.)/10.) );
152   const double_t beta_m  = 4. * std::exp( -(y_[0]+65.)/18.);
153   const double_t alpha_h = 0.07 * std::exp( -(y_[0]+65.) / 20.);
154   const double_t beta_h  = 1. / (1. + std::exp(-(y_[0]+35.) / 10. ));
155
156   y_[HH_H] = alpha_h/(alpha_h+beta_h);
157   y_[HH_N] = alpha_n/(alpha_n+beta_n);
158   y_[HH_M] = alpha_m/(alpha_m+beta_m);
159 }
160
161 nest::hh_psc_alpha::State_::State_(const State_& s)
```

```
162   : r_(s.r_)
163 {
164   for ( size_t i = 0 ; i < STATE_VEC_SIZE ; ++i )
165     y_[i] = s.y_[i];
166 }
167
168 nest::hh_psc_alpha::State_& nest::hh_psc_alpha::State_::operator=(const
    State_& s)
169 {
170   assert(this != &s);  // would be bad logical error in program
171
172   for ( size_t i = 0 ; i < STATE_VEC_SIZE ; ++i )
173     y_[i] = s.y_[i];
174   r_ = s.r_;
175   return *this;
176 }
...     ...
```

Методы чтения и записи значений переменных состояния аналогичны таким же метода структуры параметров. Их реализацию также можно найти в файле `hh_psc_alpha.cpp`.

hh_psc_alpha.cpp

```
...     ...
226 void nest::hh_psc_alpha::State_::get(DictionaryDatum &d) const
227 {
228   def<double>(d,names::V_m    , y_[V_M]);
229   def<double>(d,names::Act_m  , y_[HH_M]);
230   def<double>(d,names::Act_h  , y_[HH_H]);
231   def<double>(d,names::Inact_n, y_[HH_N]);
232 }
233
234 void nest::hh_psc_alpha::State_::set(const DictionaryDatum& d)
235 {
236   updateValue<double>(d,names::V_m    , y_[V_M]);
237   updateValue<double>(d,names::Act_m  , y_[HH_M]);
238   updateValue<double>(d,names::Act_h  , y_[HH_H]);
239   updateValue<double>(d,names::Inact_n, y_[HH_N]);
240
241   if ( y_[HH_M] < 0 || y_[HH_H] < 0 || y_[HH_N] < 0 )
      throw BadProperty("All (in)activation variables must be non-
```

```
242 negative.");
243 }
... ...
```

После структуры состояния `State_` в классе `hh_psc_alpha` следует структура буферов модели, `Buffers_`, (строчки 268-301), содержащая вспомогательные элементы для записи значений переменных при численном интегрировании, обработке входящих событий запроса данных, спайков и токов, а также собственные методы инициализации.

hh_psc_alpha.h

```
... ...
268 struct Buffers_ {
269       Buffers_(hh_psc_alpha&);                    //!<Sets buffer point-
    ers to 0
270       Buffers_(const Buffers_&, hh_psc_alpha&);  //!<Sets buffer point-
    ers to 0
271
272       //! Logger for all analog data
273       UniversalDataLogger<hh_psc_alpha> logger_;
274
275       /** buffers and sums up incoming spikes/currents */
276       RingBuffer spike_exc_;
277       RingBuffer spike_inh_;
278       RingBuffer currents_;
279
280       /** GSL ODE stuff */
281       gsl_odeiv_step*    s_;    //!< stepping function
282       gsl_odeiv_control* c_;    //!< adaptive stepsize control function
283       gsl_odeiv_evolve*  e_;    //!< evolution function
284       gsl_odeiv_system   sys_;  //!< struct describing system
285
286       // IntergrationStep_ should be reset with the neuron on
    ResetNetwork,
287       // but remain unchanged during calibration. Since it is initial-
    ized with
288       // step_, and the resolution cannot change after nodes have been
    created,
289       // it is safe to place both here.
290       double_t step_;           //!< step size in ms
```

```
291       double   IntegrationStep_;//!< current integration time step, up-
          dated by GSL
292
293       /**
294        * Input current injected by CurrentEvent.
295        * This variable is used to transport the current applied into the
296        * _dynamics function computing the derivative of the state vec-
          tor.
297        * It must be a part of Buffers_, since it is initialized once be-
          fore
298        * the first simulation, but not modified before later Simulate
          calls.
299        */
300       double_t I_stim_;
301     };
...     ...
```

Конструкторы инициализации и копирования структуры буферов также реализованы в файле `hh_psc_alpha.cpp`.

hh_psc_alpha.cpp

```
...     ...
245 nest::hh_psc_alpha::Buffers_::Buffers_(hh_psc_alpha& n)
246   : logger_(n),
247     s_(0),
248     c_(0),
249     e_(0)
250 {
251     // Initialization of the remaining members is deferred to
252     // init_buffers_().
253 }
254
255 nest::hh_psc_alpha::Buffers_::Buffers_(const Buffers_&, hh_psc_alpha& n)
256   : logger_(n),
257     s_(0),
258     c_(0),
259     e_(0)
260 {
261     // Initialization of the remaining members is deferred to
262     // init_buffers_().
263 }
```

```
...    ...
```

Данные функции инициализируют значения по умолчанию для элементов структуры буферов, относящихся к решению дифференциальных уравнений с использованием научной библиотеки GSL. Как сказано в комментариях, инициализация остальных элементов структуры буферов перенесена в отдельную функцию инициализации `init_buffers_()`. Данная функция, также реализованная в основном файле с исходными кодами модуля, осуществляет инициализацию элементов структуры буферов модуля. Первым делом сбрасываются счётчики количества спайков, а также входящих спайковых и токовых событий.

hh_psc_alpha.cpp

```
...    ...
304  void nest::hh_psc_alpha::init_buffers_()
305  {
306    B_.spike_exc_.clear();       // includes resize
307    B_.spike_inh_.clear();       // includes resize
308    B_.currents_.clear();        // includes resize
309    Archiving_Node::clear_history();
310
311    B_.logger_.reset();
312
313    B_.step_ = Time::get_resolution().get_ms();
314    B_.IntegrationStep_ = B_.step_;
315
316    static const gsl_odeiv_step_type* T1 = gsl_odeiv_step_rkf45;
317
318    if ( B_.s_ == 0 )
319      B_.s_ = gsl_odeiv_step_alloc (T1, State_::STATE_VEC_SIZE);
320    else
321      gsl_odeiv_step_reset(B_.s_);
322
323    if ( B_.c_ == 0 )
324      B_.c_ = gsl_odeiv_control_y_new (1e-3, 0.0);
325    else
326      gsl_odeiv_control_init(B_.c_, 1e-3, 0.0, 1.0, 0.0);
327
```

```
328   if ( B_.e_ == 0 )
329     B_.e_ = gsl_odeiv_evolve_alloc(State_::STATE_VEC_SIZE);
330   else
331     gsl_odeiv_evolve_reset(B_.e_);
332
333   B_.sys_.function  = hh_psc_alpha_dynamics;
334   B_.sys_.jacobian  = NULL;
335   B_.sys_.dimension = State_::STATE_VEC_SIZE;
336   B_.sys_.params    = reinterpret_cast<void*>(this);
337
338   B_.I_stim_ = 0.0;
339 }
...     ...
```

В теле функции инициализации буферов присутствует объявление константы, определяющей метод интегрирования, из набора методов научной библиотеки GSL (строчка 316). К этому мы вернёмся в разделе «Приложение 2. Написание нового модуля» при рассмотрении процесса написания нового модуля на основе небольшой модификации данного модуля.

Кроме конструкторов инициализации и копирования в структуре буферов присутствует объявление логгера, тип данных которого также является частью ядра нейросимулятора NEST и описан в файле `nest/nestkernel/universal_data_logger.h`. Данный класс представляет собой универсальный плагин для моделей нейронов, предназначенный для записи данных произвольной природы, как то мембранный потенциал, проводимость. Способ записи, обеспечиваемый данным классом, совместим с событиями `DataLoggingRequest` и `DataLoggingReply`.

Следом за структурой буферов идёт структура внутренних переменных модели, `Variables_`, (строчки 308-316).

hh_psc_alpha.h

```
...     ...
308 struct Variables_ {
309       /** initial value to normalise excitatory synaptic current */
310       double_t PSCurrInit_E_;
311
```

```
312         /** initial value to normalise inhibitory synaptic current */
313         double_t PSCurrInit_I_;
314
315         int_t    RefractoryCounts_;
316       };
...     ...
```

Завершают класс `hh_psc_alpha` объявления переменных описанных структур данных, функции доступа к данным и отображение списка имён записываемых переменных (строчка 332). Последнее, вероятно, требует пояснения. В процессе симуляции сами нейроны и синапсы в NEST не записывают эволюцию своих переменных. Для того, чтобы вывести рассчитанные численно осциллограммы мембранного потенциала, других переменных модели, а также токов, проводимостей, и других важных значений необходимо к узлу модели нейрона подключать специальные узлы записывающих устройств. Эти записывающие устройства также являются обычными модулями нейросимулятора NEST. Одним из таких записывающих устройств является мультиметр (имя модуля `‘multimeter’`). Однако не каждая переменная модели может быть записана при помощи мультиметра. Чтобы подключённый мультиметр имел возможность записывать динамику переменной в процессе симуляции необходимо эту переменную включить в список записываемых. Для этого и служит отображение списка имён записываемых переменных `RecordablesMap`.

hh_psc_alpha.h

```
...     ...
320     //! Read out state vector elements, used by UniversalDataLogger
321     template <State_::StateVecElems elem>
322     double_t get_y_elem_() const { return S_.y_[elem]; }
323
324     // ------------------------------------------------------------------
325
326     Parameters_ P_;
327     State_      S_;
328     Variables_  V_;
329     Buffers_    B_;
330
```

```
331     //! Mapping of recordables names to access functions
332     static RecordablesMap<hh_psc_alpha> recordablesMap_;
333   };
…     …
```

Далее в заголовочном файле модуля hh_psc_alpha находятся сами тела некоторых методов-функций класса `hh_psc_alpha`, выделенных ключевыми словами `inline`.

Файл `hh_psc_alpha.cpp` содержит тела функций класса `hh_psc_alpha`, объявления которых описаны в соответствующем заголовочном файле и были рассмотрены выше. Кроме того параллельно были рассмотрены реализации некоторых методов, расположенные в файле `hh_psc_alpha.cpp`. Рассмотрим теперь реализации остальных методов класса. Итак, основные члены класса модели нейрона сосредоточены в четырёх структурах: структуре параметров, структуре состояния, структуре внутренних переменных и структуре буферов – `P_`, `S_`, `V_` и `B_`, соответственно. Конструктор по умолчанию и конструктор копирования выполняют инициализацию всех этих структур.

hh_psc_alpha.cpp

```
…   …
265 /* ----------------------------------------------------------------
266  * Default and copy constructor for node, and destructor
267  * ---------------------------------------------------------------- */
268
269 nest::hh_psc_alpha::hh_psc_alpha()
270   : Archiving_Node(),
271     P_(),
272     S_(P_),
273     B_(*this)
274 {
275   recordablesMap_.create();
276 }
277
278 nest::hh_psc_alpha::hh_psc_alpha(const hh_psc_alpha& n)
279   : Archiving_Node(n),
280     P_(n.P_),
281     S_(n.S_),
```

```
282     B_(n.B_, *this)
283 {
284 }
285
286 nest::hh_psc_alpha::~hh_psc_alpha()
287 {
288   // GSL structs may not have been allocated, so we need to protect de-
289 struction
290   if ( B_.s_ ) gsl_odeiv_step_free(B_.s_);
291   if ( B_.c_ ) gsl_odeiv_control_free(B_.c_);
292   if ( B_.e_ ) gsl_odeiv_evolve_free(B_.e_);
293 }
... ...
```

Следует особо отметить основную функцию, реализующую динамику модели. Эта функция является наиболее уникальной у каждой модели и содержит собственно её математическое описание, написанное на понятном для компилятора языке.

hh_psc_alpha.cpp

```
...     ...
67 extern "C"
68   int hh_psc_alpha_dynamics (double, const double y[], double f[], void*
     pnode)
69   {
70     // a shorthand
71     typedef nest::hh_psc_alpha::State_ S;
72
73     // get access to node so we can almost work as in a member function
74     assert(pnode);
75     const nest::hh_psc_alpha& node =
     *(reinterpret_cast<nest::hh_psc_alpha*>(pnode));
76
77     // y[] here is---and must be---the state vector supplied by the in-
     tegrator,
78     // not the state vector in the node, node.S_.y[].
79
80     // The following code is verbose for the sake of clarity. We assume
     that a
81     // good compiler will optimize the verbosity away ...
```

```
82
83    // shorthand for state variables
84    const double_t& V     = y[S::V_M   ];
85    const double_t& m     = y[S::HH_M  ];
86    const double_t& h     = y[S::HH_H  ];
87    const double_t& n     = y[S::HH_N  ];
88    const double_t& dI_ex = y[S::DI_EXC];
89    const double_t&  I_ex = y[S::I_EXC ];
90    const double_t& dI_in = y[S::DI_INH];
91    const double_t&  I_in = y[S::I_INH ];
92
93    const double_t alpha_n = (0.01 * (V+55.)) / (1. -std::exp( -
(V+55.)/10.));
94    const double_t beta_n  = 0.125 * std::exp( -(V+65.)/80.);
95    const double_t alpha_m = (0.1 * (V+40.)) / (1. - std::exp( -
(V+40.)/10.) );
96    const double_t beta_m  = 4. * std::exp( -(V+65.)/18.);
97    const double_t alpha_h = 0.07 * std::exp( -(V+65.) / 20.);
98    const double_t beta_h  = 1. / (1. + std::exp(-(V+35.) / 10. ));
99
100   const double_t I_Na =  node.P_.g_Na * m * m * m * h * (V -
node.P_.E_Na);
101   const double_t I_K  =  node.P_.g_K  * n * n * n * n * (V -
node.P_.E_K );
102   const double_t I_L  =  node.P_.g_L                  * (V -
node.P_.E_L );
103
104   // V dot -- synaptic input are currents, inhib current is negative
105   f[S::V_M] = ( -(I_Na + I_K + I_L) + node.B_.I_stim_ + node.P_.I_e +
I_ex + I_in) / node.P_.C_m;
106
107   //channel dynamics
108   f[S::HH_M] = alpha_m * (1-y[S::HH_M]) - beta_m * y[S::HH_M];      //
m-variable
109   f[S::HH_H] = alpha_h * (1-y[S::HH_H]) - beta_h * y[S::HH_H]; // h-
variable
110   f[S::HH_N] = alpha_n * (1-y[S::HH_N]) - beta_n * y[S::HH_N];      //
n-variable
111
112   // synapses: alpha functions
113   f[S::DI_EXC] = -dI_ex / node.P_.tau_synE;
114   f[S::I_EXC ] =  dI_ex  - (I_ex / node.P_.tau_synE);
115   f[S::DI_INH] = -dI_in / node.P_.tau_synI;
```

```
116     f[S::I_INH ] =  dI_in  - (I_in / node.P_.tau_synI);
117
118     return GSL_SUCCESS;
119   }
...     ...
```

ПРИЛОЖЕНИЯ

Приложение 1. Установка симулятора NEST на ОС семейства Linux

Операционная система Ubuntu является одним из наиболее популярных и известных представителей операционных систем семейства Linux. В данном разделе приведены инструкции по установке симулятора NEST на операционную систему Kubuntu, которая отличается от Ubuntu набором библиотек для внешней оболочки (среды рабочего стола). Ubuntu использует среду рабочего стола Unity (в старых версиях Gnome), в то время как Kubuntu использует KDE. Установка нейросимулятора NEST на другие операционные системы семейства Linux осуществляется аналогичным образом, хотя для работы с другими операционными системами некоторые конкретные команды и шаги, приведённые в данном пособии, могут отличаться.

Наиболее простым вариантом установки NEST является использование образа операционной системы готовой сборки с предустановленным нейросимулятором, что называется, «из коробки». Данный образ предоставляется сообществом, разрабатывающим NEST, - nest-initiative (http:// nest-initiative.org) и доступен на официальном сайте на странице для скачивания по ссылке: http://nest-initiative.org/Software:Download. В данном пособии мы не будем рассматривать этот вариант, а остановимся на наиболее популярном и общепринятом способе установки нейросимулятора на собственную операционную систему. Для этого нейросимулятор собирается из исходных кодов. Несмотря на кажущуюся дл новичка сложность этого процесса сборка программ из исходных кодов является достаточно простой и общепринятой практикой для пользователей операционных систем семейства Linux. Что касается сборки NEST, данный способ имеет ряд преимуществ, как то гибкость настройки (можно изменять исходные коды), расширяемость, возможность выбора директории с установленными файлами, отсутствие необходимости обладать правами суперпользователя, возможность установки нескольких версий нейросимулятора и многие другие. Не стоит

забывать и о повышении своей квалификации – просматривая исходные коды, у нас появляется уникальная возможность перенять навыки и стиль программирования у высококлассных специалистов мирового уровня, разрабатывающих сложнейший и популярнейший открытый проект, история которого насчитывает не первый десяток лет. Кроме того, навык установки программного обеспечения посредством сборки из исходных кодов, является важным и полезным. Итак, приступим.

Для начала нужно установить операционную систему.

Для работы NEST требуется установка дополнительных пакетов. Все операции по установке будут производиться через консоль терминала. Для этого следует открыть терминал `Ctrl + Alt + t` (в операционной системе Ubuntu) Если вы используете операционную систему Kubuntu для открытия терминала необходимо нажать `Alt + F2` и в открывшейся сверху экрана выдвигающейся строке ввода напечатать `konsole`[1] и нажать клавишу `Enter`. Откроется терминал с приглашением ввода команд. Для установки необходимых для использования нейросимулятора NEST пакетов введите

```
sudo apt-get install build-essential autoconf automake libtool libltdl7-dev libreadline-gplv2-dev libncurses5-dev libgsl0-dev python-all-dev python-numpy python-scipy python-matplotlib ipython
```

Здесь и далее после ввода любой команды в терминал для её выполнения необходимо нажимать клавишу `Enter`. Процесс установки может занять несколько минут, а перед его началом возможно будет запрошено подтверждение на установку. Кроме того, при использовании утилиты `sudo` нужно ввести пароль суперпользователя. Если Вы не обладаете правами суперпользователя на этом компьютере, обратитесь за помощью, сообщив список требуемых вам пакетов

[1] Хотя в английском языке слово консоль пишется через букву “c”, здесь нет ошибки. Дело в том, что названия многих приложений среды рабочего стола KDE (а именно эту среду использует операционная система Kubuntu) начинаются на букву “k”, символизируя, что они являются частью среды рабочего стола KDE.

(`build-essential autoconf automake libtool libltdl7-dev libreadline-gplv2-dev libncurses5-dev libgsl0-dev python-all-dev python-numpy python-scipy python-matplotlib ipython`).

Далее нужно скачать архив с исходными кодами NEST. Он находится на странице для скачивания на официальном сайте проекта nest-initiative: http://nest-initiative.org/Software:Download

Заметим, что на момент написания данного методического пособия текущая версия NEST — 2.4.2. Следовательно, если Вы устанавливаете другую версию, все команды, содержащие версию 2.4.2, должны быть заменены на соответствующую версию.

Поместите скачанный архив в директорию, в которой вам будет удобно компилировать программу, например, `~/soft`[2]. Следуя дальнейшим указаниям для установки NEST, Вы можете использовать любую другую директорию.

```
mkdir ~/soft
mkdir ~/soft/nest
cp ~/downloads/nest-2.4.2.tar.gz ~/soft/nest
```

Последняя команда предполагает, что ваш архив был загружен в каталог `~/downloads`. В случае другого каталога загрузок впишите соответствующий путь к директории в последнюю команду. Для ускорения работы с командной строкой Linux существуют различные комбинации клавиш. Например, последнюю команду можно было не набирать полностью, если после `~/downloads/nest-2.4.1.tar.gz` набрать пробел и нажать `Alt + .` (удерживать клавишу Alt и нажать точку[3]). Данное действие вызовет автоподстановку последнего аргумента последней набранной команды. В данном случае это путь `~/soft/nest`. Далее следует распаковка

[2] Тильдой, «~», в пути к каталогу или файлу в операционных системах семейства Linux обозначается домашняя директория. Обычно это /home/<username>.

[3] Имеется в виду точка в английской раскладке, которая соответствует букве «ю» в русской раскладке.

```
cd ~/soft/nest
tar -xzvf nest-2.4.2.tar.gz
```

Перед сборкой исходных кодов необходимо произвести конфигурацию сборщика. Это следует делать из другой директории, куда собственно и будет производиться сборка. Создадим каталог для сборки

```
cd ..
mkdir nest-2.4.2_build
```

Переместимся в только что созданный каталог. Как и ранее здесь полного набора команды можно избежать, используя автоподстановку `Alt + .`.

```
cd nest-2.4.2_build
```

Перед конфигурацией можно прописать переменную окружения, содержащую путь к установочным файлам NEST.

```
export NEST_INSTALL_DIR=~/work/nest-2.4.2
```

Проверим, что присвоили переменной окружения значение нужной директории

```
echo $NEST_INSTALL_DIR
```

И конфигурируем, указывая в качестве аргумента префикс директории, в которую будет установлен NEST.

```
../nest-2.4.2/configure --prefix=$NEST_INSTALL_DIR
```

В процессе работы конфигуратора будет выведено множество непонятных строк в консоль терминала, после чего в консоль выведется сводка всей информации о результатах конфигурирования. Если всё было сделано правильно, результат должен выглядеть примерно так:

```
   ------------------------------------------------------------------------
--------
   NEST Configuration Summary
   ------------------------------------------------------------------------
--------

   C compiler          : gcc
   C compiler flags     : -W -Wall -pedantic -Wno-long-long -O2 -g -O2 -
fopenmp
   C++ compiler        : g++
   C++ compiler flags  : -W -Wall -pedantic -Wno-long-long -O2 -fopenmp

   Python bindings     : Yes (Python 2.7: /usr/bin/python)
   User modules        : None
   Extra modules       : models precise topology
   Dynamic modules     : None

   Use threading       : Yes (OpenMP)

   Use GSL             : Yes
   Use MPI             : No
   Use MUSIC           : No
   Use libneurosim     : No

   ------------------------------------------------------------------------
--------

   The NEST executable will be installed to:
     /home/******/work/nest-2.4.2/bin/
```

```
   Documentation and examples will be installed to:
     /home/******/work/nest-2.4.2/share/doc/nest/

   PyNEST will be installed to:
     /home/******/work/nest-2.4.2/lib/python2.7/site-packages

   ------------------------------------------------------------------------
--------

   You can now build and install NEST with
     make
     make install
     make installcheck

   If you experience problems with the installation or the use of
   NEST, please see http://nest-initiative.org/index.php/FAQ, or
   go to http://nest-initiative.org/index.php/Community to find out
   how to join the user mailing list.
```

Ознакомившись с результатом работы конфигуратора можно приступать непосредственно к компиляции NEST. Делается это простой командой

```
   make
```

Процесс компиляции NEST может занять от нескольких минут до часа в зависимости от производительности используемого компьютера. Отметим, что использование виртуальной машины с установленной операционной системой семейства Linux из-под другой операционной системы (например, из-под Windows) существенно замедлит работу компилятора. Поэтому для краткого ознакомления с нейросимулятором NEST рекомендуется использовать готовый образ операционной системы, предоставляемый сообществом разработчиков и содержащий предустановленный нейросимулятор. Данный образ может быть

запущен также и из-под виртуальной машины. Однако для работы с NEST всё-таки рекомендуется поставить свою операционную систему семейства Linux параллельно с Вашей операционной системой и собирать NEST из исходных кодов, как описано в данном пособии.

По окончании процесса компиляции терминал снова предложит Вам ввести команду, выводя стандартное приглашение. Если перед приглашением вывод компилятора не содержал никаких ошибок, значит компиляция прошла успешно, и можно приступать к установке. Для этого введите команду

```
make install
```

или в случае, если устанавливаете NEST за пределы своего домашнего каталога

```
sudo make install
```

В последнем случае потребуется ввести пароль суперадминистратора. Процесс инсталляции занимает немного времени, а о его завершении Вы узнаете, увидев приглашение для ввода команды в консоль, а перед этим что-то вроде этого

```
make[4]: Выход из каталога `/home/*******/soft/nest/nest-2.4.2_build/testsuite'
sed -e "s:++PKGDATADIR++:/home/*******/work/nest-2.4.2/share/nest:"\
            -e "s:++PKGDOCDIR++:/home/*******/work/nest-2.4.2/share/doc/nest:"\
            -e "s:++PKGSRCDIR++:/home/*******/soft/nest/nest-2.4.2:"\
            /home/*******/work/nest-2.4.2/share/nest/extras/emacs/sli.el.in > /home/*******/work/nest-2.4.2/share/nest/extras/emacs/sli.el
rm /home/*******/work/nest-2.4.2/share/nest/extras/emacs/sli.el.in
/usr/bin/install -c /home/*******/soft/nest/nest-2.4.2_build/extras/nest-config /home/*******/work/nest-2.4.2/bin/
```

```
   /usr/bin/install            -c            /home/*******/soft/nest/nest-
2.4.2/extras/nest_serial /home/*******/work/nest-2.4.2/bin/
   /usr/bin/install            -c            /home/*******/soft/nest/nest-
2.4.2/extras/nest_indirect /home/*******/work/nest-2.4.2/bin/
   /usr/bin/install  -c  -m  644  /home/*******/soft/nest/nest-2.4.2/README
/home/*******/work/nest-2.4.2/share/doc/nest/
   /usr/bin/install  -c  -m  644  /home/*******/soft/nest/nest-2.4.2/NEWS
/home/*******/work/nest-2.4.2/share/doc/nest/
   make[3]: Выход из каталога `/home/*******/soft/nest/nest-2.4.2_build'
   make[2]: Выход из каталога `/home/*******/soft/nest/nest-2.4.2_build'
   make[1]: Выход из каталога `/home/*******/soft/nest/nest-2.4.2_build'
```

Теперь всё. Установка прошла успешно. Однако напоследок, перед тем как начать пользоваться нейросимулятором, необходимо проверить правильность сборки. В состав нейросимулятора NEST входят утилиты самотестирования, запуск которых осуществляется командой

```
   make installcheck
```

или если при установке использовалась утилита `sudo`, её нужно использовать и при проверке

```
   sudo make installcheck
```

Процесс проверки занимает несколько минут, в течение которых можно наблюдать за ходом тестирования. При этом в консоль выводятся строчки с названиями фаз тестирования и тестов, как выполняющихся в настоящий момент, так и уже выполненных тестов с их соответствующими результатами.

```
   Phase 1: Testing if SLI can execute scripts and report errors
   ------------------------------------------------------------
     Running test 'selftests/test_pass.sli'... Success
```

```
    Running test 'selftests/test_goodhandler.sli'... Success
    Running test 'selftests/test_lazyhandler.sli'... Success
    Running test 'selftests/test_fail.sli'... Success
    Running test 'selftests/test_stop.sli'... Success
    Running test 'selftests/test_badhandler.sli'... Success

  Phase 2: Testing SLI's unittest library
  ---------------------------------------
    Running test 'selftests/test_pass_or_die.sli'... Success
    Running test 'selftests/test_assert_or_die_b.sli'... Success
    Running test 'selftests/test_assert_or_die_p.sli'... Success
    Running test 'selftests/test_fail_or_die.sli'... Success
    Running test 'selftests/test_crash_or_die.sli'... Success

  Phase 3: Running NEST unit tests
  --------------------------------
    Running    test    'unittests/test_aeif_cond_alpha_multisynapse.sli'...
Success
    Running test 'unittests/test_binary.sli'... Success
    Running test 'unittests/test_connect.sli'... Success
    Running test 'unittests/test_convergent_connect.sli'... Success
    Running test 'unittests/test_copymodel.sli'... Success
    Running test 'unittests/test_corr_det.sli'... Success
    Running test 'unittests/test_corr_matrix_det.sli'... Success
    Running test 'unittests/test_cva.sli'... Success
    Running test 'unittests/test_cvi.sli'... Success
  …
```

В конце выводится результирующая информация о всех проведённых тестах с указанием ошибок, возникших в ходе проверки, а также с указанием пути, по которому можно найти файл с детальным отчётом о проведённом тестировании.

```
    PyNEST tests: 199
       Passed: 199
```

```
      Failed: 0

NEST Testsuite Summary
----------------------
  NEST Executable: /home/******/work/nest-2.4.2/bin/nest
  SLI Executable : /home/******/work/nest-2.4.2/bin/sli
  Total number of tests: 373
     Passed: 372
     Failed: 1 (0 PyNEST)

***
*** There were errors detected during the run of the NEST test suite!
***
*** Please send the archived content of these directories:
***
***     - '/home/******/soft/nest/nest-2.4.2_build/reports'
***     - '/home/******/soft/nest/nest-2.4.2_build/reports/nest.UHy8E'
***
*** to the nest_user@nest-initiative.org mailing list.
***

make[1]: Выход из каталога `/home/******/soft/nest/nest-2.4.2_build'
```

Напоследок важно прописать переменную окружения PYTHONPATH для интерпретатора python. Это нужно для того, чтобы интерпретатор мог найти установленный модуль PyNEST в числе прочих своих модулей. Делается это следующим образом. Для начала проверим значение переменной окружения PYTHONPATH:

```
echo $PYTHONPATH
```

Ответом терминала на введённую команду, вероятнее всего, окажется пустая строка, либо строка, содержащая один или несколько разделенных двоеточиями

адресов каталогов, где интерпретатор python будет искать установленные модули, при попытке их импортировать командой `import`.

Для добавления нового адреса в переменную окружения выполните команду

```
export PYTHONPATH=$PYTHONPATH:$NEST_INSTALL_DIR/lib/python2.7/site-packages
```

Убедитесь, что версия интерпретатора python задана правильно.

На этом установку NEST можно считать завершённой. Теперь можно переходить к использованию нейросимулятора. Основные команды и способ запуска NEST приведены в разделе Введения. Более детальную информацию о применении нейросимулятора можно найти на официальном сайте сообщества nest-initiative в разделе документации (http://nest-initiative.org/Software:Documentation).

Приложение 2. Написание нового модуля

Нейросимулятор NEST содержит в составе большое число реализованных моделей нейронов и синапсов для использования при построении больших моделей спайковых нейронных сетей. Однако зачастую стандартных моделей оказывается недостаточно, и для определённой задачи нужно использовать модель нейрона или синапса, отсутствующую в списке. Для этого необходимо написать новый модуль нейросимулятора, учитывая все особенности устройства NEST, принципы его организации, взаимодействия между компонентами и др. Рассмотрим пример написания нового модуля для нейросимулятора NEST. Стоит сказать, что написание собственного модуля «с нуля» достаточно сложный и трудоёмкий процесс. Для начала мы приведём пример простейшей модификации одного из существующих стандартных модулей, а именно, модуля `hh_pcs_alpha`, исходный код которого был подробно рассмотрен в пункте “3.2. Описание модуля модели нейрона” данного пособия. Суть модификации заключается в изменении способа интегрирования дифференциальных уравнений модели. Указанная модель использует алгоритм интегрирования Дорманда-Принца, являющегося надстройкой над методом Рунге-Кутты 4-5 порядка. Мы изменим используемый алгоритм на простой способ интегрирования Рунге-Кутты 4-5 порядка. Как было показано в разделе с обзором исходных кодов модуля нейрона Ходжкина-Хаксли, интегрирование уравнений, описывающих его динамику, осуществляется с использованием научной библиотеки GSL.

Использование библиотеки GSL в нейросимуляторе NEST является одной из причин, почему NEST работает только под семейством операционных систем Linux. Другая причина заключается в использовании потоков POSIX для организации параллельных вычислений.

Один из компонентов научной библиотеки GSL предназначен для решения систем обыкновенных дифференциальных уравнений. Данный компонент содержит множество реализованных алгоритмов интегрирования. Просматривая исходные модуля hh_psc_alpha, коды было отмечено, что данный модуль использует алгоритм интегрирования Дорманда-Принца. Выбор алгоритма

осуществляется в функции инициализации структуры буферов модели `init_buffers_()` (строчка316 в файле `hh_psc_alpha.cpp`)

```
static const gsl_odeiv_step_type* T1 = gsl_odeiv_step_rkf45;
```

Как было показано при обзоре исходных кодов модуля нейросимулятора, модули обладают схожей структурой. Они содержат одинаковые структуры данных и методы для инициализации, записи и чтения значений параметров и переменных, обработки входящих событий разлисного типа и пр. Основные отличия в модулях, реализующих различные модели, заключаются в функции, определяющей динамику самой модели. Поэтому для создания новых модулей удобно за основу брать один из существующих стандартных модулей. Возьмём модуль hh_psc_alpha, состоящий из двух файлов, и скопируем эти файлы с другими именами - `hh_psc_alpha_rk.h` и `hh_psc_alpha_rk.cpp`. Далее в новом файле заменим строчку 316 на строчку

```
static const gsl_odeiv_step_type* T1 = gsl_odeiv_step_rk;
```

Модуль с другим алгоритмом численного интегрирования уравнений готов. Если мы хотим изменить сами уравнения, необходимо модифицировать функцию определяющую динамику модели. Кроме того, некоторые функции инициализации также зависят от самих уравнений. Соответственно, их также необходимо изменить, если изменения затронули эти уравнения.

Здесь мы приведём пример составления нового модуля из двух уже существующих стандартных модулей. Новый модуль будет обладать особенностями обоих модулей и учитывать динамику обеих моделей. Первая модель имитирует динамику кратковременной синаптической пластичности [27], вторая – долговременной STDP пластичности [28]. Новый модуль в этом случае лучше наследовать от модуля с STDP пластичностью в виду его более сложной динамики. Создадим копии файлов модуля с STDP пластичностью

`stdp_synapse.h` и `stdp_synapse.cpp` и назовём их, соответственно, `stdp_tsodyks2_synapse.h` и `stdp_tsodyks2_synapse.cpp`. В заголовочном файле нужно заменить имена класса, конструкторов и деструктора, а также все вхождения имени класса на `STDPTsodyks2Connection`. Все закрытые члены класса должны быть объединены. Для этого нужно скопировать строки с объявлениями членов класса из заголовочного файла модуля с кратковременной пластичностью в заголовочный файл нового модуля, исключая дубликаты. Если в новый модуль были добавлены новые методы, их реализацию также нужно скопировать в соответствующие места заголовочного файла после ключевого слова `inline` или в файл с исходным кодом `*.cpp`. Все методы, оперирующие новыми данными, скопированными из модуля с кратковременной пластичностью, должны быть также обновлены. Для этого в их тела нужно добавить строки из соответствующих методов модуля кратковременной пластичности и, при необходимости, обновить аргументы этих функций. Наибольшие изменения коснутся метода вычисления мгновенного веса переданного через синапс спайка. Этот метод и содержит реализацию моделей синаптической пластичности. Приведём здесь листинг кода готового варианта метода send.

stdp_tsodyks2_synapse.cpp

```
…  …
341  void nest::STDPTsodyks2Connection::send(nest::Event& e, nest::double_t
     t_lastspike, const nest::CommonSynapseProperties &)
342  {
343
344    ////////////////////////////// STDP
     ////////////////////////////////////////
345
346    // synapse STDP depressing/facilitation dynamics
347
348    nest::double_t t_spike = e.get_stamp().get_ms();
349    // t_lastspike_ = 0 initially
350    nest::double_t dendritic_delay =
     nest::Time(nest::Time::step(dendritic_delay_)).get_ms();
351    nest::double_t axonal_delay =
     nest::Time(nest::Time::step(axonal_delay_)).get_ms();
```

```
352
353   //get spike history in relevant range (t1, t2] from post-synaptic neu-
      ron
354   std::deque<nest::histentry>::iterator start;
355   std::deque<nest::histentry>::iterator finish;
356
357   // For a new synapse, t_lastspike contains the point in time of the
      last spike.
358   // So we initially read the history(t_last_spike - dendritic_delay,
      ...,  T_spike-dendritic_delay]
359   // which increases the access counter for these entries.
360   // At registration, all entries' access counters of history[0, ...,
      t_last_spike - dendritic_delay] have been
361   // incremented by Archiving_Node::register_stdp_connection(). See bug
      #218 for details.
362   target_->get_history(t_lastspike + axonal_delay - dendritic_delay,
      t_spike + axonal_delay - dendritic_delay,
363                        &start, &finish);
364   //facilitation due to post-synaptic spikes since last pre-synaptic
      spike
365   nest::double_t minus_dt;
366   while (start != finish)
367   {
368     minus_dt = (t_lastspike  + axonal_delay) - (start->t_ + dendrit-
      ic_delay);
369     start++;
370     if (minus_dt == 0)
371       continue;
372     weight_ = facilitate_(weight_, Kplus_ * std::exp(minus_dt /
      tau_plus_));
373   }
374
375   //depression due to new pre-synaptic spike
376   weight_ = depress_(weight_, target_->get_K_value(t_spike + axon-
      al_delay - dendritic_delay));
377
378   ////////////////////////////// Tsodyks2
      ////////////////////////////////////////
379
380   nest::double_t h = e.get_stamp().get_ms() - t_lastspike;
381   nest::double_t f = std::exp(-h/tau_rec_);
382   nest::double_t u_decay = (tau_fac_ < 1.0e-10) ? 0.0 : std::exp(-
      h/tau_fac_);
```

```
383
384   x_= x_*(1.0-u_)*f + u_*(1.0-f); // Eq. 2 from reference [1]
385   u_ *= u_decay;
386   u_+= U_*(1.0-u_); // for tau_fac=0 and u_=0, this will render u_==U_
387
388   // send the spike to the target
389   e.set_receiver(*target_);
390   e.set_weight( x_*weight_ );
391   e.set_delay( delay_ );
392   e.set_rport( rport_ );
393   e();
394
395
396   Kplus_ = Kplus_ * std::exp((t_lastspike - t_spike) / tau_plus_) + 1.0;
397 }
... ...
```

Как видно из комментариев, функция разделена на две части, каждая из которых отвечает за динамику отдельной модели. Динамика модели долговременной пластичности реализована после строчки 344, а кратковременной – после строчки 378. Строчки 389 – 393 обеспечивают вызов стандартных методов класса события для отправки спайкового события на постсинаптический нейрон с соответствующей информацией о времени спайка, эффективном вычисленном весе, номере порта соединения и задержке.

В завершение, остаётся сформировать блок комментариев для справочной информации по новому модулю и можно приступать к установке.

Приложение 3. Установка с добавленным модулем

Как уже было сказано в Главе 3 данного пособия, добавление пользовательского модуля в нейросимулятор NEST может быть осуществлено как минимум двумя способами. Один из способов предполагает использование интерфейса для добавления новых моделей в NEST (он находится в каталоге с исходными кодами NEST `nest/examples/MyModule`). Этот способ не предполагает изменение исходных кодов нейросимулятора и подробно описан на официальном сайте сообщества. Читатель может самостоятельно ознакомиться с ним по ссылке http://nest-initiative.org/Writing_an_Extension_Module. В настоящем пособии предлагается использовать другой способ, о котором речь пойдёт ниже.

В Приложении 2 был описан процесс создания нового модуля. Здесь и далее предполагается, что у нас есть два написанных пользовательских модуля – модуль синаптической связи и модуль узла-нейрона. Созданный модуль синаптической связи состоит из двух файлов с именами `stdp_tsodyks2_synapse.h` и `stdp_tsodyks2_synapse.cpp`. Созданный модуль нейрона состоит из двух файлов с именами `hh_psc_alpha_rk.h` и `hh_psc_alpha_rk.cpp`. Для того чтобы начать работу с новыми модулями необходимо перекомпиллировать ядро нейросимулятора со всеми модулями, в том числе и с нашими новыми модулями. Процесс добавления нового модуля в NEST немного различен, и существует своя специфика как для нового узла (модели нейрона, а также стимулирующего или записывающего устройства), так и для новой связи. В качестве примера рассмотрим добавление обоих модулей.

Файлы новых модулей необходимо поместить в директорию с исходными кодами NEST, содержащую все модели, представленные в нейросимуляторе (`/path/to/nest/models`).

Перед тем как приступить к компиляции, необходимо сообщить ядру NEST о добавленном модуле. Для этого в директории с моделями редактируются следующие файлы: `Makefile.am`, `Makefile.in` и `modelsmodule.cpp`.

В файле `Makefile.am` после строки `libmodelsmodule_la_SOURCES= \` (обычно это строка №17) следует список модулей по одному на каждой строке в формате `modulename.h modulename.cpp\`

```
16 …
17 libmodelsmodule_la_SOURCES= \
18      ac_generator.h ac_generator.cpp\
19      aeif_cond_alpha.h aeif_cond_alpha.cpp\
20      aeif_cond_alpha_RK5.h aeif_cond_alpha_RK5.cpp\
21      aeif_cond_alpha_multisynapse.h aeif_cond_alpha_multisynapse.cpp\
22      aeif_cond_exp.h aeif_cond_exp.cpp\
23      binary_neuron.h binary_neuron_impl.h\
24      cont_delay_connection.h cont_delay_connection.cpp\
25      correlation_detector.h correlation_detector.cpp\
26      correlomatrix_detector.h correlomatrix_detector.cpp\
27      dc_generator.h dc_generator.cpp\
28      gamma_sup_generator.h gamma_sup_generator.cpp\
29      ginzburg_neuron.h ginzburg_neuron.cpp\
30      hh_cond_exp_traub.h hh_cond_exp_traub.cpp\
31      hh_psc_alpha.h hh_psc_alpha.cpp\
32      …
```

Следует вставить названия файлов наших модулей `hh_psc_alpha_rk.h` и `hh_psc_alpha_rk.cpp` и `stdp_tsodyks2_synapse.h` и `stdp_tsodyks2_synapse.cpp` в соответствующие места файла автосборки `Makefile.am` (см. строчки 32 и 79):

```
27      …
28      gamma_sup_generator.h gamma_sup_generator.cpp\
29      ginzburg_neuron.h ginzburg_neuron.cpp\
30      hh_cond_exp_traub.h hh_cond_exp_traub.cpp\
31      hh_psc_alpha.h hh_psc_alpha.cpp\
32      hh_psc_alpha_rk.h hh_psc_alpha_rk.cpp\
33      ht_connection.h ht_connection.cpp\
34      ht_neuron.h ht_neuron.cpp\
35      iaf_chxk_2008.cpp iaf_chxk_2008.h\
36      iaf_chs_2007.cpp iaf_chs_2007.h\
37      …
```

```
74      …
75      stdp_connection_facetshw_hom.h stdp_connection_facetshw_hom.cpp\
76      stdp_connection_hom.h stdp_connection_hom.cpp\
77      stdp_dopa_connection.h stdp_dopa_connection.cpp\
78      stdp_pl_connection_hom.h stdp_pl_connection_hom.cpp\
79      stdp_tsodyks2_connection.h stdp_tsodyks2_connection.cpp\
80      step_current_generator.h step_current_generator.cpp\
81      tsodyks2_connection.h tsodyks2_connection.cpp\
82      tsodyks_connection.h tsodyks_connection.cpp\
83      volume_transmitter.h volume_transmitter.cpp
84      …
```

В файле `Makefile.in` потребуется внести несколько больше изменений. Соответствующие строки, указывающие компилятору на присутствие новых модулей, прописываются во фрагменте кода со списком объектов (строчки 118 и 163).

```
113   …
114   libmodelsmodule_la-gamma_sup_generator.lo \
115   libmodelsmodule_la-ginzburg_neuron.lo \
116   libmodelsmodule_la-hh_cond_exp_traub.lo \
117   libmodelsmodule_la-hh_psc_alpha.lo \
118   libmodelsmodule_la-hh_psc_alpha_rk.lo \
119   libmodelsmodule_la-ht_connection.lo \
120   libmodelsmodule_la-ht_neuron.lo \
121   libmodelsmodule_la-iaf_chxk_2008.lo \
122   libmodelsmodule_la-iaf_chs_2007.lo \
123   …
```

```
158   …
159   libmodelsmodule_la-stdp_connection_facetshw_hom.lo \
160   libmodelsmodule_la-stdp_connection_hom.lo \
161   libmodelsmodule_la-stdp_dopa_connection.lo \
162   libmodelsmodule_la-stdp_pl_connection_hom.lo \
163   libmodelsmodule_la-stdp_tsodyks2_connection.lo \
164   libmodelsmodule_la-step_current_generator.lo \
165   libmodelsmodule_la-tsodyks2_connection.lo \
166   libmodelsmodule_la-tsodyks_connection.lo \
167   libmodelsmodule_la-volume_transmitter.lo
168   …
```

Соответствующие строки, прописываются во фрагменте кода со списком файлов исходных кодов модулей (строчки 516 и 563).

```
511     …
512     gamma_sup_generator.h gamma_sup_generator.cpp\
513     ginzburg_neuron.h ginzburg_neuron.cpp\
514     hh_cond_exp_traub.h hh_cond_exp_traub.cpp\
515     hh_psc_alpha.h hh_psc_alpha.cpp\
516     hh_psc_alpha_rk.h hh_psc_alpha_rk.cpp\
517     ht_connection.h ht_connection.cpp\
518     ht_neuron.h ht_neuron.cpp\
519     iaf_chxk_2008.cpp iaf_chxk_2008.h\
520     iaf_chs_2007.cpp iaf_chs_2007.h\
521     …
```

```
558     …
559     stdp_connection_facetshw_hom.h stdp_connection_facetshw_hom.cpp\
560     stdp_connection_hom.h stdp_connection_hom.cpp\
561     stdp_dopa_connection.h stdp_dopa_connection.cpp\
562     stdp_pl_connection_hom.h stdp_pl_connection_hom.cpp\
563     stdp_tsodyks2_connection.h stdp_tsodyks2_connection.cpp\
564     step_current_generator.h step_current_generator.cpp\
565     tsodyks2_connection.h tsodyks2_connection.cpp\
566     tsodyks_connection.h tsodyks_connection.cpp\
567     volume_transmitter.h volume_transmitter.cpp
568     …
```

Далее вносим по одной записи о каждом модуле соответственно в список, следующий за `distclean-compile`:

```
626 …
627 mostlyclean-compile:
628     -rm -f *.$(OBJEXT)
629
630 distclean-compile:
631     -rm -f *.tab.c
632 …
```

```
641 …
642 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        gamma_sup_generator.Plo@am__quote@
643 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        ginzburg_neuron.Plo@am__quote@
644 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        hh_cond_exp_traub.Plo@am__quote@
645 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        hh_psc_alpha.Plo@am__quote@
646 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        hh_psc_alpha_rk.Plo@am__quote@
647 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        ht_connection.Plo@am__quote@
648 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        ht_neuron.Plo@am__quote@
649 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        iaf_chs_2007.Plo@am__quote@
650 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        iaf_chxk_2008.Plo@am__quote@
651 …
```

```
686 …
687 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        stdp_connection_facetshw_hom.Plo@am__quote@
688 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        stdp_connection_hom.Plo@am__quote@
689 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        stdp_dopa_connection.Plo@am__quote@
690 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        stdp_pl_connection_hom.Plo@am__quote@
691 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        stdp_tsodyks2_connection.Plo@am__quote@
692 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        step_current_generator.Plo@am__quote@
693 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        tsodyks2_connection.Plo@am__quote@
694 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        tsodyks_connection.Plo@am__quote@
695 @AMDEP_TRUE@@am__include@ @am__quote@./$(DEPDIR)/libmodelsmodule_la-
        volume_transmitter.Plo@am__quote@
696 …
```

Наконец за секцией с информацией по сборке библиотеки `cpp.lo`

```
712 …
713 .cpp.lo:
714 @am__fastdepCXX_TRUE@  $(AM_V_CXX)depbase=`echo $@ | sed
        's|[^/]*$$|$(DEPDIR)/&|;s|\.lo$$||'`;\
715 @am__fastdepCXX_TRUE@  $(LTCXXCOMPILE) -MT $@ -MD -MP -MF $$depbase.Tpo
        -c -o $@ $< &&\
716 @am__fastdepCXX_TRUE@  $(am__mv) $$depbase.Tpo $$depbase.Plo
    @AMDEP_TRUE@@am__fastdepCXX_FALSE@    $(AM_V_CXX)source='$<' object='$@'
717     libtool=yes @AMDEPBACKSLASH@
    @AMDEP_TRUE@@am__fastdepCXX_FALSE@    DEPDIR=$(DEPDIR) $(CXXDEPMODE)
718     $(depcomp) @AMDEPBACKSLASH@
719 @am__fastdepCXX_FALSE@ $(AM_V_CXX@am__nodep@)$(LTCXXCOMPILE) -c -o $@ $<
720 …
```

следуют по несколько строк с информацией по сборке всех модулей с моделями узлов и связей. Для каждого из наших модулей необходимо добавить по строчке. Информацию по модулю модели нейрона с изменённым интегратором добавляем в строки №№ 812-817, а информацию по модулю синаптической связи добавляем в строки №№ 1127-1132 – всего по 14 вхождений каждого названия модуля.

```
811 …
812 libmodelsmodule_la-hh_psc_alpha_rk.lo: hh_psc_alpha_rk.cpp
813 @am__fastdepCXX_TRUE@  $(AM_V_CXX)$(LIBTOOL) $(AM_V_lt) --tag=CXX
        $(AM_LIBTOOLFLAGS) $(LIBTOOLFLAGS) --mode=compile $(CXX) $(DEFS)
        $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS)
        $(libmodelsmodule_la_CXXFLAGS) $(CXXFLAGS) -MT libmodelsmodule_la-
        hh_psc_alpha_rk.lo -MD -MP -MF $(DEPDIR)/libmodelsmodule_la-
        hh_psc_alpha_rk.Tpo -c -o libmodelsmodule_la-hh_psc_alpha_rk.lo
        `test -f 'hh_psc_alpha_rk.cpp' || echo
        '$(srcdir)/'`hh_psc_alpha_rk.cpp
814 @am__fastdepCXX_TRUE@  $(AM_V_at)$(am__mv) $(DEPDIR)/libmodelsmodule_la-
        hh_psc_alpha_rk.Tpo $(DEPDIR)/libmodelsmodule_la-hh_psc_alpha_rk.Plo
815 @AMDEP_TRUE@@am__fastdepCXX_FALSE@
         $(AM_V_CXX)source='hh_psc_alpha_rk.cpp' object='libmodelsmodule_la-
        hh_psc_alpha_rk.lo' libtool=yes @AMDEPBACKSLASH@
816 @AMDEP_TRUE@@am__fastdepCXX_FALSE@    DEPDIR=$(DEPDIR) $(CXXDEPMODE)
        $(depcomp) @AMDEPBACKSLASH@
817 @am__fastdepCXX_FALSE@ $(AM_V_CXX@am__nodep@)$(LIBTOOL) $(AM_V_lt) --
        tag=CXX $(AM_LIBTOOLFLAGS) $(LIBTOOLFLAGS) --mode=compile $(CXX)
```

```
    $(DEFS) $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS)
    $(libmodelsmodule_la_CXXFLAGS) $(CXXFLAGS) -c -o libmodelsmodule_la-
    hh_psc_alpha_rk.lo `test -f 'hh_psc_alpha_rk.cpp' || echo
    '$(srcdir)/'`hh_psc_alpha_rk.cpp
…
```

```
…
libmodelsmodule_la-stdp_tsodyks2_connection.lo:
    stdp_tsodyks2_connection.cpp
@am__fastdepCXX_TRUE@  $(AM_V_CXX)$(LIBTOOL) $(AM_V_lt) --tag=CXX
    $(AM_LIBTOOLFLAGS) $(LIBTOOLFLAGS) --mode=compile $(CXX) $(DEFS)
    $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS)
    $(libmodelsmodule_la_CXXFLAGS) $(CXXFLAGS) -MT libmodelsmodule_la-
    stdp_tsodyks2_connection.lo -MD -MP -MF
    $(DEPDIR)/libmodelsmodule_la-stdp_tsodyks2_connection.Tpo -c -o
    libmodelsmodule_la-stdp_tsodyks2_connection.lo `test -f
    'stdp_tsodyks2_connection.cpp' || echo
    '$(srcdir)/'`stdp_tsodyks2_connection.cpp
@am__fastdepCXX_TRUE@  $(AM_V_at)$(am__mv) $(DEPDIR)/libmodelsmodule_la-
    stdp_tsodyks2_connection.Tpo $(DEPDIR)/libmodelsmodule_la-
    stdp_tsodyks2_connection.Plo
@AMDEP_TRUE@@am__fastdepCXX_FALSE@
     $(AM_V_CXX)source='stdp_tsodyks2_connection.cpp' ob-
    ject='libmodelsmodule_la-stdp_tsodyks2_connection.lo' libtool=yes
    @AMDEPBACKSLASH@
@AMDEP_TRUE@@am__fastdepCXX_FALSE@    DEPDIR=$(DEPDIR) $(CXXDEPMODE)
    $(depcomp) @AMDEPBACKSLASH@
@am__fastdepCXX_FALSE@ $(AM_V_CXX@am__nodep@)$(LIBTOOL) $(AM_V_lt) --
    tag=CXX $(AM_LIBTOOLFLAGS) $(LIBTOOLFLAGS) --mode=compile $(CXX)
    $(DEFS) $(DEFAULT_INCLUDES) $(INCLUDES) $(AM_CPPFLAGS) $(CPPFLAGS)
    $(libmodelsmodule_la_CXXFLAGS) $(CXXFLAGS) -c -o libmodelsmodule_la-
    stdp_tsodyks2_connection.lo `test -f 'stdp_tsodyks2_connection.cpp'
    || echo '$(srcdir)/'`stdp_tsodyks2_connection.cpp
…
```

Третий файл, подлежащий модификации для добавления написанных модулей в нейросимулятор – `modelsmodule.cpp`. В секцию заголовков моделей нейронов включаем заголовок нашей модели с изменённым интегратором (строка 51), а в секцию заголовков прототипов моделей синапсов включаем заголовок нашей модели синапса (строка 117). Отметим, что файл содержит также другие секции

заголовков для стимулирующих девайсов, записывающих девайсов, шаблонов для разработки прототипов синапсов. При добавлении этих модулей аналогичным образом включаются их соответствующие файлы заголовков.

```
43  …
44  // Neuron models
45  #include "aeif_cond_alpha.h"
46  #include "aeif_cond_alpha_RK5.h"
47  #include "aeif_cond_alpha_multisynapse.h"
48  #include "aeif_cond_exp.h"
49  #include "hh_cond_exp_traub.h"
50  #include "hh_psc_alpha.h"
51  #include "hh_psc_alpha_rk.h"
52  #include "ht_neuron.h"
53  #include "iaf_chs_2007.h"
54  #include "iaf_chxk_2008.h"
55  #include "iaf_cond_alpha.h"
56  …
```

```
102 …
103 // Prototypes for synapses
104
105 #include "common_synapse_properties.h"
106 #include "static_connection.h"
107 #include "static_connection_hom_wd.h"
108 #include "cont_delay_connection.h"
109 #include "tsodyks_connection.h"
110 #include "tsodyks2_connection.h"
111 #include "quantal_stp_connection.h"
112 #include "stdp_connection.h"
113 #include "stdp_connection_hom.h"
114 #include "stdp_connection_facetshw_hom.h"
115 #include "stdp_pl_connection_hom.h"
116 #include "stdp_dopa_connection.h"
117 #include "stdp_tsodyks2_connection.h"
118 #include "ht_connection.h"
119 …
```

Напоследок, необходимо добавить регистрацию новых модулей в ядре нейросимулятора при инициализации. Для этого в теле конструктора модуля всех модулей пропишем соответствующие строки.

```
149 …
150 void ModelsModule::init(SLIInterpreter *)
151   {
152 …
  …
194 …
195 #ifdef HAVE_GSL
196     register_model<iaf_chxk_2008>(net_,       "iaf_chxk_2008");
197     register_model<iaf_cond_alpha>(net_,      "iaf_cond_alpha");
198     register_model<iaf_cond_exp>(net_,        "iaf_cond_exp");
199     register_model<iaf_cond_exp_sfa_rr>(net_, "iaf_cond_exp_sfa_rr");
200     register_model<iaf_cond_alpha_mc>(net_,   "iaf_cond_alpha_mc");
201     register_model<hh_psc_alpha>(net_,        "hh_psc_alpha");
202     register_model<hh_psc_alpha_rk>(net_,       "hh_psc_alpha_rk");
203     register_model<hh_cond_exp_traub>(net_,   "hh_cond_exp_traub");
204     register_model<sinusoidal_gamma_generator>
          (net_,"sinusoidal_gamma_generator");
205 #endif
206 …
  …
223 …
224     // register synapses
225
226     // static connection with weight, delay, rport, target
227     register_prototype_connection<StaticConnection>(net_,    "stat-
    ic_synapse");
228
229     // static connection with rport, target and common weight and delay
230     register_prototype_connection_commonproperties_hom_d
                                       <     StaticConnectionHomWD,
231                                            CommonPropertiesHomWD
232                                       > (net_, "static_synapse_hom_wd");
233
234     register_prototype_connection<ContDelayConnection>(net_,
          "cont_delay_synapse");
235     register_prototype_connection<TsodyksConnection>(net_,
          "tsodyks_synapse");
```

```
236     register_prototype_connection<Tsodyks2Connection>(net_,
           "tsodyks2_synapse");
237     register_prototype_connection<STDPConnection>(net_,
           "stdp_synapse");
238     register_prototype_connection<STDPTsodyks2Connection>(net_,
           "stdp_tsodyks2_synapse");
239     register_prototype_connection<HTConnection>(net_,
           "ht_synapse");
240     register_prototype_connection< Quantal_StpConnection>(net_,
           "quantal_stp_synapse");
241 …
  …
257 …
258   }
259 …
```

Не забудьте сохранить изменения, сделанные в файлах. После этого можно приступать к компиляции NEST. Процедура стандартная и описана в разделе «Приложение 1. Установка симулятора NEST на ОС семейства Linux». По окончании процедуры компиляции добавленные модели станут доступны в списке моделей нейросимулятора NEST. Например, можно проверить после приведённых выше действий по изменению некоторых файлов исходного кода и файлов компилятора NEST, появились ли добавленные модели в списке моделей. Для этого откроем терминал, запустим интерпретатор языка python, командой `ipython`, импортируем модуль PyNEST командой `import nest` и выведем список моделей при помощи команды `nest.Models()`. В результате интерпретатор ipython должен выдать кортеж, содержащий строки юникода с названиями всех моделей и упорядоченный по алфавиту.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Роль математического моделирования в науке.
2. Почему важно использовать стандартизированные симуляторы?
3. Какие вы знаете стандартизированные симуляторы активности нейронных сетей?
4. Для чего предназначен программный пакет NEST?
5. Как устроен NEST?
6. Какие модели реализованы в NEST? Как посмотреть список моделей?
7. Какие технологии параллельных вычислений использует NEST?
8. Как организовать расчёт модели на вычислительном кластере?
9. Какие методы реализованы во всех моделях нейронов в NEST?
10. Как осуществляется установка NEST?
11. Напишите модель активации одного нейрона за счёт спайк-генератора, подающего периодическую последовательность спайков.
12. Исследуйте порог генерации в модели Ходжкина-Хаксли, подавая входной ток различной амплитуды.
13. Соедините два нейрона посредством модели STDP пластичности. Подавая различные последовательности спайков на оба нейрона при помощи спайк-генераторов, продемонстрируйте увеличение и уменьшение синаптической эффективности.

СПИСОК ЦИТИРУЕМЫХ ИСТОЧНИКОВ

1. Lim D.-J., Anderson T.R., Shott T. Technological forecasting of supercomputer development: The March to Exascale computing // Omega. 2015. Vol. 51. P. 128–135.
2. Макаров И.М. Фундаментальные основы математического моделирования. Наука, 1997. P. 201.
3. Piccinini G., Shagrir O. Foundations of computational neuroscience. // Curr. Opin. Neurobiol. 2014. Vol. 25. P. 25–30.
4. Sompolinsky H. Computational neuroscience: beyond the local circuit. // Curr. Opin. Neurobiol. 2014. Vol. 25. P. xiii – xviii.
5. Izhikevich E.M. Dynamical systems in neuroscience: the geometry of excitability and bursting // Dyn. Syst. The MIT press, 2007.
6. Dayan P., Abarbanel H. Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems // Neuroscience / ed. Dayan P., Abbott L. MIT Press, 2001. Vol. 39, № 3. P. 460.
7. Rabinovich M. et al. Dynamical principles in neuroscience // Rev. Mod. Phys. 2006. Vol. 78, № 4. P. 1213–1265.
8. Sandler U., Tsitolovsky L. Neural Cell Behavior and Fuzzy Logic / ed. Sandler U., Tsitolovsky L. Springer US, 2008. № 000.
9. Nicholls J.C., Martin A.R., Wallace B.C. От нейрона к мозгу. 2003. P. 672.
10. Kandel E.R., Schwartz J.H., Jessell T.M. Principles of Neural Science // Neurology / ed. Kandel E.R., Schwartz J.H., Jessell T.M. McGraw-Hill, 2000. Vol. 4, № 22. P. 1414.
11. Nordlie E., Gewaltig M.-O., Plesser H.E. Towards reproducible descriptions of neuronal network models. // PLoS Comput. Biol. 2009. Vol. 5, № 8. P. e1000456.
12. Brette R. et al. Simulation of networks of spiking neurons: A review of tools and strategies. № Bat 33. P. 1–66.
13. Migliore M. et al. Parallel network simulations with NEURON. // J. Comput. Neurosci. Springer, 2006. Vol. 21, № 2. P. 119–129.

14. King J.G. et al. A Component-Based Extension Framework for Large-Scale Parallel Simulations in NEURON // Front. Neuroinform. Frontiers Research Foundation, 2009. Vol. 3, № April. P. 11.

15. Pecevski D., Natschläger T., Schuch K. PCSIM: A Parallel Simulation Environment for Neural Circuits Fully Integrated with Python // Front. Neuroinform. Frontiers Research Foundation, 2009. Vol. 3, № May. P. 15.

16. Vassanelli S. et al. On the Way to Large-Scale and High-Resolution Brain-Chip Interfacing // Cognit. Comput. 2012. Vol. 4, № 1. P. 71–81.

17. Van Albada S.J. et al. International Workshop, BrainComp 2013, Cetraro, Italy, July 8-11, 2013, Revised Selected Papers / ed. Grandinetti L., Lippert T., Petkov N. Cham: Springer International Publishing, 2014. Vol. 8603. P. pp 22–32.

18. Indiveri G. et al. Neuromorphic silicon neuron circuits. // Front. Neurosci. 2011. Vol. 5. P. 73.

19. Hammarlund P., Ekeberg O. Large neural network simulations on multiple hardware platforms. // J. Comput. Neurosci. 1998. Vol. 5, № 4. P. 443–459.

20. Mokhtar M., Halliday D. Hippocampus-inspired spiking neural network on FPGA // Evolvable Syst. From Biol. to. 2008. P. 362–371.

21. Gewaltig M.-O., Diesmann M. NEST (NEural Simulation Tool) // Scholarpedia / ed. Izhikevich E. Eugene Izhikevich, 2007. Vol. 2, № 4. P. 1430.

22. Carnevale N.T., Hines M.L. The NEURON Book // Neuron. Cambridge University Press, 2006. Vol. 30, № 2. P. 457.

23. Drewes R., Zou Q., Goodman P.H. Brainlab: A Python Toolkit to Aid in the Design, Simulation, and Analysis of Spiking Neural Networks with the NeoCortical Simulator // Front. Neuroinform. Frontiers Research Foundation, 2009. Vol. 3, № May. P. 10.

24. Kunkel S. et al. Spiking network simulation code for petascale computers // Front. Neuroinform. 2014. Vol. 8. P. 78.

25. Eppler J.M. et al. PyNEST: A Convenient Interface to the NEST Simulator // Front. Neuroinform. Frontiers Research Foundation, 2009. Vol. 2, № January. P. 12.

26. Zaytsev Y. V, Morrison A. CyNEST: a maintainable Cython-based interface for the NEST simulator. // Front. Neuroinform. 2014. Vol. 8. P. 23.

27. Markram H., Wang Y., Tsodyks M. Differential signaling via the same axon of neocortical pyramidal neurons // Proc. Natl. Acad. Sci. 1998. Vol. 95, № 9. P. 5323–5328.

28. Gütig R. et al. Learning input correlations through nonlinear temporally asymmetric Hebbian plasticity. // J. Neurosci. 2003. Vol. 23, № 9. P. 3697–3714.

ИСПОЛЬЗОВАНИЕ СИМУЛЯТОРА НЕЙРОННЫХ СЕТЕЙ NEST В ВЫСОКОПРОИЗВОДИТЕЛЬНЫХ ВЫЧИСЛЕНИЯХ

Автор:
Александр Юрьевич **Симонов**

Учебно-методическое пособие

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
"Нижегородский государственный университет им. Н.И. Лобачевского".

603950, Нижний Новгород, пр. Гагарина, 23.

Подписано в печать . Формат 60x84 1/16.
Бумага офсетная. Печать офсетная. Гарнитура Таймс.
Усл. печ. л. . Уч.-изд. л. .
Заказ № . Тираж 70 экз.

Отпечатано в типографии Нижегородского госуниверситета
им. Н.И. Лобачевского
603600, г. Нижний Новгород, ул. Большая Покровская, 37
Лицензия ПД № 18-0099 от 14.05.01