Министерство науки и высшего образования РФ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Уфимский государственный авиационный технический университет»

Кафедра Высокопроизводительных вычислительных технологий и систем

	1	2	3	4	5	6	7	8	9	10
100										
90										
80										
70										
60										
50										
40										
30										
20										
10										
0										

ИСПОЛЬЗОВАНИЕ ДИНАМИЧЕСКИХ СИСТЕМ НА ПРИМЕРЕ НЕЙРОННЫХ СИГНАЛОВ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе по дисциплине

«Дифференциальные уравнения»

3952.334113.000 ПЗ

Группа ПМ-255	Фамилия И.О.	Подпись	Дата	Оценка
Студент	Султанов Р.Р			
Консультант	Касаткин А.А.			
Принял	Лукащук В.О.			

Уфа 2022

Федеральное государственное бюджетное образовательное учреждение высшего образования
«Уфимский государственный авиационный технический университет»

Кафедра Высокопроизводительных вычислительных технологий и систем

ЗАДАНИЕ

на курсовую работу по дисциплине

«Дифференциальные уравнения»

Студент: Султанов Равиль Радикович Группа: ПМ-255

Консультант: Касаткин Алексей Александрович

1. Тема курсовой работы

Использование динамических систем на примере нейронных сигналов

2. Основное содержание

2.1. Изучить различные модели нейронов.
2.2. Реализовать программу для численного моделирования нейронов.
2.3. На основе численных моделей исследовать свойства и поведение нейронов.
2.4. Оформить пояснительную записку к курсовой работе.

3. Требования к оформлению материалов работы

Требования к оформлению пояснительной записки

Пояснительная записка к курсовой работе должна быть оформлена в соответствии с требованиями ГОСТ и содержать

- титульный лист,
- задание на курсовую работу,
- содержание,
- введение,
- заключение,

- список литературы,
- приложение, содержащее листинг разработанной программы, если таковая имеется.

Дата выдачи задания Дата окончания работы

"__" ________ 202_ г. "__" __________ 202_ г.

Консультант _________________ Касаткин А.А.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель: изучить динамические системы описывающие биологические нейроны и их сигналы

Актуальность: мозг является очень сложной структурой, свойства и процессы которого до сих пор до конца не раскрыты и не объяснены. Благодаря различным исследованиям удаётся найти некоторые структурные особенности мозга на клеточном или молекулярном уровне, количественные зависимости между свойствами и то, как они определяют дальнейшее поведение мозга. Это позволяет создать модель, которая с некоторой точностью сможет описать мозг или его часть с помощью математических законов. Благодаря такой модели появляется возможность изучать свойства мозга и нервных тканей используя уже только вычислительные мощности компьютеров. Также, поскольку мозг и нейронные связи обладают особой структурой и алгоритмическими свойствами, открывается возможность создавать нетипичные архитектуры процессоров, вычислительных устройств, составлять принципиально новые алгоритмы, которые будут эффективнее справляться с определёнными вычислительными задачами, чем устройства с архитектурой предыдущего поколения.

В работе поставлены следующие задачи:

1) Изучить существующие динамические модели нейронов и их особенности как динамических систем
2) Выполнить программную реализацию моделей с применением численных методов
3) Исследовать поведение каждой модели при различных состояниях

1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1. Динамические системы

Динамической системой [1] на некотором множестве E называют непрерывно-дифференцируемое отображение $\phi: \mathbb{R} \times E \to E$, где E - открытое подмножество $\mathbb{R}^n$, для которого выполняются следующие условия:

1) $\phi_0(\boldsymbol{x}) = \boldsymbol{x}, \forall \boldsymbol{x} \in E$,
2) $\phi_t \circ \phi_s\,(\boldsymbol{x}) = \phi_{t+s}(\boldsymbol{x}), \forall s, t \in \mathbb{R}$ и $\forall \boldsymbol{x} \in E$,

где $\phi_t(\boldsymbol{x}) = \phi(t, \boldsymbol{x})$.

Динамические системы описывают объекты и процессы, для которых определено понятие состояния, как совокупности некоторых величин в некоторый момент времени, и задан *закон эволюции* ϕ, описывающий эволюцию начального состояния с течением времени [2]. Такими объектами могут послужить физические, химические, биологические и другие процессы.

Закон эволюции некоторых систем может определяться решением автономной системы обыкновенных дифференциальных уравнений (ОДУ):

$$\frac{dx_k}{dt} = f_k(x_1, \dots, x_k), \quad k = 1, 2, \dots, n$$

Система называется неавтономной, если хотя бы одна из функций f_k зависит от времени, т.е. $f_k = f_k(x_1, \dots, x_k, t)$.

Если система зависит от некоторых параметров, которые влияют на её поведение, то правую часть уравнения можно записать в виде $f_k(x_1, \dots, x_k, a)$, где $a \in \mathbb{R}^m$ набор из m параметров.

Изменение параметров может приводить к кардинальному изменению поведения системы, что проявляется в виде качественных изменений фазового портрета – *бифуркациях*. Значения параметров, при которых возникают бифуркации называют *бифуркационными*.

Под качественными изменениями подразумевается появление новых точек равновесия или их исчезновение (например, “складка” или “вилка”), возникновение предельных циклов, смена устойчивости аттракторов и т.д.

1.2. Электрофизиология нейронов

Нейроны — электрически возбудимые клетки организма, которые отвечают за передачу, обработку и хранение информации. Нервные клетки передают по аксонам различные выходные сигналы в зависимости от вида входного сигнал, полученного по дендритам, и характеристик самого нейрона.

Входные импульсы с синапсов меняют мембранный потенциал нейрона – постсинаптический потенциал (ПСП). При малом токе порождаются малый ПСП, но при достаточно больших токах генерируются высокий ПСП, который затем усиливается потенциал-чувствительными каналами в мембране и влечёт за собой быстрое короткое изменение мембранного потенциала, который далее распространяется по отростку – *спайк (потенциал действия)*. Тип такого сигнала называют «всё или ничего» («all-or-none»), так как на выходе нейрон либо спокоен, либо передаёт сигнал в окрестности некоторого предельного значения. Затем происходит возращение нейрона к исходному состоянию.

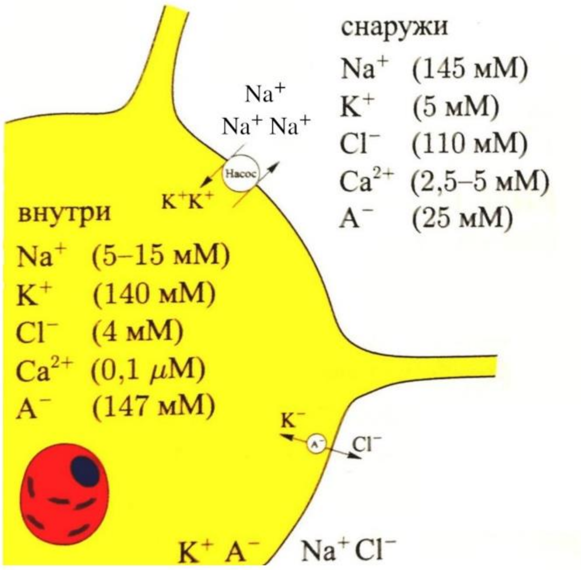

Рисунок 1 - Ионные концентрации снаружи и внутри клетки

Данное поведение нейрона можно объяснить с помощью его электрофизиологических особенностей – ионных токов, проходящих по каналам через мембрану. Внутриклеточная среда содержит высокую концентрацию ионов калия (K^+) и отрицательно заряженных молекул (обозначенных за A^-). Во внеклеточной среде поддерживается высокая концентрация ионов натрия (Na^+), кальция (Ca^{2+}) и относительно высокая концентрация ионов хлора (Cl^-) (Рисунок 1) [3]. Полученный градиент концентрации является основной движущей силой трансмембранного тока.

Ввиду различной концентрации вне и внутри мембраны возникает диффузия, которая влечёт за собой изменение разности электрического потенциала на мембране. Диффузия продолжается до тех пор, пока силы, вызванные диффузией и разностью потенциалов, не уравновесят друг друга, иначе говоря, пока напряжение на мембране, не станет равным *равновесному потенциалу* для соответствующего иона. Его значение можно отыскать по формуле Нернста:

$$E_{ion} = \frac{RT}{zF}\ln\frac{[\text{Ion}]_{out}}{[\text{Ion}]_{in}},$$

где R – универсальная газовая постоянная (8,315 Дж/(К° · моль)), T – температура в кельвинах, F – постоянная Фарадея (96 480 Кл/моль), z – валентность ионов, $[\text{Ion}]_{in}$ и $[\text{Ion}]_{out}$ – концентрация иона внутри и снаружи клетки соответственно.

Ионный ток вычисляется следующим образом:

$$I_{ion} = g_{ion}(V - E_{ion}),$$

где $g_{ion} = g_{ion}(V,t)$ – проводимость для соответствующего иона (мСм/см^2).

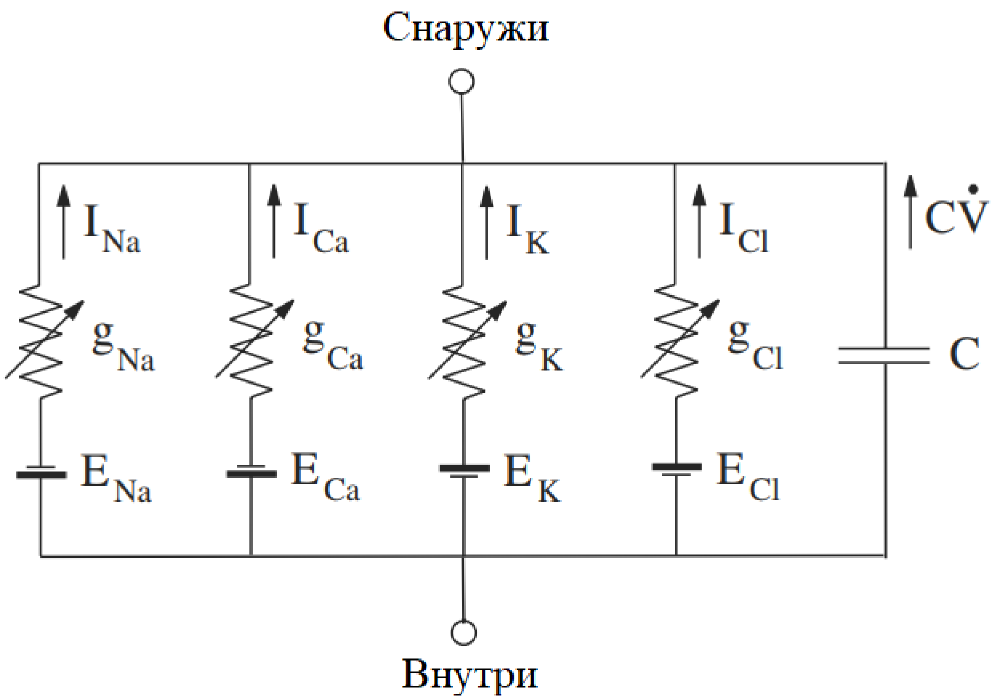

Рисунок 2 - Электрическая схема, эквивалентная мембране

Электрические свойства мембраны можно описать с помощью эквивалентной электрической схемы (Рисунок 2). I – общий ток через мембрану, C – ёмкость мембраны. Уравнение для цепи:

$$C\dot{V} = I - I_{Na} - I_{Ca} - I_K - I_{Cl},$$

$$C\dot{V} = I - g_{Na}(V - E_{Na}) - g_{Ca}(V - E_{Ca}) - g_K(V - E_K) - g_{Cl}(V - E_{Cl}).$$

Проводимость ионных каналов управляется специальными “воротами”, способными открывать и закрывать каналы. В данной работе будут рассмотрены модели, в которых состояние ворот определяется мембранным потенциалом.

Ток через популяцию идентичных каналов можно выразить как:

$$I = \bar{g}p(V - E),$$

где p – усреднённая доля каналов в открытом состоянии, $\bar{g}$ – максимальная проводимость ионных каналов, E – потенциал, при котором ток меняет своё направление - *реверсивный потенциал* (если каналы пропускают только один тип ионов, то реверсивный потенциал равен равновесному потенциалу для данного иона).

Выделяют два типа ворот: активирующие (открывают каналы), инактивирующие (закрывают каналы). Вероятность открытия активирующих каналов обозначают переменной m (переменной n в случае K^+ и Cl^- каналов), вероятность срабатывания инактивационных ворот - переменной h. Пусть a и b – количество активационных и инактивационных ворот соответственно. Тогда доля открытых каналов:

$$p = m^a h^b.$$

В свою очередь значение активационных и инактивационных переменных описывается дифференциальными уравнениями:

$$\dot{m} = (m_\infty(V) - m)/\tau_m(V), \quad (1)$$

$$\dot{h} = (h_\infty(V) - h)/\tau_h(V), \quad (2)$$

где $m_\infty(V)$ и $h_\infty(V)$ – потенциал-чувствительные равновесные активационная и инактивационная функции (асимптотическое значение при $t \to \infty$ при фиксированном потенциале). Функции $\tau_m(V)$ и $\tau_h(V)$ – временные постоянные, которые определяют динамичность переменных (чем меньше τ, тем быстрее изменятся переменная).

С точки зрения динамических систем возбудимость нейронов обусловлена нахождением состояния нейрона вблизи точек бифуркации. В качестве бифуркационного параметра выступает ток извне. Изменение тока влечёт за собой изменение фазового портрета, в результате нейрон переходит в иной режим активности. Характер бифуркации и соответственно вид нового фазового портрета определяет дальнейшее поведение нейрона. Например, состояние покоя может возникнуть из-за появления устойчивого фокуса или узла, а периодические спайки нейрона – из-за появления устойчивых предельных циклов.

2. ПРАКТИЧЕСКАЯ ЧАСТЬ

В этой части рассмотрены некоторые динамические модели нейронов. Каждая модель была реализована на языке Python с помощью библиотек *numpy, matplotlib, pygame* с использованием метода Рунге-Кутта 4-го порядка.

Для каждой модели была реализовано интерактивная программа, моделирующая поведение одиночного нейрона. Она позволяет наблюдать за изменением состояния нейрона на фазовой плоскости, отображать нульклины и точки равновесия; воздействовать на нейрон с помощью постоянного тока или периодических прямоугольных импульсов; строить графики изменения характеристик нейрона по собранным данным. Чтоб можно было проследить не только состояние нейрона в данный момент, но и судить о предшествующем поведении, на фазовой плоскости отображается траектория нейрона, его “след”

Также, для демонстрации особенностей фазового портрета систем, для каждой модели была создана программа моделирующая одновременно большое количество нейронов с различными случайными стартовыми значениями. Модель отображает их на одновременно на одной плоскости, а также их траекторию, что позволяет судить о характере фазового портрета системы, наличии узлов, фокусов, предельных циклов, об их устойчивости. Далее моментальные снимки фазового пространства с точками на них, полученные от данной программы, будут называться *картами траекторий*.

2.1. Модель Ходжкина-Хаксли (1952)

Модель гигантского аксона кальмара, который имеет три основных тока: потенциал-зависимый K^+ с 4 активационными воротами, потенциал-зависимый Na^+ с 3 активационными воротами и с 1 инактивационными воротами и омический ток (ввиду постоянной проводимости) утечки, создаваемый ионами Cl^-[4].

$$\begin{cases} C\dot{V} = I - \bar{g}_K n^4 (V - E_K) - \bar{g}_{Na} m^3 h (V - E_{Na}) - g_L (V - E_L) \\ \dot{n} = \alpha_n(V)(1-n) - \beta_n(V) n \\ \dot{m} = \alpha_m(V)(1-m) - \beta_m(V) m \\ \dot{h} = \alpha_h(V)(1-h) - \beta_{h(V)} h \end{cases}$$

где

$$\alpha_n(V) = 0.01 \frac{10 - V}{exp\left(\frac{10 - V}{10}\right) - 1},$$

$$\beta_n(V) = 0.125 \exp\left(\frac{-V}{80}\right),$$

$$\alpha_m(V) = 0.1 \frac{25 - V}{\exp\left(\frac{25 - V}{10}\right) - 1},$$

$$\beta_m(V) = 4 \exp\left(\frac{-V}{18}\right),$$

$$\alpha_h(V) = 0.07 \exp\left(\frac{-V}{20}\right),$$

$$\beta_h(V) = \frac{1}{\exp\left(\frac{30 - V}{10}\right) + 1}.$$

Значения равновесных потенциалов и проводимостей:

$$E_K = -12 \text{ мВ}, \quad E_{Na} = 120 \text{ мВ}, \quad E_L = 10{,}6 \text{ мВ},$$

$$\bar{g}_K = 36 \text{ мСм/см}^2, \quad \bar{g}_{Na} = 120 \text{ мСм/см}^2, \quad \bar{g}_L = 0{,}3 \text{ мСм/см}^2.$$

Потенциал покоя равен $V \approx 0$ мВ.

Или можно представить динамику воротных переменных согласно формулам (1)(2):

$$\dot{n} = \frac{(n_\infty(V) - n)}{\tau_n(V)}, \qquad n_\infty = \frac{\alpha_n}{\alpha_n + \beta_n}, \qquad \tau_n = \frac{1}{\alpha_n + \beta_n};$$

$$\dot{m} = \frac{(m_\infty(V) - m)}{\tau_m(V)}, \qquad m_\infty = \frac{\alpha_m}{\alpha_m + \beta_m}, \qquad \tau_m = \frac{1}{\alpha_m + \beta_m};$$

$$\dot{h} = \frac{(h_\infty(V) - h)}{\tau_h(V)}, \qquad h_\infty = \frac{\alpha_h}{\alpha_h + \beta_h}, \qquad \tau_h = \frac{1}{\alpha_h + \beta_h};$$

Модель Ходжкина-Хаксли является 4-мерной, что усложняет её подробный анализ. Но она подходит для демонстрации поэтапного цикла возникновения спайка. Далее приведён анализ модели, реализованный средствами языка Python.

Изначально модель находится в состоянии покоя (Рисунок 3, промежуток 1):

$$V = 0 \text{ мВ}, \qquad n = 0{,}318, \qquad m = 0{,}053, \qquad h = 0{,}59.$$

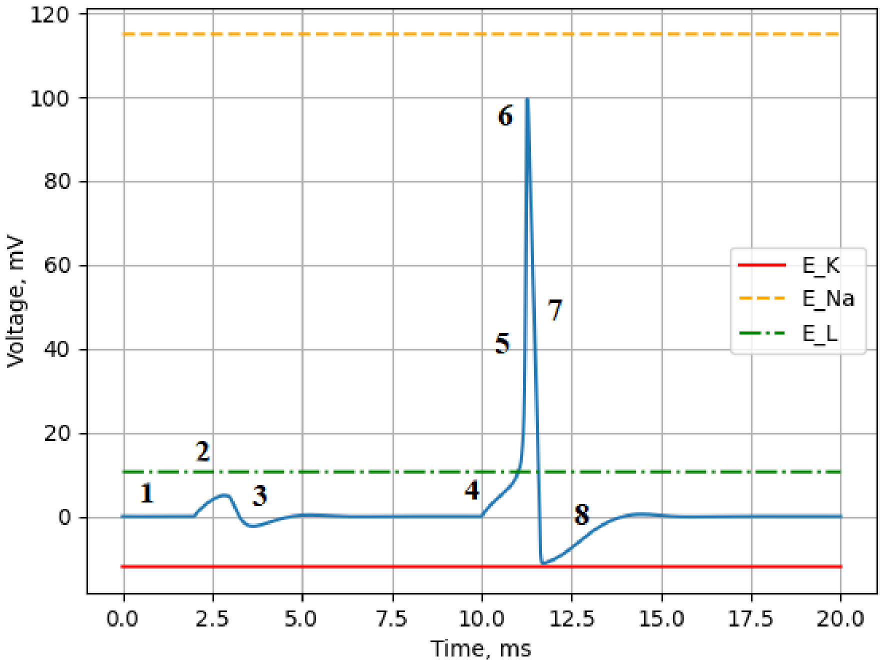

Рисунок 3 - Замеры значения мембранного потенциала при малой и большой деполяризации

На модель в течение 1мс подаётся ток величиной 2 мА в момент времени $t = 2$ мс и ток величиной 2,3 мА в момент времени $t = 10$ мс.

Поскольку состояние покоя устойчиво, первый импульс тока вызывает малое изменение мембранного потенциала (малую деполяризацию - промежуток 2), которое вызывает ток, возвращающий систему в состояние покоя (реполяризация - промежуток 3).

Второй импульс тока вызывает достаточное изменение мембранного потенциала (сильная деполяризация – промежуток 4), для возбуждения нейрона. Такая деполяризация вызывает увеличение активационных переменных n и m и уменьшение инактивационной h (Рисунок 4). Резкое возрастание m обусловлено малым значением временной постоянной, из-за чего возрастает проводимость и ток для иона Na^+, происходит дальнейшая деполяризация (промежуток 5), скачкообразное возрастание V и спайк (промежуток 6). К моменту возникновение спайка медленные переменные m и h достаточно возросли. Ток Na^+ постепенно инактивируется, ионный ток K^+ возрастает, что возвращает напряжение к состоянию покоя (промежуток 7). Поскольку соответствующие им временные постоянные велики, при достижении нулевого значения V ионный ток K^+остаётся активированным, что вызывает послегиперполяризацию (промежуток 8).

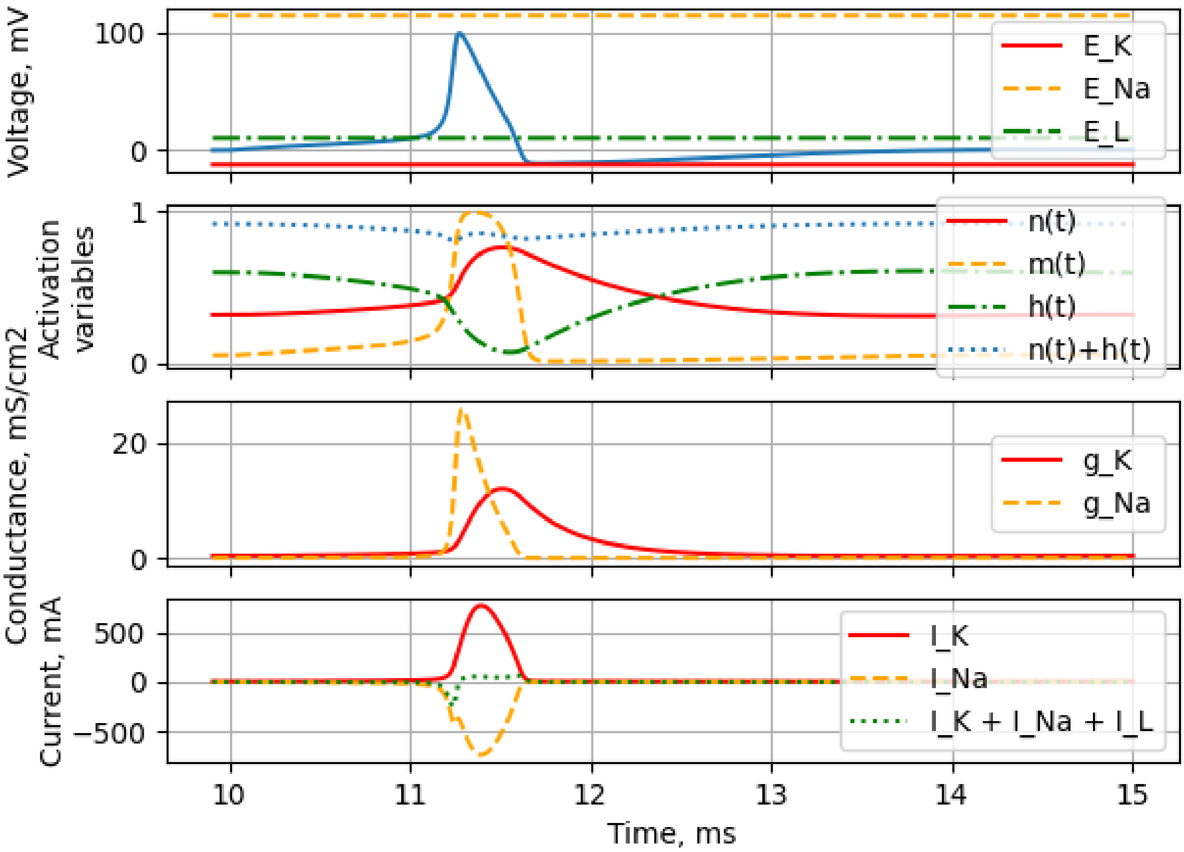

Рисунок 4 - Изменение значений мембранного потенциала, активационных переменных, проводимости и ионных токов во время генерации спайка

2.2. Редукция модели Ходжкина-Хаксли

Чтобы изучить динамику данной многомерной системы можно прибегнуть к методу редукции динамической системы до двумерной системы.

На Рисунке 4 заметна приближенная связь между воротными переменными $n(t) + h(t) \approx const.$ Более точной взаимосвязь описывается уравнением $h = 0{,}89 - 1{,}1n$ (Рисунок 5). Таким образом модель сведена к 3-мерной.

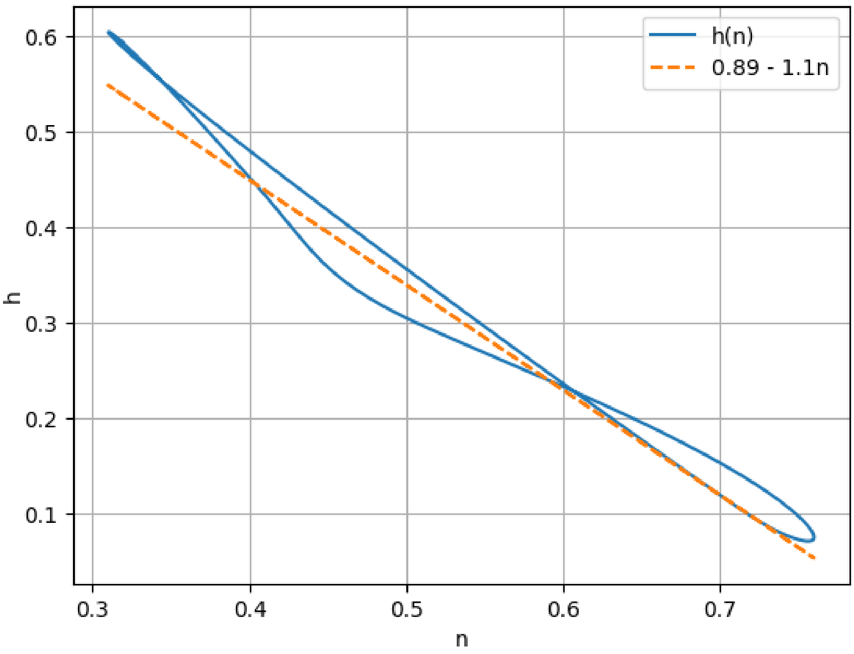

Рисунок 5 - Взаимосвязь между $\boldsymbol{n(t)}$ и $\boldsymbol{h(t)}$, а также график функции $\boldsymbol{h(n) = 0{,}89 - 1{,}1 * n}$

Также, на основании результатов с полной модели, можно с некоторой точностью считать активационную кинетику Na^{+}-тока мгновенной $m = m_{\infty}(V)$. В итоге получается двумерная система, которая отражает некоторые качественные и количественные характеристики исходной системы:

$$\begin{cases} C\dot{V} = I - \bar{g}_K n^4 (V - E_K) - \bar{g}_{Na} m_{\infty}(V)^3 (0{,}89 - 1{,}1n)(V - E_{Na}) - g_L (V - E_L) \\ \dot{n} = \dfrac{(n_{\infty}(V) - n)}{\tau_n(V)} = \alpha_n(V)(1 - n) - \beta_n(V) n \end{cases}$$

Как видно из Рисунка 6, потенциал действия упрощенной системы немного превысил потенциал оригинальной.

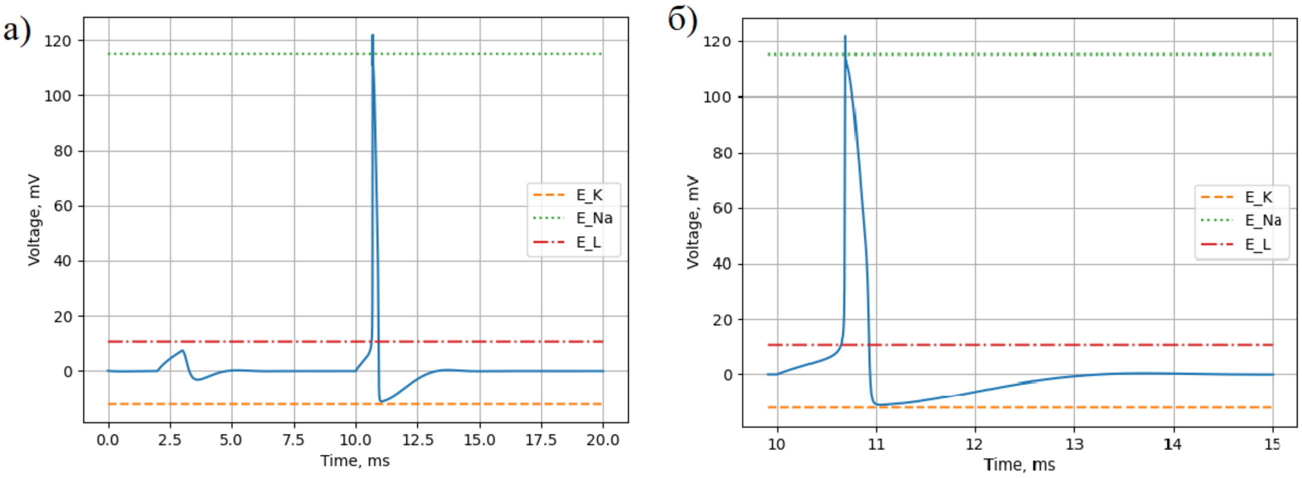

Рисунок 6 - Значения мембранного потенциала при тех же внешних токах, что и для полной модели Ходжкина-Хаксли

Для анализа фазового портрета системы была реализована математическая модель нейрона, способная наблюдать за состоянием нескольких независимых нейронов на фазовом пространстве с различными исходными состояниями. При величине тока $I = 0$ мА существует только одно состояние равновесия, причём устойчивое, и нейрон находится в состоянии покоя (Рисунок 7). При увеличении внешнего тока, например, до $I = 16$ мА можно наблюдать субкритическую бифуркацию Андронова-Хопфа, при которой исходное положение равновесие становится неустойчивым, нейрон генерирует периодические спайки, двигаясь по устойчивому предельному циклу (Рисунок 8).

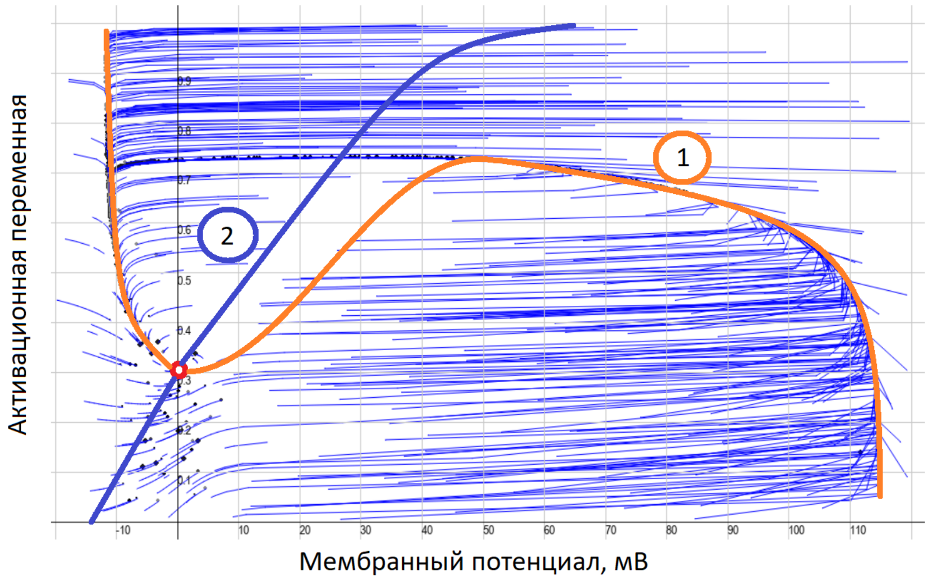

Рисунок 7 - Карта траекторий для упрощённой модели Ходжкина-Хаксли при внешнем токе $\boldsymbol{I = 0}$ мА. Линия 1 – V-нульклина, линия 2 – n-нульклина. На их пересечении можно заметить устойчивое положение равновесия – фокус.

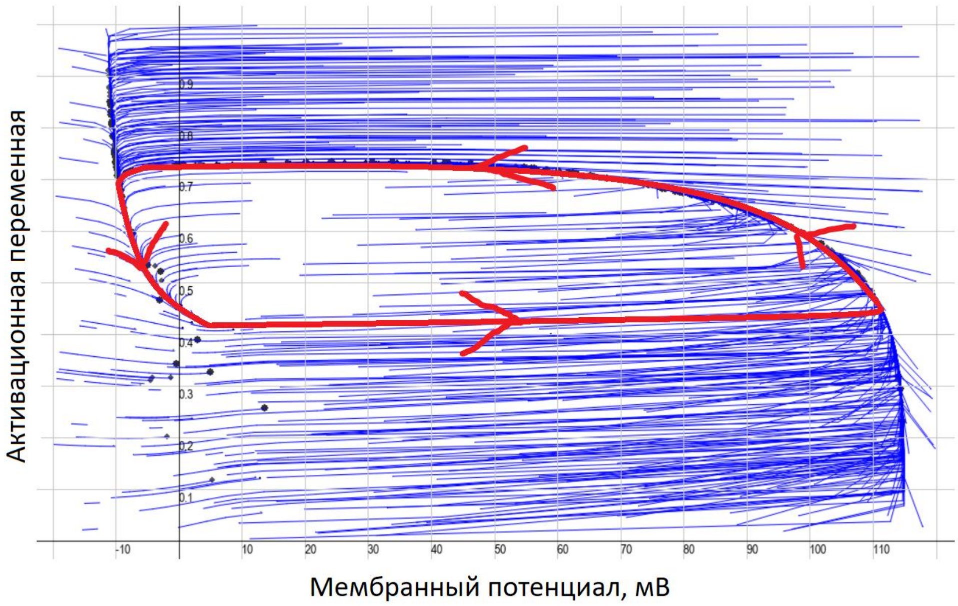

Рисунок 8 - Карта траекторий для упрощённой модели Ходжкина-Хаксли при внешнем токе $\boldsymbol{I = 16}$ мА. Фокус потерял свою устойчивость, нейрон “движется” по устойчивому предельному циклу.

2.3. Модель Фитцхью-Нагумо

Двумерная модель, имитирующая генерацию спайков моделей [5]

$$\begin{cases} \dot{V} = V(V-a)(1-V) - w + I \\ \dot{w} = \epsilon(V - \gamma w) \end{cases}$$

V – мембранный потенциал, w играет роль восстанавливающей переменной.

Параметр a задаёт окрестности порогового значения, при преодолении которого возникает спайк, ϵ и γ отвечают за кинетику переменной w. Модель масштабирована так, чтобы максимальное значения было равно 1, а устойчивое состояние равновесия при отсутствии токов находилось в нуле.

В зависимости от значения параметров a, ϵ, γ и значения внешнего тока I можно получить разное поведение нейрона. Ниже рассмотрено поведение модели при значениях параметров $a = 0.25$, $\epsilon = 0.05$ и $\gamma = 2$ при различных значениях входного тока ($I = 0;\ 0{,}3;\ 0{,}675$). V-нульклина имеет кубическую N-образную форму, w-нульклина представлена прямой.

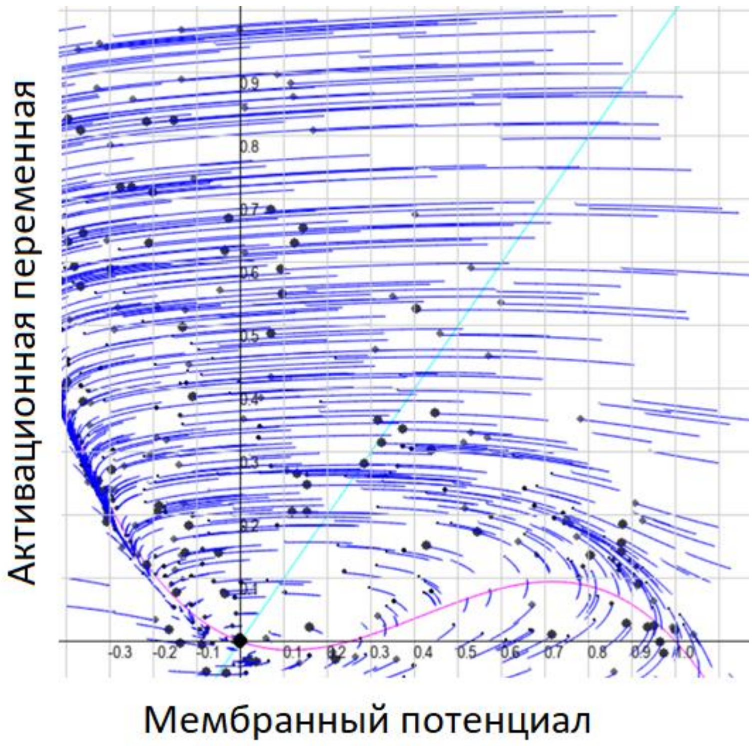

Рисунок 9 - Карта траекторий для модели Фитцхью-Нагумо при нулевом внешнем токе. В точке (0, 0) можно заметить устойчивый фокус.

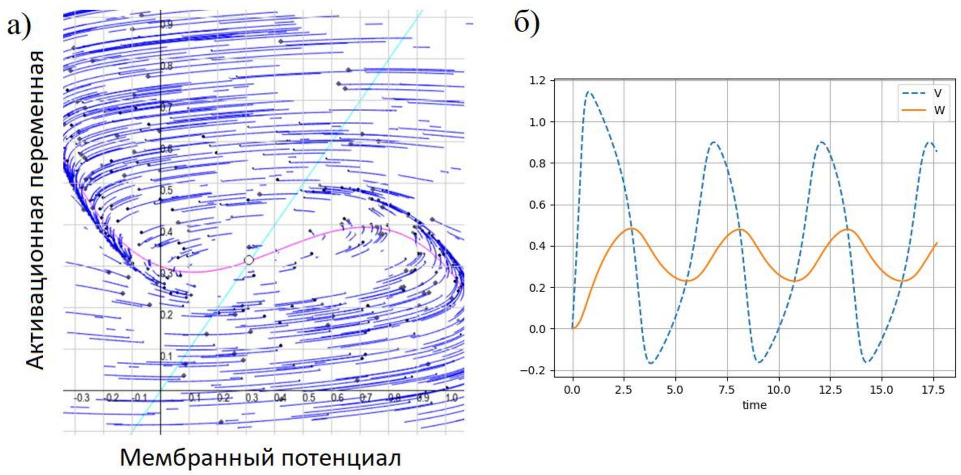

Рисунок 10 - Карта траекторий (а) и зависимость значения мембранного потеницала и активационной переменной от времени (б) при внешнем токе $\boldsymbol{I = 0,03}$

В первом случае система имеет устойчивый фокус (на пересечении нульклин), нейрон находится в состоянии покоя (Рисунок 9). При инъекции достаточного тока происходит смена фазового портрета, и система проходит через суперкритическую бифуркацию Андронова-Хопфа – фокус меняет свою устойчивость, а вокруг него образуется устойчивый предельный цикл (Рисунок 10.а). В свою очередь нейрон генерирует периодически спайки (Рисунок 10.б). При ещё большем увеличении система снова проходит через ту же бифуркацию (только в обратную сторону) и появляется устойчивый фокус, а соответственно нейрон генерирует постоянный сигнал (Рисунок 11).

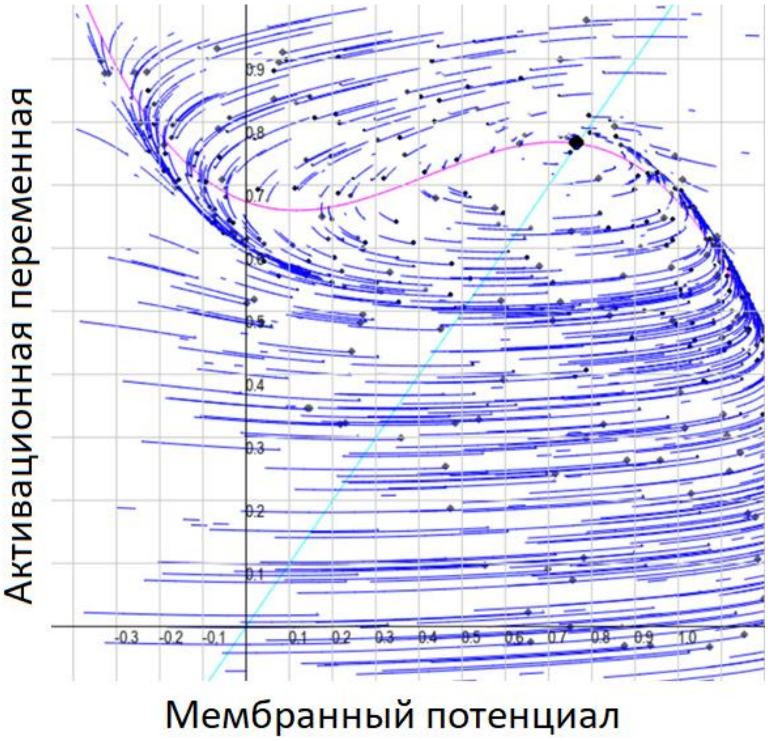

Рисунок 11 - Карта при траекторий (а)
внешнем токе I=0,675

По поведению модели можно судить, что нейрон всегда имеет может иметь только одно устойчивое состояние при фиксированном значении параметра. Также при небольших инъекциях тока наблюдаются *подпороговые колебания*, а при определённой частоте импульсов тока, можно добиться резонанса, получив периодические спайки (Рисунок 13 и Рисунок 14). По данным проявлениям нейрона можно назвать его *моностабильным резонатором*, согласно классификации нейронов, принятой в главе 7.2 книги [3]).

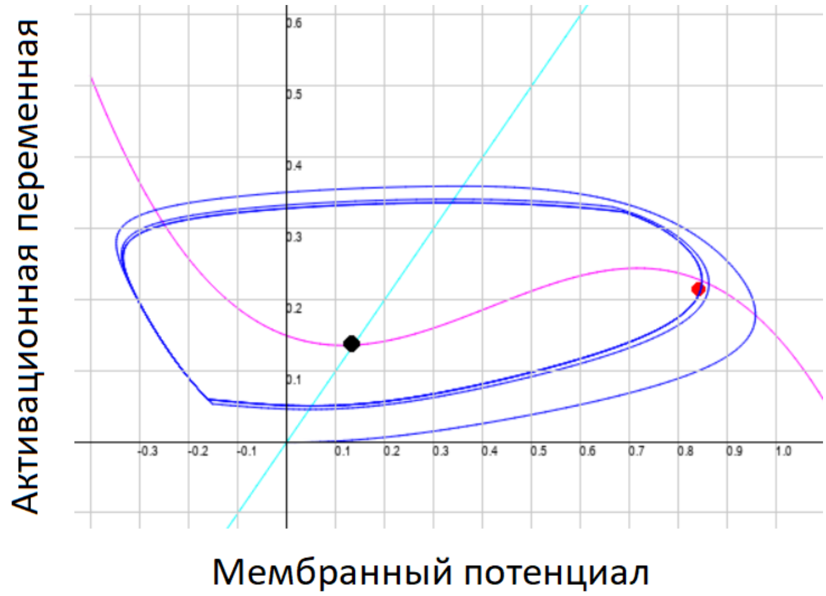

Рисунок 12 - Траектория состояния нейрона при инъекции тока I=0,15 каждые 5 мс

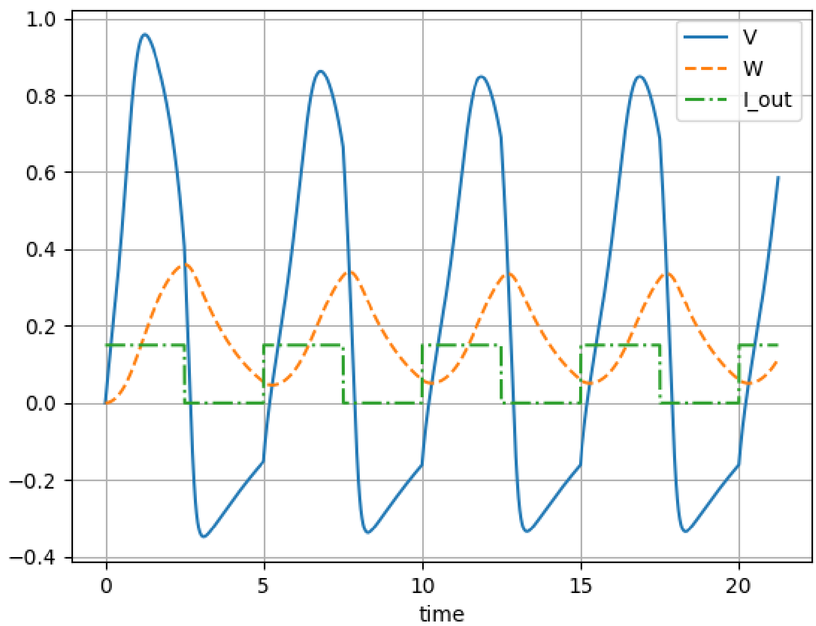

Рисунок 13 - Периодические колебания значений мембранного потенциала и активационной переменной при инъекции тока $\boldsymbol{I = 0,15}$ каждые 5 мс

2.4. Модель Ижикевича (Простая модель)

$$\begin{cases} \dot{V} = 0.04V^2 + 5v\ + 140 - w + I \\ \dot{w} = a(bV - w) \end{cases}$$

если $V \geq +30\ mV,$ то устанавливаются значения $\begin{cases} V = \ c \\ w = \ w + d \end{cases},$
и модель продолжает функционировать с новыми значенями

Аналогично модели Фитцхью-Нагумо V обозначает мембранный потенциал, w – переменную восстановления, отвечающую за активацию K^+ тока и инактивацию тока Na^+. Численные коэффициенты в первом равенстве выбраны так из-за размерности величин V (мВ) и времени (мс).

Особенность данной модели заключается в том, что благодаря регулировке параметров a, b, c, d она позволяет достаточно правдоподобно описать реальные нейроны различных фундаментальных классов по типу реагирования (стимулирующие, тормозящие, резонирующие и т.д.) [6][7].

Также, данная модель является более эффективной с точки зрения затрат вычислительных мощностей, т.к. данная модель является двумерной [8].

В совокупности два этих свойства позволяют создавать модели крупных сетей нейронов, для которой нужно отобразить большой набор нейровычислительных свойств.

Ниже представлены параметры моделей некоторых классов нейронов и их поведение.

1) Регулярноразрядные нейроны (regular spiking, RS)
$a = 0{,}02 \qquad b = 0{,}2 \qquad c = -65 \qquad d = 8$
Основной класс возбуждающих нейронов в неокортексе.
В ответ на повышение тока генерируют периодические спайки частотой пропорционально величине тока.
(Значение внешнего тока $I = 10$ мА)

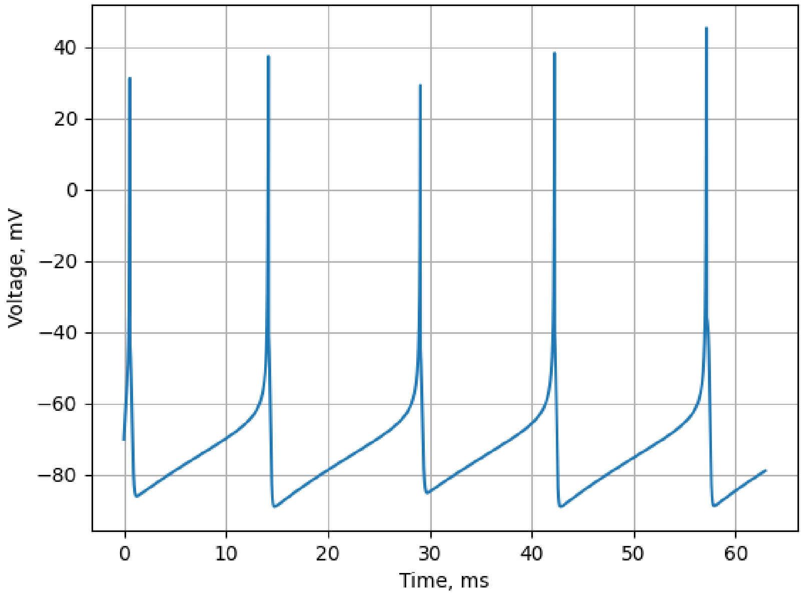

Рисунок 14 - Поведение возбужденного RS-нейрона

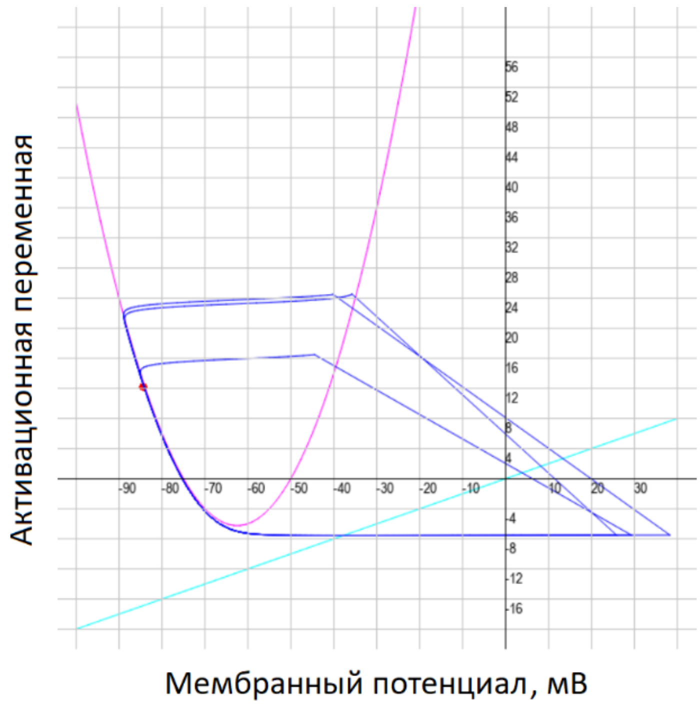

Рисунок 15 - Фазовая траектория состояния возбуждённого RS-нейрона

2) Эндогенные пачечные (intrinsically bursting, IB)
$a = 0{,}02$ $b = 0{,}2$ $c = -55$ $d = 4$

Генерируют пачки импульсов, с некоторой частотой. Значение внешнего тока $I = 10$ мА.

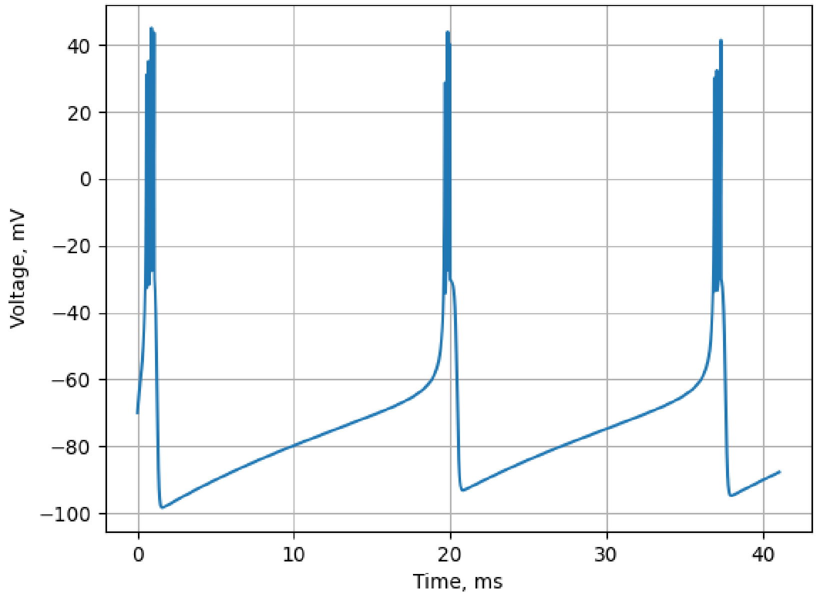

Рисунок 16 - Поведение возбужденного IB-нейрона

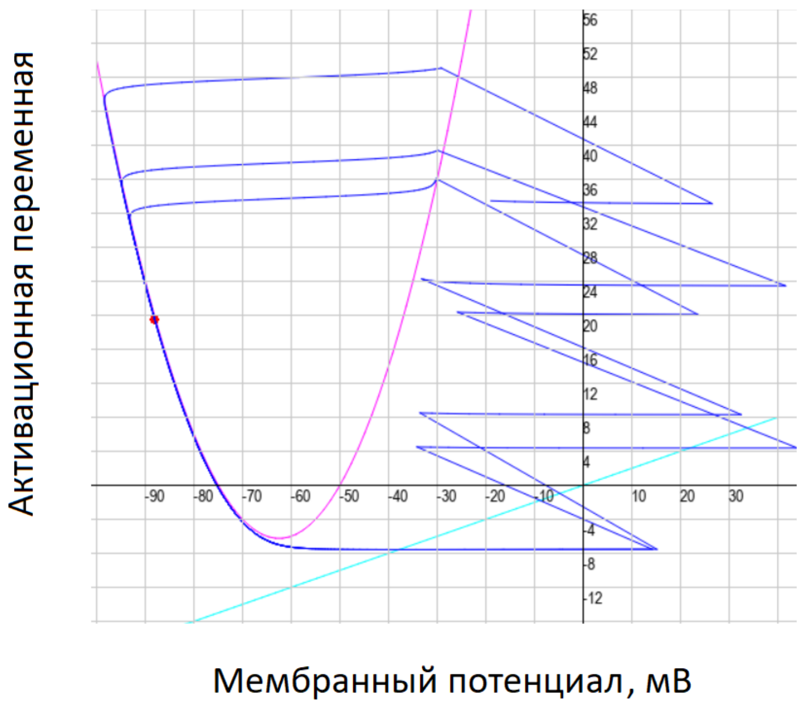

Рисунок 17 - Фазовая траектория состояния возбуждённого IB-нейрона

3) Стрекочущие (chattering, CH)
$a = 0{,}02 \qquad b = 0{,}2 \qquad c = -50 \qquad d = 2$

Генерирует высокочастотные пачки спайков с короткими межпачечным периодом.
Значение внешнего тока $I = 10$ мА.

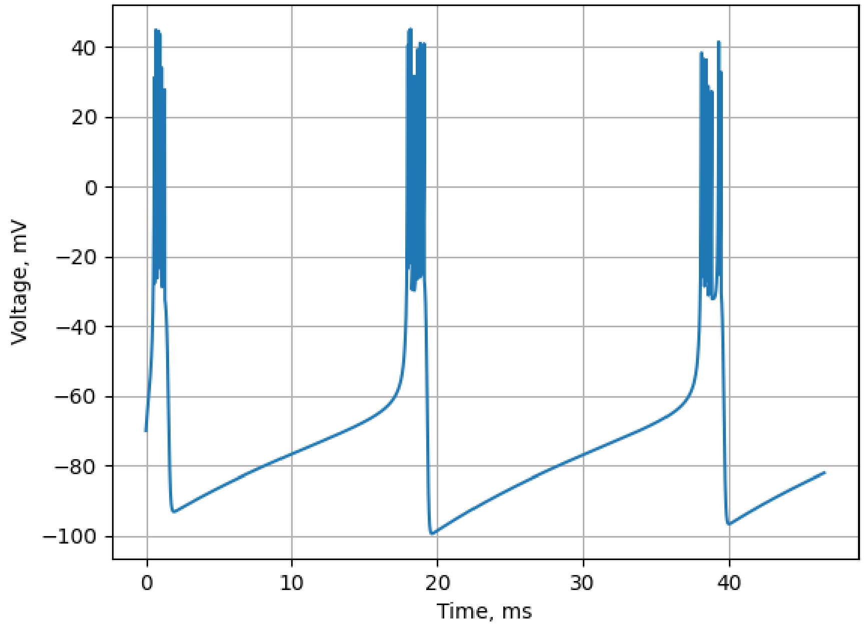

Рисунок 18 - Поведение возбужденного CH-нейрона

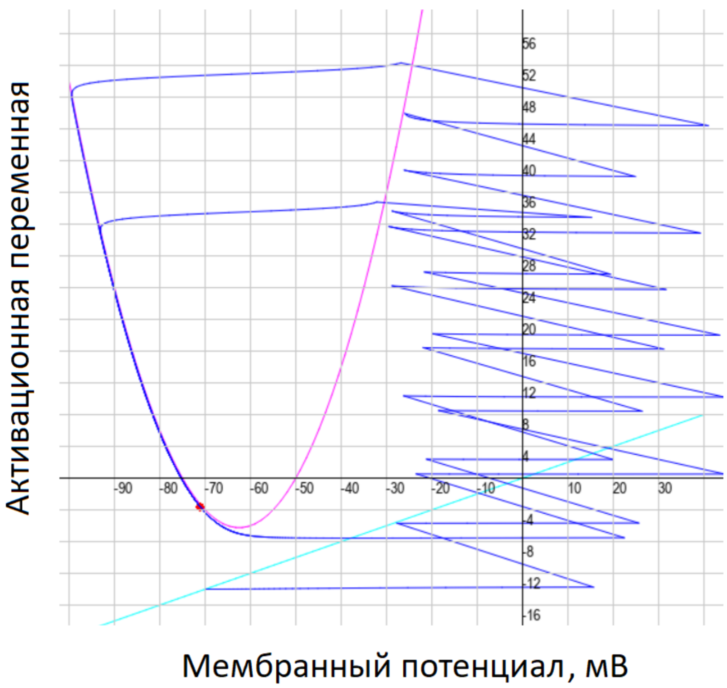

Рисунок 19 – Фазовая траектория состояния возбуждённого CH-нейрона

4) Быстроразрядные (fast spiking, FS)

$a = 0{,}1 \qquad b = 0{,}2 \qquad c = -65 \qquad d = 2$

Генерирует высокочастотные последовательности спайков с постоянным периодом.
Значение внешнего тока $I = 30$ мА.

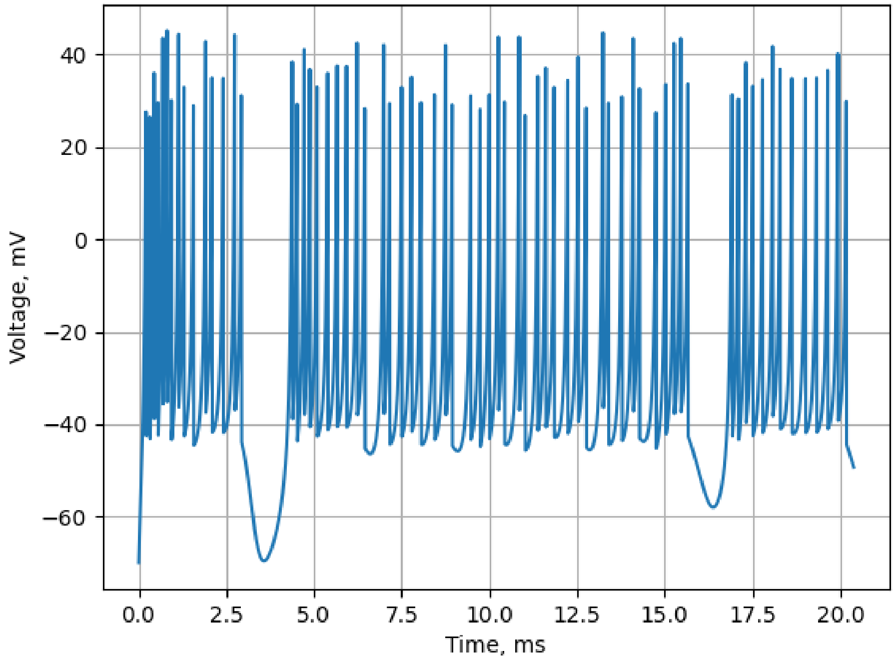

Рисунок 20 - Поведение возбужденного FS-нейрона

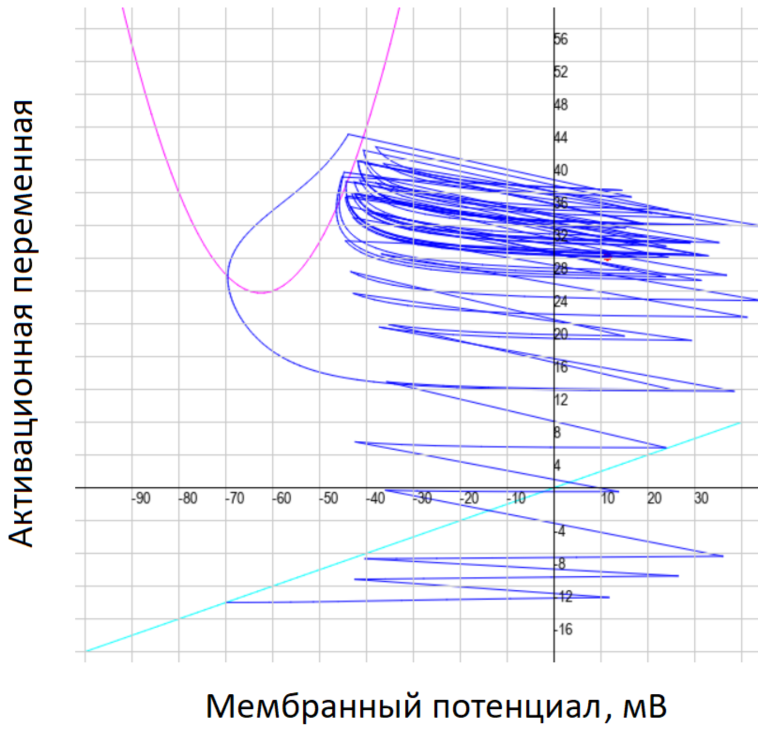

Рисунок 21 - Фазовая траектория состояния возбуждённого FS-нейрона

3. ЗАКЛЮЧЕНИЕ

Были изучены основные понятия динамические систем и базовые понятия теории бифуркации. На основе полученных знаний, были изучены динамические системы нейронов такие, как модель Ходжкина-Хаксли, модель Фитцхью-Нагумо и модель Ижикевича. С помощью дополнительных библиотек на языке Python были реализованы соответствующие математические модели нейронов с использованием численного метода Рунге-Кутты 4го порядка, демонстрирующие поведение нейрона при различных исходных состояниях, а также позволяющие интерактивно изменять модель через регулирование величины входного тока и, как следствие, наблюдать изменение поведения нейрона, в частности, качественные изменения всей системы при прохождении через бифуркацию.

СПИСОК ЛИТЕРАТУРЫ

1. Differential Equations and Dynamical Systems (Texts in Applied Mathematics, 7) - 2nd. ed. [Текст] / L. Perko // Springer-Verlag New York, Inc. — 1996. — с. 180

2. Элементы теории бифуркаций и динамических систем. Часть 1: учебно-методическое пособие по курсу Аналитическая механика / сост. А.В. Фомичев. – М.: МФТИ, 2019. – с. 4

3. Динамические системы в нейронауке. Геометрия возбудимости и пачечной активности. — М.-Ижевск: Ижевский институт компьютерных исследований, 2018. – с. 46

4. Journal of Physiology : Hodgkin A.L. and Huxley A.F. A quantitative description of membrane current and application to conduction and excitation in nerve.: 1952 — P. 500–544.

5. E.M. Izhikevich, R.FitzHugh. FitzHugh-Nagumo model [Электронный ресурс] : Scholarpedia. — Электрон. дан. — 2006. — Режим доступа: http://www.scholarpedia.org/article/FitzHugh-Nagumo_model. — Загл. с экрана. — Англ..

6. E.M. Izhikevich. Simple Model of Spiking Neurons [Электронный ресурс] : Eugene M. Izhikevich old (NSI) home page — Электрон. дан. — 2003 — Режим доступа: http://www.izhikevich.org/publications/spikes.htm . — Загл. с экрана. — Англ.

7. B.W. Connors, M.J. Gutnick. Intrinsic firing patterns of diverse neocortical neurons // Trends in Neuroscience, Volume 13, Issue 3 — 1990 — P. 99-104

8. E.M. Izhikevich. Which Model to Use for Cortical Spiking Neurons? [Электронный ресурс] : Eugene M. Izhikevich old (NSI) home page — Электрон. дан. — 2004 — Режим доступа: http://www.izhikevich.org/publications/whichmod.pdf. — Загл. с экрана. — Англ.

ПРИЛОЖЕНИЕ А

Листинг программы

HH.py (модель Ходжкина-Хаксли одиночного нейрона)

```
import pygame
import numpy as np
from matplotlib import pyplot as plt

pygame.init()
WIDTH = 1000
HEIGHT = 600
MARGIN_X = 20
MARGIN_Y = 20
WORK_WIDTH = WIDTH - MARGIN_X * 2
WORK_HEIGHT = HEIGHT - MARGIN_Y * 2
MAX_X = 20
MIN_X = -2
MAX_Y = 120
MIN_Y = -20
STEP_X = 2
STEP_Y = 10
SCALE_X = WORK_WIDTH / (MAX_X - MIN_X)
SCALE_Y = WORK_HEIGHT / (MAX_Y - MIN_Y)
# CENTER_XY = np.array([MAX_X+MIN_X, MAX_Y+MIN_Y]) / 2
CENTER_XY = np.array([0, 0])
SCALE_T = 4
PYGAME_START_TIME = 0

def init_model_update_timer():
    global PYGAME_START_TIME
    PYGAME_START_TIME = pygame.time.get_ticks()

def get_time():
    return pygame.time.get_ticks() - PYGAME_START_TIME

sc = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("FHN")
font = pygame.font.SysFont('arial', 10)
FPS = 60
```

```
clock = pygame.time.Clock()
WHITE = (255, 255, 255)
GREY = (200, 200, 200)
BLACK = (0, 0, 0)
BLUE = (0, 0, 255)
GREEN = (0, 255, 0)
RED = (255, 0, 0)

def draw_text(text, pos, col=BLACK):
    text_surface = font.render(text, True, col)
    sc.blit(text_surface, pos)

def real_to_pygame(r_cord):
    return np.array([(r_cord[0] - MIN_X) * SCALE_X + MARGIN_X, (MAX_Y - r_cord[1]) * SCALE_Y + MARGIN_Y])

def draw_net():
    pg_center = real_to_pygame(CENTER_XY)
    vert = np.append(np.arange(pg_center[0], 0, -SCALE_X * STEP_X), np.arange(pg_center[0], WIDTH, SCALE_X * STEP_X))
    horiz = np.append(np.arange(pg_center[1], 0, -SCALE_Y * STEP_Y), np.arange(pg_center[1], HEIGHT, SCALE_Y * STEP_Y))
    for v_line in vert:
        pygame.draw.line(sc, GREY, (v_line, 0), (v_line, HEIGHT))
    for h_line in horiz:
        pygame.draw.line(sc, GREY, (0, h_line), (WIDTH, h_line))
    pygame.draw.line(sc, BLACK, (pg_center[0], 0), (pg_center[0], HEIGHT))
    pygame.draw.line(sc, BLACK, (0, pg_center[1]), (WIDTH, pg_center[1]))
    for i in range(1, int((CENTER_XY[0] - MIN_X) / STEP_X)):
        draw_text(str(round(CENTER_XY[0] - i * STEP_X, 2)), (pg_center[0] - i * STEP_X * SCALE_X, pg_center[1]))
    for i in range(1, int((MAX_X - CENTER_XY[0]) / STEP_X)):
        draw_text(str(round(CENTER_XY[0] + i * STEP_X, 2)), (pg_center[0] + i * STEP_X * SCALE_X, pg_center[1]))
    for i in range(1, int((CENTER_XY[1] - MIN_Y) / STEP_Y)):
        draw_text(str(round(CENTER_XY[1] - i * STEP_Y, 2)), (pg_center[0], pg_center[1] + i * STEP_Y * SCALE_Y))
    for i in range(1, int((MAX_Y - CENTER_XY[1]) / STEP_Y)):
        draw_text(str(round(CENTER_XY[1] + i * STEP_Y, 2)), (pg_center[0], pg_center[1] - i * STEP_Y * SCALE_Y))
```

```
# ------------------------
v_0 = 0
n_0 = 0.318
m_0 = 0.053
h_0 = 0.59
x = np.array([v_0, n_0, m_0, h_0])
TV_curve = [real_to_pygame([0, x[0]])]


def get_I_app(t):
    if 2 <= t <= 3:
        return 2
    if 10 <= t <= 11:
        return 2.3
    return 0


def a_n(v):
    return 0.01 * (10 - v) / (np.exp((10 - v) / 10) - 1)


def b_n(v):
    return 0.125 * np.exp(-v / 80)


def a_m(v):
    return 0.1 * (25 - v) / (np.exp((25 - v) / 10) - 1)


def b_m(v):
    return 4 * np.exp(-v / 18)


def a_h(v):
    return 0.07 * np.exp(-v / 20)


def b_h(v):
    return 1 / (np.exp((30 - v) / 10) + 1)


C = 1
E_K = -12
E_Na = 115
E_L = 10.613
g_K = 36
g_Na = 120
```

```
g_L = 0.3

def HH(v, n, m, h, t):
    return np.array([(get_I_app(t) - g_K * (n ** 4) * (v - E_K)
                  - g_Na * (m ** 3) * h * (v - E_Na) - g_L * (v - E_L))/C,
                  a_n(v)*(1-n)-b_n(v)*n,
                  a_m(v) * (1 - m) - b_m(v) * m,
                  a_h(v) * (1 - h) - b_h(v) * h])
# -------------------------

def RK4_step(y, dt, t):
    v = y[0]
    n = y[1]
    m = y[2]
    h = y[3]
    k1, q1, w1, z1 = HH(v, n, m, h, t)
    k2, q2, w2, z2 = HH(v + 0.5 * k1 * dt, n + 0.5 * q1 * dt, m + 0.5 * w1 * dt, h + 0.5 * z1 * dt, t)
    k3, q3, w3, z3 = HH(v + 0.5 * k2 * dt, n + 0.5 * q2 * dt, m + 0.5 * w2 * dt, h + 0.5 * z2 * dt, t)
    k4, q4, w4, z4 = HH(v + k3 * dt, n + q3 * dt, m + w3 * dt, h + z3 * dt, t)
    return dt * np.array([k1 + 2 * k2 + 2 * k3 + k4, q1 + 2 * q2 + 2 * q3 + q4,
                    w1 + 2 * w2 + 2 * w3 + w4, z1 + 2 * z2 + 2 * z3 + z4])

max_time = 20
delta_t = 0.001
time_measure = np.array([0])
# time-stepping solution
V = np.array([v_0])
N = np.array([n_0])
M = np.array([m_0])
H = np.array([h_0])
I_out = np.array([get_I_app(0)])
last_upd = 0
time_from_last_update = 0
init_model_update_timer()
model_time = 0
escape = False
measure_cnt = 1
prespike_moment = 0
afterspike_moment = 0
```

```
while 1:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            exit()
        elif event.type == pygame.KEYDOWN:
            event_keys = pygame.key.get_pressed()
            if event.key == pygame.K_ESCAPE:
                escape = True
                # pygame.event.post(pygame.event.Event(pygame.QUIT))
            elif event.key == pygame.K_LEFTBRACKET:
                SCALE_T -= 0.5
                print(f"TIME SCALE = {SCALE_T}")
            elif event.key == pygame.K_RIGHTBRACKET:
                SCALE_T += 0.5
                print(f"TIME SCALE = {SCALE_T}")
    if escape or model_time >= max_time:
        break
    if get_time() < 1000*delta_t:
        continue
    init_model_update_timer()
    time_from_last_update += SCALE_T * delta_t
    model_time = time_measure[-1] + SCALE_T * delta_t
    x = x + RK4_step(x, SCALE_T * delta_t, model_time)
    V = np.append(V, x[0])
    N = np.append(N, x[1])
    M = np.append(M, x[2])
    H = np.append(H, x[3])
    I_out = np.append(I_out, get_I_app(model_time))
    time_measure = np.append(time_measure, model_time)
    measure_cnt += 1
    if prespike_moment == 0 and model_time >= 9.9:
        prespike_moment = measure_cnt
    if afterspike_moment == 0 and model_time >= 15.:
        afterspike_moment = measure_cnt
    if model_time % 1 < 2*delta_t:
        print(model_time)
        print(x)
        print(f'I={get_I_app(model_time)}')
    if time_from_last_update - last_upd >= 1 / 60:
        sc.fill(WHITE)
```

```
        last_upd = time_from_last_update
        # E_k,na,l
        pygame.draw.line(sc, BLACK, real_to_pygame([0, E_K]), real_to_pygame([max_time, E_K]), 3)
        pygame.draw.line(sc, BLACK, real_to_pygame([0, E_Na]), real_to_pygame([max_time, E_Na]), 3)
        pygame.draw.line(sc, BLACK, real_to_pygame([0, E_L]), real_to_pygame([max_time, E_L]), 3)
        # draw point
        point = real_to_pygame((model_time, x[0]))
        pygame.draw.circle(sc, RED, point, 3)
        # trajectory
        TV_curve.append([point[0], point[1]])
        pygame.draw.aalines(sc, BLUE, False, TV_curve[:measure_cnt])
        # net
        draw_net()
        pygame.display.update()
max_time = time_measure[-1]

plt.plot(time_measure, V)
plt.plot(time_measure, np.ones_like(time_measure)*E_K, label='E_K', color='red')
plt.plot(time_measure, np.ones_like(time_measure)*E_Na, '--', label='E_Na', color='orange')
plt.plot(time_measure, np.ones_like(time_measure)*E_L, '-.', label='E_L', color='green')
plt.xlabel('Time, ms')
plt.ylabel('Voltage, mV', labelpad=0)
plt.legend()
plt.grid()
plt.show()

plt.plot(time_measure, N, '--', label='n(t)')
plt.plot(time_measure, M, label='m(t)')
plt.plot(time_measure, H, '-.', label='h(t)')
plt.xlabel('Time, ms')
plt.ylabel('Activation variables', labelpad=0)
plt.legend()
plt.grid()
plt.show()

plt.plot(time_measure, (N**4)*g_K, '--', label='g_K')
plt.plot(time_measure, (M**3)*H*g_Na, label='g_Na')
plt.xlabel('Time, ms')
plt.ylabel('Conductance, mS/cm2', labelpad=0)
plt.legend()
```

```
plt.grid()
plt.show()

plt.plot(time_measure, (N**4)*g_K*(V-E_K), '--', label='I_K')
plt.plot(time_measure, (M**3)*H*g_Na*(V-E_Na), label='I_Na')
plt.plot(time_measure, (N**4)*g_K*(V-E_K) + (M**3)*H*g_Na*(V-E_Na) + g_L*(V-E_L), label='I_K + I_Na + I_L')
plt.xlabel('Time, ms')
plt.ylabel('Current, mA', labelpad=0)
plt.legend()
plt.grid()
plt.show()

plt.plot(time_measure, [get_I_app(i) for i in time_measure])
plt.xlabel('Time, ms')
plt.ylabel('I_out, mA', labelpad=0)
plt.grid()
plt.show()

# repeat plots for [prespike_moment:afterspike_moment]
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, sharex=True)

ax1.plot(time_measure[prespike_moment:afterspike_moment],
    V[prespike_moment:afterspike_moment])
ax1.plot(time_measure[prespike_moment:afterspike_moment],
    np.ones_like(time_measure[prespike_moment:afterspike_moment])*E_K, label='E_K', color='red')
ax1.plot(time_measure[prespike_moment:afterspike_moment],
    np.ones_like(time_measure[prespike_moment:afterspike_moment])*E_Na, '--', label='E_Na', color='orange')
ax1.plot(time_measure[prespike_moment:afterspike_moment],
    np.ones_like(time_measure[prespike_moment:afterspike_moment])*E_L, '-.', label='E_L', color='green')
ax1.set_ylabel('Voltage, mV', labelpad=20)
ax1.legend(loc=7)
ax1.grid()

ax2.plot(time_measure[prespike_moment:afterspike_moment],
    N[prespike_moment:afterspike_moment], label='n(t)', color='red')
ax2.plot(time_measure[prespike_moment:afterspike_moment],
    M[prespike_moment:afterspike_moment], '--', label='m(t)', color='orange')
```

```
ax2.plot(time_measure[prespike_moment:afterspike_moment],
        H[prespike_moment:afterspike_moment], '-.', label='h(t)', color='green')
ax2.plot(time_measure[prespike_moment:afterspike_moment],
        H[prespike_moment:afterspike_moment]+N[prespike_moment:afterspike_moment],                    ':',
label='n(t)+h(t)')
ax2.set_ylabel('Activation\nvariables', labelpad=10)
ax2.legend(loc=7)
ax2.grid()

ax3.plot(time_measure[prespike_moment:afterspike_moment],
        ((N**4)*g_K)[prespike_moment:afterspike_moment], label='g_K', color='red')
ax3.plot(time_measure[prespike_moment:afterspike_moment],
        ((M**3)*H*g_Na)[prespike_moment:afterspike_moment], '--', label='g_Na', color='orange')
ax3.set_ylabel('Conductance, mS/cm2', labelpad=27)
ax3.legend(loc=7)
ax3.grid()

ax4.plot(time_measure[prespike_moment:afterspike_moment],
        ((N**4)*g_K*(V-E_K))[prespike_moment:afterspike_moment], label='I_K', color='red')
ax4.plot(time_measure[prespike_moment:afterspike_moment],
        ((M**3)*H*g_Na*(V-E_Na))[prespike_moment:afterspike_moment], '--', label='I_Na', color='orange')
ax4.plot(time_measure[prespike_moment:afterspike_moment],
        ((N**4)*g_K*(V-E_K)              +              (M**3)*H*g_Na*(V-E_Na)              +              g_L*(V-
E_L))[prespike_moment:afterspike_moment],
        ':', color='green', label='I_K + I_Na + I_L')
ax4.set_xlabel('Time, ms')
ax4.set_ylabel('Current, mA', labelpad=0)
ax4.legend(loc=7)
ax4.grid()
plt.show()

plt.plot(time_measure[prespike_moment:afterspike_moment],
        [get_I_app(i) for i in time_measure[prespike_moment:afterspike_moment]])
plt.xlabel('Time, ms')
plt.ylabel('I_out, mA', labelpad=0)
plt.grid()
plt.show()

plt.plot(N[prespike_moment:afterspike_moment],
        H[prespike_moment:afterspike_moment], label='h(n)')
```

```
plt.plot(N[prespike_moment:afterspike_moment],
       [0.89 - 1.1*i for i in N[prespike_moment:afterspike_moment]], '--', label='0.89 - 1.1n')
plt.xlabel('n')
plt.ylabel('h', labelpad=0)
plt.legend()
plt.grid()
plt.show()
exit()
```

HH_reduced.py (редукция модели Ходжкина-Хаксли одиночного нейрона)

```
MAX_X = 120
MIN_X = -20
MAX_Y = 1
MIN_Y = 0
STEP_X = 10
STEP_Y = 0.1
SCALE_T = 0.5
# -------------------------


v_0 = 0
n_0 = 0.318
x = np.array([v_0, n_0])
VT_curve = [real_to_pygame([0, x[0]])]


def get_I_app(t):
   if 2 <= t <= 3:
      return 2
   if 10 <= t <= 11:
      return 2.3
   return 0


def a_n(v):
   return 0.01 * (10 - v) / (np.exp((10 - v) / 10) - 1)


def b_n(v):
   return 0.125 * np.exp(-v / 80)


def a_m(v):
```

```
    return 0.1 * (25 - v) / (np.exp((25 - v) / 10) - 1)


def b_m(v):
    return 4 * np.exp(-v / 18)


def n_inf(v):
    return a_n(v)/(a_n(v) + b_n(v))


def m_inf(v):
    return a_m(v)/(a_m(v) + b_m(v))


def tau_n(v):
    return 1/(a_n(v) + b_n(v))


C = 1
E_K = -12
E_Na = 115
E_L = 10.613
g_K = 36
g_Na = 120
g_L = 0.3


def HH_reduced(v, n, t):
    return np.array([(get_I_app(t) - g_K * (n ** 4) * (v - E_K)
                     - g_Na * (m_inf(v) ** 3) * (0.89 - 1.1*n) * (v - E_Na) - g_L * (v - E_L))/C,
                     a_n(v)*(1-n)-b_n(v)*n])
# -------------------------


def RK4_step(y, dt, t):
    v = y[0]
    n = y[1]
    k1, q1 = HH_reduced(v, n, t)
    k2, q2 = HH_reduced(v + 0.5 * k1 * dt, n + 0.5 * q1 * dt, t)
    k3, q3 = HH_reduced(v + 0.5 * k2 * dt, n + 0.5 * q2 * dt, t)
    k4, q4 = HH_reduced(v + k3 * dt, n + q3 * dt, t)
    return dt * np.array([k1 + 2 * k2 + 2 * k3 + k4, q1 + 2 * q2 + 2 * q3 + q4])


max_time = 20
delta_t = 0.001
time_measure = np.array([0])
```

```
V = np.array([v_0])
N = np.array([n_0])
I_out = np.array([get_I_app(0)])
last_upd = 0
time_from_last_update = 0
init_model_update_timer()
model_time = 0
escape = False
measure_cnt = 1
prespike_moment = 0
afterspike_moment = 0

while 1:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            exit()
        elif event.type == pygame.KEYDOWN:
            event_keys = pygame.key.get_pressed()
            if event.key == pygame.K_ESCAPE:
                escape = True
            elif event.key == pygame.K_LEFTBRACKET:
                SCALE_T -= 0.5
                print(f"TIME SCALE = {SCALE_T}")
            elif event.key == pygame.K_RIGHTBRACKET:
                SCALE_T += 0.5
                print(f"TIME SCALE = {SCALE_T}")
    if escape or model_time >= max_time:
        break
    if get_time() < 1000*delta_t:
        continue
    init_model_update_timer()
    time_from_last_update += SCALE_T * delta_t
    model_time = time_measure[-1] + SCALE_T * delta_t
    x = x + RK4_step(x, SCALE_T * delta_t, model_time)
    V = np.append(V, x[0])
    N = np.append(N, x[1])
    I_out = np.append(I_out, get_I_app(model_time))
    time_measure = np.append(time_measure, model_time)
    measure_cnt += 1
    if prespike_moment == 0 and model_time >= 9.9:
```

```
            prespike_moment = measure_cnt
        if afterspike_moment == 0 and model_time >= 15.:
            afterspike_moment = measure_cnt
        if model_time % 1 < 2*delta_t:
            print(model_time)
            print(x)
            print(f'I={get_I_app(model_time)}')
        if time_from_last_update - last_upd >= 1 / 60:
            sc.fill(WHITE)
            last_upd = time_from_last_update
            # E_k,na,l
            pygame.draw.line(sc, BLACK, real_to_pygame([0, E_K]), real_to_pygame([max_time, E_K]), 3)
            pygame.draw.line(sc, BLACK, real_to_pygame([0, E_Na]), real_to_pygame([max_time, E_Na]), 3)
            pygame.draw.line(sc, BLACK, real_to_pygame([0, E_L]), real_to_pygame([max_time, E_L]), 3)
            # draw point
            point = real_to_pygame((model_time, x[0]))
            pygame.draw.circle(sc, RED, point, 3)
            # trajectory
            VT_curve.append([point[0], point[1]])
            pygame.draw.aalines(sc, BLUE, False, VT_curve[:measure_cnt])
            # net
            draw_net()
            pygame.display.update()
    max_time = time_measure[-1]
    # plot the result
    plt.plot(time_measure, V)
    plt.plot(time_measure, np.ones_like(time_measure)*E_K, '--', label='E_K')
    plt.plot(time_measure, np.ones_like(time_measure)*E_Na, ':', label='E_Na')
    plt.plot(time_measure, np.ones_like(time_measure)*E_L, '-.', label='E_L')
    plt.xlabel('Time, ms')
    plt.ylabel('Voltage, mV', labelpad=0)
    plt.legend()
    plt.grid()
    plt.show()

    # repeat plots for [prespike_moment:]
    plt.plot(time_measure[prespike_moment:afterspike_moment],
             V[prespike_moment:afterspike_moment])
    plt.plot(time_measure[prespike_moment:afterspike_moment],
             np.ones_like(time_measure[prespike_moment:afterspike_moment])*E_K, '--', label='E_K')
```

```
plt.plot(time_measure[prespike_moment:afterspike_moment],
    np.ones_like(time_measure[prespike_moment:afterspike_moment])*E_Na, ':', label='E_Na')
plt.plot(time_measure[prespike_moment:afterspike_moment],
    np.ones_like(time_measure[prespike_moment:afterspike_moment])*E_L, '-.', label='E_L')
plt.xlabel('Time, ms')
plt.ylabel('Voltage, mV', labelpad=0)
plt.legend()
plt.grid()
plt.show()

exit()
```

HH_reduced_many.py (редукция модели Ходжкина-Хаксли совокупности нейронов)

```
# -------------------------
dot_cnt = 1000
v_0 = 0
n_0 = 0.318
x = np.zeros((dot_cnt, 2))
rads = np.ones(dot_cnt)*3
curves_dot_cnt = 10
VN_curves = np.array([[real_to_pygame(xx) for i in range(curves_dot_cnt)] for xx in x])
I_const = 0.0
I_impulse_flag = False
I_per = 23
I_last_imp = 0.0
I_incr = True
I_impulse_val = 0.15

def get_I_app(t):
    global I_last_imp
    if not I_impulse_flag:
        return I_const
    if t - I_last_imp > I_per:
        I_last_imp = t
    if t - I_last_imp < I_per/2:
        return I_const + I_impulse_val
    else:
        return I_const

def a_n(v):
```

```
    return 0.01 * (10 - v) / (np.exp((10 - v) / 10) - 1)

def b_n(v):
    return 0.125 * np.exp(-v / 80)

def a_m(v):
    return 0.1 * (25 - v) / (np.exp((25 - v) / 10) - 1)

def b_m(v):
    return 4 * np.exp(-v / 18)

def n_inf(v):
    return a_n(v)/(a_n(v) + b_n(v))

def m_inf(v):
    return a_m(v)/(a_m(v) + b_m(v))

def tau_n(v):
    return 1/(a_n(v) + b_n(v))

C = 1
E_K = -12
E_Na = 115
E_L = 10.613
g_K = 36
g_Na = 120
g_L = 0.3

def HH_reduced(v, n, t):
    return np.array([(get_I_app(t) - g_K * (n ** 4) * (v - E_K)
                    - g_Na * (m_inf(v) ** 3) * (0.89 - 1.1*n) * (v - E_Na) - g_L * (v - E_L))/C,
                    a_n(v)*(1-n)-b_n(v)*n])

def respawn_dots(i, dots_gens):
    low_b = int(i/dots_gens * dot_cnt)
    up_b = int(min((i+1)/dots_gens * dot_cnt, dot_cnt))
    x[low_b:up_b, 0] = np.random.uniform(MIN_X, MAX_X, up_b-low_b)
    x[low_b:up_b, 1] = np.random.uniform(MIN_Y, MAX_Y, up_b-low_b)
    rads[low_b:up_b] = np.ones(up_b-low_b)*(i+1)*5/dots_gens + 1
    VN_curves[low_b:up_b] = np.array([[real_to_pygame(xx) for i in range(curves_dot_cnt)] for xx in x[low_b:up_b]])
# -------------------------
```

```
def RK4_step(y, dt, t):
    v = y[:, 0]
    n = y[:, 1]
    k1, q1 = HH_reduced(v, n, t)
    k2, q2 = HH_reduced(v + 0.5 * k1 * dt, n + 0.5 * q1 * dt, t)
    k3, q3 = HH_reduced(v + 0.5 * k2 * dt, n + 0.5 * q2 * dt, t)
    k4, q4 = HH_reduced(v + k3 * dt, n + q3 * dt, t)
    return dt * np.array([k1 + 2 * k2 + 2 * k3 + k4, q1 + 2 * q2 + 2 * q3 + q4]).T

max_time = 40
delta_t = 0.001
time_measure = np.array([0])
last_upd = 0
time_from_last_update = 0
init_model_update_timer()
model_time = 0
escape = False
dots_lifetime = 2000
dots_generations = 10
for i in range(dots_generations):
    respawn_dots(i, dots_generations)
last_respawn = [pygame.time.get_ticks() - dots_lifetime*i/dots_generations for i in range(dots_generations)]
while 1:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            exit()
        elif event.type == pygame.KEYDOWN:
            event_keys = pygame.key.get_pressed()
            if event.key == pygame.K_UP:
                if event_keys[pygame.K_LSHIFT]:
                    I_impulse_val += 0.075
                    print(f"I_impulse_val = {I_impulse_val}")
                elif event_keys[pygame.K_LCTRL]:
                    I_per += 0.5
                    print(f"I_per = {I_per}")
                else:
                    # I_const += 1
                    I_const += 10
                    print(f"I_const = {I_const}")
            elif event.key == pygame.K_DOWN:
```

```
                if event_keys[pygame.K_LSHIFT]:
                    I_impulse_val -= 0.075
                    print(f"I_impulse_val = {I_impulse_val}")
                elif event_keys[pygame.K_LCTRL]:
                    I_per -= 0.5
                    print(f"I_per = {I_per}")
                else:
                    # I_const -= 1
                    I_const -= 10
                    print(f"I_const = {I_const}")
            elif event.key == pygame.K_SPACE:
                I_impulse_flag = not I_impulse_flag
            elif event.key == pygame.K_ESCAPE:
                pygame.event.post(pygame.event.Event(pygame.QUIT))
            elif event.key == pygame.K_LEFTBRACKET:
                SCALE_T -= 0.1
                print(f"TIME SCALE = {SCALE_T}")
            elif event.key == pygame.K_RIGHTBRACKET:
                SCALE_T += 0.1
                print(f"TIME SCALE = {SCALE_T}")
    if get_time() < 1000*delta_t:
        continue
    init_model_update_timer()
    time_from_last_update += SCALE_T * delta_t
    model_time = time_measure[-1] + SCALE_T * delta_t
    x = x + RK4_step(x, SCALE_T * delta_t, model_time)
    time_measure = np.append(time_measure, model_time)
    for i in range(dots_generations):
        if pygame.time.get_ticks()-last_respawn[i] >= dots_lifetime:
            last_respawn[i] = pygame.time.get_ticks()
            respawn_dots(i, dots_generations)
    if time_from_last_update - last_upd >= 1 / 60:
        sc.fill(WHITE)
        last_upd = time_from_last_update
        # draw points
        for i in range(dots_generations):
            low_b = int(i / dots_generations * dot_cnt)
            up_b = int(min((i + 1) / dots_generations * dot_cnt, dot_cnt))
            dot_rad = np.sin(np.pi * (pygame.time.get_ticks() - last_respawn[i]) / dots_lifetime) * rads[low_b]
            dot_col = min(
                max(255 - np.sin(np.pi * (pygame.time.get_ticks() - last_respawn[i]) / dots_lifetime) * 255, 0),
255)
```

```
            for j in range(low_b, up_b):
                point = real_to_pygame(x[j])
                pygame.draw.circle(sc, (dot_col, dot_col, dot_col), point, dot_rad)
        # trajectory
        for i in range(dot_cnt):
            VN_curves[i][:-1] = VN_curves[i][1:]
            VN_curves[i][-1] = real_to_pygame((x[i][0], x[i][1]))
            pygame.draw.aalines(sc, BLUE, False, VN_curves[i])
        # net
        draw_net()
        pygame.display.update()
max_time = time_measure[-1]
exit()
```

FHN.py (модель Фитцхью-Нагумо одиночного нейрона)

```
WIDTH = 600
HEIGHT = 600
MARGIN_X = 20
MARGIN_Y = 20
MAX_X = 1.2
MIN_X = -0.4
MAX_Y = 1.0
MIN_Y = -0.1
STEP_X = 0.1
STEP_Y = 0.1
SCALE_T = 5

# -------------------------
a = 0.25
eps = 0.05
I_const = 0.0
v_0 = 0
w_0 = 0
gamma = 1
x = np.array([v_0, w_0])
curve_dot_cnt = 2000
VW_curve = [real_to_pygame(x) for i in range(curve_dot_cnt)]
I_impulse_flag = True
I_per = 5
I_last_imp = 0.0
I_incr = True
```

```
I_impulse_val = 0.15

def get_I_app(t):
    global I_last_imp
    if not I_impulse_flag:
        return I_const
    if t - I_last_imp > I_per:
        I_last_imp = t
    if t - I_last_imp < I_per/2:
        return I_const + I_impulse_val
    else:
        return I_const

def FHN(v, w, t):
    return np.array([v * (1 - v) * (v - a) - w + get_I_app(t), eps * (v - gamma * w)])

def get_equilibrium_points(t):
    poly = [-gamma**3, (a+1)*(gamma**2), -(a*gamma + 1), get_I_app(t)]
    eq_points_w = np.roots(poly)
    res = []
    for eq_w in eq_points_w:
        if np.isreal(eq_w) and MAX_X >= eq_w >= MIN_X:
            eq_point = (eq_w*gamma, eq_w)
            res.extend([eq_point])
    return res
# -------------------------

def RK4_step(y, dt, t):
    v = y[0]
    w = y[1]
    [k1, q1] = FHN(v, w, t)
    [k2, q2] = FHN(v + 0.5 * k1 * dt, w + 0.5 * q1 * dt, t)
    [k3, q3] = FHN(v + 0.5 * k2 * dt, w + 0.5 * q2 * dt, t)
    [k4, q4] = FHN(v + k3 * dt, w + q3 * dt, t)
    return dt * np.array([k1 + 2 * k2 + 2 * k3 + k4, q1 + 2 * q2 + 2 * q3 + q4])

max_time = 40
delta_t = 0.001
time_measure = np.array([0])
V = np.array([v_0])
W = np.array([w_0])
I_out = np.array([get_I_app(0)])
```

```
last_upd = 0
time_from_last_update = 0
init_model_update_timer()
model_time = 0
escape = False
while 1:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            escape = True
        elif event.type == pygame.KEYDOWN:
            event_keys = pygame.key.get_pressed()
            if event.key == pygame.K_UP:
                if event_keys[pygame.K_LSHIFT]:
                    I_impulse_val += 0.075
                    print(f"I_impulse_val = {I_impulse_val}")
                elif event_keys[pygame.K_LCTRL]:
                    I_per += 0.5
                    print(f"I_per = {I_per}")
                else:
                    I_const += 0.075
                    print(f"I_const = {I_const}")
            elif event.key == pygame.K_DOWN:
                if event_keys[pygame.K_LSHIFT]:
                    I_impulse_val -= 0.075
                    print(f"I_impulse_val = {I_impulse_val}")
                elif event_keys[pygame.K_LCTRL]:
                    I_per -= 0.5
                    print(f"I_per = {I_per}")
                else:
                    I_const -= 0.075
                    print(f"I_const = {I_const}")
            elif event.key == pygame.K_SPACE:
                if not event_keys[pygame.K_LSHIFT]:
                    I_impulse_flag = not I_impulse_flag
                else:
                    x[0] += a * 1.3
            elif event.key == pygame.K_ESCAPE:
                pygame.event.post(pygame.event.Event(pygame.QUIT))
            elif event.key == pygame.K_LEFTBRACKET:
                SCALE_T -= 0.5
                print(f"TIME SCALE = {SCALE_T}")
            elif event.key == pygame.K_RIGHTBRACKET:
```

```
                SCALE_T += 0.5
                print(f"TIME SCALE = {SCALE_T}")
        if escape:
            break
        if get_time() < delta_t * 1000:
            continue
        init_model_update_timer()
        time_from_last_update += SCALE_T * delta_t
        model_time = time_measure[-1] + SCALE_T * delta_t
        x = x + RK4_step(x, SCALE_T * delta_t, model_time)
        V = np.append(V, x[0])
        W = np.append(W, x[1])
        I_out = np.append(I_out, get_I_app(model_time))
        time_measure = np.append(time_measure, model_time)
        if time_from_last_update - last_upd >= 1 / 60:
            sc.fill(WHITE)
            last_upd = time_from_last_update
            # nullclines (v, w)
            v_nullcl = []
            w_nullcl = []
            for v in np.linspace(MIN_X, MAX_X, 100):
                v_nullcl.extend([real_to_pygame([v, v*(1-v)*(v-a) + get_I_app(model_time)])])
                w_nullcl.extend([real_to_pygame([v, v/gamma])])
            pygame.draw.aalines(sc, (255, 0, 255), False, v_nullcl)
            pygame.draw.aalines(sc, (0, 255, 255), False, w_nullcl)
            # eq points
            eq_points = get_equilibrium_points(model_time)
            for eq_p in eq_points:
                f_v = np.poly1d([-gamma**3, (a+1)*gamma**2, -a*gamma, (get_I_app(model_time) - eq_p[1])]).deriv()(eq_p[0])
                f_w = np.poly1d([-gamma**3, (a+1)*(gamma**2), -(a*gamma + 1), get_I_app(model_time)]).deriv()(eq_p[1])
                g_v = np.poly1d([eps, -eps*gamma*eq_p[1]]).deriv()(eq_p[0])
                g_w = np.poly1d([-eps*gamma, eps*eq_p[0]]).deriv()(eq_p[1])
                eig_val, eig_vec = np.linalg.eig([[f_v, f_w], [g_v, g_w]])
                is_stable = np.real(eig_val[0]) < 0 and np.real(eig_val[1]) < 0
                pygame.draw.circle(sc, BLACK, real_to_pygame(eq_p), 6)
                if is_stable:
                    pygame.draw.circle(sc, BLACK, real_to_pygame(eq_p), 5)
                else:
                    pygame.draw.circle(sc, WHITE, real_to_pygame(eq_p), 5)
            # draw point
```

```
        point = real_to_pygame(x)
        pygame.draw.circle(sc, RED, point, 5)
        # trajectory
        VW_curve.pop(0)
        VW_curve.append([point[0], point[1]])
        # print(len(VW_curve))
        pygame.draw.aalines(sc, BLUE, False, VW_curve[-curve_dot_cnt:])
        # net
        draw_net()
        pygame.display.update()
max_time = time_measure[-1]
plt.plot(time_measure, V, label='V')
plt.plot(time_measure, W, '--', label='W')
plt.plot(time_measure, I_out, '-.', label='I_out')
plt.grid(True)
plt.axis()
plt.xlabel('time')
plt.legend(loc=1)
plt.show()
exit()
```

FHN_many.py (модель Фитцхью-Нагумо совокупности нейронов)

```
# -------------------------
dot_cnt = 1000
a = 0.25
eps = 0.05
I_app = 0.0
v_0 = 0
w_0 = 0
gamma = 1
x = np.zeros((dot_cnt, 2))
rads = np.ones(dot_cnt)*3
curves_dot_cnt = 10
VW_curves = np.array([[real_to_pygame(xx) for i in range(curves_dot_cnt)] for xx in x])

def FHN(v, w):
    return np.array([v * (1 - v) * (v - a) - w + I_app, eps * (v - gamma * w)])

def get_equilibrium_points():
    poly = [-gamma**3, (a+1)*(gamma**2), -(a*gamma + 1), I_app]
    eq_points_w = np.roots(poly)
```

```
    res = []
    for eq_w in eq_points_w:
        if np.isreal(eq_w) and MAX_X >= eq_w >= MIN_X:
            eq_point = (eq_w*gamma, eq_w)
            res.extend([eq_point])
    return res

def respawn_dots(i, dots_gens):
    low_b = int(i/dots_gens * dot_cnt)
    up_b = int(min((i+1)/dots_gens * dot_cnt, dot_cnt))
    x[low_b:up_b, 0] = np.random.uniform(MIN_X, MAX_X, up_b-low_b)
    x[low_b:up_b, 1] = np.random.uniform(MIN_Y, MAX_Y, up_b-low_b)
    rads[low_b:up_b] = np.ones(up_b-low_b)*(i+1)*5/dots_gens + 1
    VW_curves[low_b:up_b] = np.array([[real_to_pygame(xx) for i in range(curves_dot_cnt)] for xx in x[low_b:up_b]])
# -------------------------

def RK4_step(y, dt):
    v = y[:, 0]
    w = y[:, 1]
    k1, q1 = FHN(v, w)
    [k2, q2] = FHN(v + 0.5 * k1 * dt, w + 0.5 * q1 * dt)
    [k3, q3] = FHN(v + 0.5 * k2 * dt, w + 0.5 * q2 * dt)
    [k4, q4] = FHN(v + k3 * dt, w + q3 * dt)
    return dt * np.array([k1 + 2 * k2 + 2 * k3 + k4, q1 + 2 * q2 + 2 * q3 + q4]).T

max_time = 40
delta_t = 0.001
time_measure = np.array([0])
last_upd = 0
time_from_last_update = 0
init_model_update_timer()
model_time = 0
escape = False
dots_lifetime = 2000
dots_generations = 10
for i in range(dots_generations):
    respawn_dots(i, dots_generations)
last_respawn = [pygame.time.get_ticks() - dots_lifetime*i/dots_generations for i in range(dots_generations)]

while 1:
```

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        exit()
    elif event.type == pygame.KEYDOWN:
        if event.key == pygame.K_UP:
            I_app += 0.075
            print(f"I_app = {I_app}")
        elif event.key == pygame.K_DOWN:
            I_app -= 0.075
            print(f"I_app = {I_app}")
        elif event.key == pygame.K_SPACE:
            print("dV")
            x[:, 0] += a * 0.7
            event_keys = pygame.key.get_pressed()
            if not event_keys[pygame.K_LSHIFT]:
                x[:, 0] += a * 0.7
        elif event.key == pygame.K_ESCAPE:
            pygame.event.post(pygame.event.Event(pygame.QUIT))
        elif event.key == pygame.K_LEFTBRACKET:
            SCALE_T -= 0.5
            print(f"TIME SCALE = {SCALE_T}")
        elif event.key == pygame.K_RIGHTBRACKET:
            SCALE_T += 0.5
            print(f"TIME SCALE = {SCALE_T}")
if get_time() < delta_t * 1000:
    continue
init_model_update_timer()
time_from_last_update += SCALE_T * delta_t
model_time = time_measure[-1] + SCALE_T * delta_t
x = x + RK4_step(x, SCALE_T * delta_t)
time_measure = np.append(time_measure, model_time)
for i in range(dots_generations):
    if pygame.time.get_ticks()-last_respawn[i] >= dots_lifetime:
        last_respawn[i] = pygame.time.get_ticks()
        respawn_dots(i, dots_generations)
if time_from_last_update - last_upd >= 1 / 60:
    sc.fill(WHITE)
    last_upd = time_from_last_update
    # nullclines (v, w)
    v_nullcl = []
    w_nullcl = []
    for v in np.linspace(MIN_X, MAX_X, 100):
```

```
            v_nullcl.extend([real_to_pygame([v, v*(1-v)*(v-a) + I_app])])
            w_nullcl.extend([real_to_pygame([v, v/gamma])])
        pygame.draw.aalines(sc, (255, 0, 255), False, v_nullcl)
        pygame.draw.aalines(sc, (0, 255, 255), False, w_nullcl)
        # eq points
        eq_points = get_equilibrium_points()
        for eq_p in eq_points:
            f_v = np.poly1d([-gamma**3, (a+1)*gamma**2, -a*gamma, (I_app - eq_p[1])]).deriv()(eq_p[0])
            f_w = np.poly1d([-gamma**3, (a+1)*(gamma**2), -(a*gamma + 1), I_app]).deriv()(eq_p[1])
            g_v = np.poly1d([eps, -eps * gamma * eq_p[1]]).deriv()(eq_p[0])
            g_w = np.poly1d([-eps * gamma, eps * eq_p[0]]).deriv()(eq_p[1])
            eig_val, eig_vec = np.linalg.eig([[f_v, f_w], [g_v, g_w]])
            # print(eig_val)
            is_stable = np.real(eig_val[0]) < 0 and np.real(eig_val[1]) < 0
            pygame.draw.circle(sc, BLACK, real_to_pygame(eq_p), 6)
            if is_stable:
                pygame.draw.circle(sc, BLACK, real_to_pygame(eq_p), 5)
            else:
                pygame.draw.circle(sc, WHITE, real_to_pygame(eq_p), 5)
        # draw points
        for i in range(dots_generations):
            low_b = int(i / dots_generations * dot_cnt)
            up_b = int(min((i + 1) / dots_generations * dot_cnt, dot_cnt))
            dot_rad = np.sin(np.pi*(pygame.time.get_ticks() - last_respawn[i]) / dots_lifetime) * rads[low_b]
            dot_col = min(max(255-np.sin(np.pi*(pygame.time.get_ticks() - last_respawn[i]) / 
dots_lifetime)*255, 0), 255)
            for j in range(low_b, up_b):
                point = real_to_pygame(x[j])
                pygame.draw.circle(sc, (dot_col, dot_col, dot_col), point, dot_rad)
        # trajectory
        for i in range(dot_cnt):
            VW_curves[i][:-1] = VW_curves[i][1:]
            VW_curves[i][-1] = real_to_pygame((x[i][0], x[i][1]))
            pygame.draw.aalines(sc, BLUE, False, VW_curves[i])
        # net
        draw_net()
        pygame.display.update()
```

IZH.py (модель Ижикевича одиночного нейрона)

```
MAX_X = 40
MIN_X = -100
MAX_Y = 60
```

```
MIN_Y = -20
STEP_X = 10
STEP_Y = 4
SCALE_T = 10

# ------------------------
curr_type = 0
# Excitatory: [RS, IB, CH]
# Inhibitory: [FS, LTS]
# Thalamo- : [TC]
# Rezonator : [RZ]
a = [0.02, 0.02, 0.02, 0.1, 0.02, 0.02, 0.1]
b = [0.2, 0.2, 0.2, 0.2, 0.25, 0.25, 0.25]
c = [-65., -55., -50., -65., -65., -65., -65.]
d = [8., 4., 2., 2., 2., 0.05, 2.]
I_app = 30
v_0 = -70
w_0 = -14
v_thresh = 30.0
x = np.array([v_0, w_0])
curve_dot_cnt = 2000
VW_curve = [real_to_pygame(x) for i in range(curve_dot_cnt)]

def IZH(v, w):
    if v >= v_thresh:
        x[0] = c[curr_type]
        x[1] = x[1] + d[curr_type]
    return np.array([0.04 * v**2 + 5 * v + 140 - w + I_app, a[curr_type] * (b[curr_type] * v - w)])

def change_model_type(new_type):
    global curr_type
    curr_type = new_type

def get_equilibrium_points():
    poly = [0.04, (5-b[curr_type]), (140+I_app)]
    eq_points_v = np.roots(poly)
    res = []
    for eq_v in eq_points_v:
        if np.isreal(eq_v) and MAX_X >= eq_v >= MIN_X:
            eq_point = (eq_v, b[curr_type]*eq_v)
            res.extend([eq_point])
    return res
```

```
# -------------------------

def RK4_step(y, dt):
    v = y[0]
    w = y[1]
    [k1, q1] = IZH(v, w)
    [k2, q2] = IZH(v + 0.5 * k1 * dt, w + 0.5 * q1 * dt)
    [k3, q3] = IZH(v + 0.5 * k2 * dt, w + 0.5 * q2 * dt)
    [k4, q4] = IZH(v + k3 * dt, w + q3 * dt)
    return dt * np.array([k1 + 2 * k2 + 2 * k3 + k4, q1 + 2 * q2 + 2 * q3 + q4])

max_time = 40
delta_t = 0.001
time_measure = np.array([0])
V = np.array([v_0])
W = np.array([w_0])
last_upd = 0
time_from_last_update = 0
init_model_update_timer()
model_time = 0
escape = False

while 1:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            escape = True
        elif event.type == pygame.KEYDOWN:
            event_keys = pygame.key.get_pressed()
            if event.key == pygame.K_UP:
                I_app += 1
                if event_keys[pygame.K_LSHIFT]:
                    I_app += 4
                print(f"I_app = {I_app}")
            elif event.key == pygame.K_DOWN:
                I_app -= 1
                if event_keys[pygame.K_LSHIFT]:
                    I_app -= 4
                print(f"I_app = {I_app}")
            elif event.key == pygame.K_ESCAPE:
                pygame.event.post(pygame.event.Event(pygame.QUIT))
            elif event.key == pygame.K_LEFTBRACKET:
                SCALE_T -= 2
```

```
            print(f"TIME SCALE = {SCALE_T}")
        elif event.key == pygame.K_RIGHTBRACKET:
            SCALE_T += 2
            print(f"TIME SCALE = {SCALE_T}")
        elif event.key == pygame.K_1:
            change_model_type(0)
            print("Excitatory cortical neuron model.\n"
                  "Regular spiking.")
        elif event.key == pygame.K_2:
            change_model_type(1)
            print("Excitatory cortical neuron model.\n"
                  "Instrinsically bursting.")
        elif event.key == pygame.K_3:
            change_model_type(2)
            print("Excitatory cortical neuron model.\n"
                  "Chattering.")
        elif event.key == pygame.K_4:
            change_model_type(3)
            print("Inhibitotary cortical neuron model.\n"
                  "Fast spiking.")
        elif event.key == pygame.K_5:
            change_model_type(4)
            print("Inhibitotary cortical neuron model.\n"
                  "Low-threshold spiking.")
        elif event.key == pygame.K_6:
            change_model_type(5)
            print("Thalamo-cortical neuron model.\n")
        elif event.key == pygame.K_7:
            change_model_type(6)
            print("Resonator neuron model.\n")
if escape:
    break
if model_update_timer() < delta_t * 1000:
    continue
init_model_update_timer()
time_from_last_update += SCALE_T * delta_t
model_time = time_measure[-1] + SCALE_T * delta_t
x = x + RK4_step(x, SCALE_T * delta_t)
V = np.append(V, x[0])
W = np.append(W, x[1])
time_measure = np.append(time_measure, model_time)
if time_from_last_update - last_upd >= 1 / 60:
```

```
        sc.fill(WHITE)
        last_upd = time_from_last_update
        # nullclines (v, w)
        v_nullcl = []
        w_nullcl = []
        for v in np.linspace(MIN_X, MAX_X, 100):
            v_nullcl.extend([real_to_pygame([v, 0.04*v**2 + 5*v + 140 + I_app])])
            w_nullcl.extend([real_to_pygame([v, b[curr_type]*v])])
        pygame.draw.aalines(sc, (255, 0, 255), False, v_nullcl)
        pygame.draw.aalines(sc, (0, 255, 255), False, w_nullcl)
        # eq points
        eq_points = get_equilibrium_points()
        for eq_p in eq_points:
            f_v = np.poly1d([0.04, 5, (140 + I_app - eq_p[1])]).deriv()(eq_p[0])
            f_w = np.poly1d([-1, 0.04*(eq_p[0]**2) + 5*eq_p[0] + 140 + I_app]).deriv()(eq_p[1])
            g_v = np.poly1d([a[curr_type]*b[curr_type], -a[curr_type]*eq_p[1]]).deriv()(eq_p[0])
            g_w = np.poly1d([-a[curr_type], a[curr_type]*b[curr_type]*eq_p[0]]).deriv()(eq_p[1])
            eig_val, eig_vec = np.linalg.eig([[f_v, f_w], [g_v, g_w]])
            # print(eig_val)
            is_stable = np.real(eig_val[0]) < 0 and np.real(eig_val[1]) < 0
            pygame.draw.circle(sc, BLACK, real_to_pygame(eq_p), 6)
            if is_stable:
                pygame.draw.circle(sc, BLACK, real_to_pygame(eq_p), 5)
            else:
                pygame.draw.circle(sc, WHITE, real_to_pygame(eq_p), 5)
        # draw point
        point = real_to_pygame(x)
        pygame.draw.circle(sc, RED, point, 4)
        # trajectory
        VW_curve.pop(0)
        VW_curve.append([point[0], point[1]])
        # print(len(VW_curve))
        pygame.draw.aalines(sc, BLUE, False, VW_curve[-curve_dot_cnt:])
        # net
        draw_net()
        pygame.display.update()

max_time = time_measure[-1]
plt.plot(time_measure, V)
plt.grid(True)
plt.axis()
plt.xlabel('Time, ms')
```

```
plt.ylabel('Voltage, mV')
plt.show()
exit()
```

IZH_many.py (модель Ижикевича совокупности нейронов)

```
# -------------------------
dot_cnt = 2000
curr_type = 0

# Excitatory: [RS, IB, CH]
# Inhibitory: [FS, LTS]
# Thalamo- : [TC]
# Rezonator : [RZ]
a = [0.02, 0.02, 0.02, 0.1, 0.02, 0.02, 0.1]
b = [0.2, 0.2, 0.2, 0.2, 0.25, 0.25, 0.25]
c = [-65., -55., -50., -65., -65., -65., -65.]
d = [8., 4., 2., 2., 2., 0.05, 2.]

I_app = 0.0
v_0 = -70
w_0 = -14
v_thresh = 30.0
x = np.zeros((dot_cnt, 2))
rads = np.ones(dot_cnt)*3
curves_dot_cnt = 5
VW_curves = np.array([[real_to_pygame(xx) for i in range(curves_dot_cnt)] for xx in x])

def IZH(v, w):
    for i in range(dot_cnt):
        if v[i] >= v_thresh:
            x[i][0] = c[curr_type]
            x[i][1] = x[i][1] + d[curr_type]
            VW_curves[i] = [real_to_pygame(x[i]) for j in range(curves_dot_cnt)]
    return np.array([0.04 * v**2 + 5 * v + 140 - w + I_app, a[curr_type] * (b[curr_type] * v - w)])

def change_model_type(new_type):
    global curr_type
    curr_type = new_type

def get_equilibrium_points():
    poly = [0.04, (5-b[curr_type]), (140+I_app)]
    eq_points_v = np.roots(poly)
```

```
    res = []
    for eq_v in eq_points_v:
        if np.isreal(eq_v) and MAX_X >= eq_v >= MIN_X:
            eq_point = (eq_v, b[curr_type]*eq_v)
            res.extend([eq_point])
    return res

def respawn_dots(i, dots_gens):
    low_b = int(i/dots_gens * dot_cnt)
    up_b = int(min((i+1)/dots_gens * dot_cnt, dot_cnt))
    x[low_b:up_b, 0] = np.random.uniform(MIN_X, MAX_X, up_b-low_b)
    x[low_b:up_b, 1] = np.random.uniform(MIN_Y, MAX_Y, up_b-low_b)
    rads[low_b:up_b] = np.ones(up_b-low_b)*(i+1)*5/dots_gens + 1
    VW_curves[low_b:up_b] = np.array([[real_to_pygame(xx) for i in range(curves_dot_cnt)] for xx in x[low_b:up_b]])
# -------------------------

def RK4_step(y, dt):
    v = y[:, 0]
    w = y[:, 1]
    k1, q1 = IZH(v, w)
    [k2, q2] = IZH(v + 0.5 * k1 * dt, w + 0.5 * q1 * dt)
    [k3, q3] = IZH(v + 0.5 * k2 * dt, w + 0.5 * q2 * dt)
    [k4, q4] = IZH(v + k3 * dt, w + q3 * dt)
    return dt * np.array([k1 + 2 * k2 + 2 * k3 + k4, q1 + 2 * q2 + 2 * q3 + q4]).T

max_time = 40
delta_t = 0.001
time_measure = np.array([0])
last_upd = 0
time_from_last_update = 0
init_model_update_timer()
model_time = 0
dots_lifetime = 2000
dots_generations = 10
for i in range(dots_generations):
    respawn_dots(i, dots_generations)
last_respawn = [pygame.time.get_ticks() - dots_lifetime*i/dots_generations for i in range(dots_generations)]

while 1:
    for event in pygame.event.get():
```

```
if event.type == pygame.QUIT:
    exit()
elif event.type == pygame.KEYDOWN:
    if event.key == pygame.K_UP:
        print(f"I_app = {I_app}")
        I_app += 0.4
    elif event.key == pygame.K_DOWN:
        print(f"I_app = {I_app}")
        I_app -= 0.4
    elif event.key == pygame.K_SPACE:
        print("dV")
        x[:, 0] += 40
        event_keys = pygame.key.get_pressed()
        if not event_keys[pygame.K_LSHIFT]:
            x[:, 0] += 40
    elif event.key == pygame.K_ESCAPE:
        pygame.event.post(pygame.event.Event(pygame.QUIT))
    elif event.key == pygame.K_LEFTBRACKET:
        SCALE_T -= 0.5
        print(f"TIME SCALE = {SCALE_T}")
    elif event.key == pygame.K_RIGHTBRACKET:
        SCALE_T += 0.5
        print(f"TIME SCALE = {SCALE_T}")
    elif event.key == pygame.K_1:
        change_model_type(0)
        print("Excitatory cortical neuron model."
              "Regular spiking.")
    elif event.key == pygame.K_2:
        change_model_type(1)
        print("Excitatory cortical neuron model."
              "Instrinsically bursting.")
    elif event.key == pygame.K_3:
        change_model_type(2)
        print("Excitatory cortical neuron model."
              "Chattering.")
    elif event.key == pygame.K_4:
        change_model_type(3)
        print("Inhibitotary cortical neuron model."
              "Fast spiking.")
    elif event.key == pygame.K_5:
        change_model_type(4)
        print("Inhibitotary cortical neuron model."
```

```
                "Low-threshold spiking.")
            elif event.key == pygame.K_6:
                change_model_type(5)
                print("Thalamo-cortical neuron model.")
            elif event.key == pygame.K_7:
                change_model_type(6)
                print("Resonator neuron model.")
    if model_update_timer() < delta_t * 1000:
        continue
    init_model_update_timer()
    time_from_last_update += SCALE_T * delta_t
    model_time = time_measure[-1] + SCALE_T * delta_t
    x = x + RK4_step(x, SCALE_T * delta_t)
    time_measure = np.append(time_measure, model_time)
    for i in range(dots_generations):
        if pygame.time.get_ticks()-last_respawn[i] >= dots_lifetime:
            last_respawn[i] = pygame.time.get_ticks()
            respawn_dots(i, dots_generations)
    if time_from_last_update - last_upd >= 1 / 60:
        sc.fill(WHITE)
        last_upd = time_from_last_update
        # nullclines (v, w)
        v_nullcl = []
        w_nullcl = []
        for v in np.linspace(MIN_X, MAX_X, 100):
            v_nullcl.extend([real_to_pygame([v, 0.04*v**2 + 5*v + 140 + I_app])])
            w_nullcl.extend([real_to_pygame([v, b[curr_type]*v])])
        pygame.draw.aalines(sc, (255, 0, 255), False, v_nullcl)
        pygame.draw.aalines(sc, (0, 255, 255), False, w_nullcl)
        # eq points
        eq_points = get_equilibrium_points()
        for eq_p in eq_points:
            f_v = np.poly1d([0.04, 5, (140 + I_app - eq_p[1])]).deriv()(eq_p[0])
            f_w = np.poly1d([-1, 0.04*(eq_p[0]**2) + 5*eq_p[0] + 140 + I_app]).deriv()(eq_p[1])
            g_v = np.poly1d([a[curr_type]*b[curr_type], -a[curr_type]*eq_p[1]]).deriv()(eq_p[0])
            g_w = np.poly1d([-a[curr_type], a[curr_type]*b[curr_type]*eq_p[0]]).deriv()(eq_p[1])
            eig_val, eig_vec = np.linalg.eig([[f_v, f_w], [g_v, g_w]])
            # print(eig_val)
            is_stable = np.real(eig_val[0]) < 0 and np.real(eig_val[1]) < 0
            pygame.draw.circle(sc, BLACK, real_to_pygame(eq_p), 6)
            if is_stable:
                pygame.draw.circle(sc, BLACK, real_to_pygame(eq_p), 5)
```

```
            else:
                pygame.draw.circle(sc, WHITE, real_to_pygame(eq_p), 5)
        # draw point
        for i in range(dots_generations):
            low_b = int(i / dots_generations * dot_cnt)
            up_b = int(min((i + 1) / dots_generations * dot_cnt, dot_cnt))
            dot_rad = np.sin(np.pi * (pygame.time.get_ticks() - last_respawn[i]) / dots_lifetime) * rads[low_b]
            dot_col = min(
                max(255 - np.sin(np.pi * (pygame.time.get_ticks() - last_respawn[i]) / dots_lifetime) * 255, 0), 255)
            for j in range(low_b, up_b):
                point = real_to_pygame(x[j])
                pygame.draw.circle(sc, (dot_col, dot_col, dot_col), point, dot_rad)
        # trajectory
        for i in range(dot_cnt):
            VW_curves[i][:-1] = VW_curves[i][1:]
            VW_curves[i][-1] = real_to_pygame((x[i][0], x[i][1]))
            pygame.draw.aalines(sc, BLUE, False, VW_curves[i])
        # net
        draw_net()
        pygame.display.update()
```

ПЛАН-ГРАФИК

выполнения курсовой работы

обучающегося Султанова Р.Р.

Наименование этапа работ	Трудоемкость выполнения, час.	Процент к общей трудоемкости выполнения	Срок предъявления консультанту
Получение и согласование задания	0,3	0,8	6 неделя
Знакомство с литературой по теме курсовой работы	2,7	7,5	7 неделя
Теоретический анализ динамических моделей нейронов	3	8,3	8 неделя
Поиск программной среды и библиотек, подходящих для реализации систем	2	5,6	9 неделя
Программирование моделей одиночного нейрона	5	13,9	10 неделя
Программирование моделей совокупности нейронов	3	8,3	11 неделя
Отладка моделей	4	11,2	12 неделя
Программирование элементов отображения и управления модели	5	13,9	13 неделя
Отладка программы	5	13,9	14 неделя
Изучение моделей с помощью программы	3	8,3	15 неделя
Составление и оформление пояснительной записки и подготовка к защите	2,7	7,5	16 неделя
Защита	0,3	0,8	17 неделя
Итого	36	100	-