**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#__RefHeading___Toc1789_3851806917)

[1 Задача оптимального управления для сингулярных линейных систем 5](#__RefHeading___Toc1793_3851806917)

[1.1 Теория оптимального управления линейных систем 5](#__RefHeading___Toc8379_387698889)

[1.2 Условия корректности задачи LQR для сингулярных систем 8](#__RefHeading___Toc9758_3653515131)

[1.3 Выводы 9](#__RefHeading___Toc9760_3653515131)

[2 нейронные оду для решения уравнения Риккати 11](#__RefHeading___Toc1795_3851806917)

[2.1 Описание нейросетевой модели 11](#__RefHeading___Toc9633_646844236)

[2.2 Предыстория 12](#__RefHeading___Toc6644_387698889)

[2.3 Нейронные обыкновенные дифференциальные уравнения 14](#__RefHeading___Toc3806_1222046770)

[2.4 Методы численного интегрирования ОДУ 16](#__RefHeading___Toc1243_619138733)

[2.4.1 Метод Рунге-Кутты 16](#__RefHeading___Toc659_463851313)

[2.4.2 Метод Адамса-Башфорта 17](#__RefHeading___Toc663_463851313)

[2.4.3 Метод Фельберга 18](#__RefHeading___Toc665_463851313)

[2.4.4 Метод Ингленда 19](#__RefHeading___Toc1809_3851806917_%2525D)

[2.4.5 Метод Нюстрема 19](#__RefHeading___Toc1809_3851806917_%25251)

[2.4.6 Метод Милны 20](#__RefHeading___Toc1809_3851806917_%25252)

[2.4.7 Метод Хемминга 20](#__RefHeading___Toc1809_3851806917_%25253)

[2.5 Решение уравнения Риккати с помощью Neural ODE 20](#__RefHeading___Toc7649_1276381941)

[2.6 Выводы 21](#__RefHeading___Toc1805_3851806917)

[3 Разработка по 23](#__RefHeading___Toc1807_3851806917)

[3.1 Разработка архитектуры системы программного обеспечения 23](#__RefHeading___Toc1809_3851806917)

[3.2 Алгоритм численного решения ОДУ 24](#__RefHeading___Toc2944_2044766803)

[3.3 Разработка модулей приложения 26](#__RefHeading___Toc2946_2044766803)

[3.3.1 Модуль запуска (main.cpp) 26](#__RefHeading___Toc3157_3740196720)

[3.3.2 Конфигурационный модуль (Config) 26](#__RefHeading___Toc3159_3740196720)

[3.3.3 Абстрактный решатель (RiccatiSolver) 27](#__RefHeading___Toc3161_3740196720)

[3.3.4 Шаблонный класс (Solver<Method>) 27](#__RefHeading___Toc3163_3740196720)

[3.3.5 Фабрика решателей (fabric.hpp) 28](#__RefHeading___Toc3165_3740196720)

[3.3.6 Методы интегрирования (methods.hpp) 29](#__RefHeading___Toc3167_3740196720)

[3.3.7 Вспомогательные функции (utils.cpp/hpp) 30](#__RefHeading___Toc9881_3740196720)

[3.4 Диаграммы и блок-схемы 32](#__RefHeading___Toc1811_3851806917)

[3.5 Используемый инструментарий 35](#__RefHeading___Toc1813_3851806917)

[3.6 Тестирование 35](#__RefHeading___Toc1815_3851806917)

[3.6.1 Выбор стратегии тестирования 35](#__RefHeading___Toc2918_1901113919)

[3.6.2 Аппаратно-программная платформа для тестирования 36](#__RefHeading___Toc9635_646844236)

[3.6.3 Результаты тестирования 36](#__RefHeading___Toc2920_1901113919)

[3.7 Выводы 39](#__RefHeading___Toc1817_3851806917)

[4 Исследование работы методов 40](#__RefHeading___Toc1819_3851806917)

[4.1 Особенности методов и нюансы реализации 40](#__RefHeading___Toc3644_1669025185)

[4.2 Результаты экспериментов 42](#__RefHeading___Toc3646_1669025185)

[4.2.1 Матрицы 60х60 42](#__RefHeading___Toc6801_1669025185)

[4.2.2 Матрицы 100x100 45](#__RefHeading___Toc6803_1669025185)

[4.2.3 Матрицы 40x40 48](#__RefHeading___Toc6805_1669025185)

[4.2.4 Матрицы 10x10 49](#__RefHeading___Toc6807_1669025185)

[4.2.5 Матрицы 6х6 50](#__RefHeading___Toc6809_1669025185)

[4.2.6 Матрицы 4х4 51](#__RefHeading___Toc6811_1669025185)

[4.2.7 Матрицы 2х2 52](#__RefHeading___Toc6813_1669025185)

[4.3 Выводы 52](#__RefHeading___Toc669_463851313)

[ЗАКЛЮЧЕНИЕ 55](#__RefHeading___Toc1827_3851806917)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 57](#__RefHeading___Toc1829_3851806917)

[ПРИЛОЖЕНИЕ А код программы 58](#__RefHeading___Toc1831_3851806917)

# ВВЕДЕНИЕ

В условиях стремительного развития интеллектуальных технологий и усложнения технических объектов задача построения эффективных систем управления приобретает особую значимость. Классические методы оптимального управления, такие как линейно-квадратичный регулятор (LQR), широко применяются благодаря своей математической строгости и эффективности в линейных системах.

Однако на практике часто возникают ситуации, когда система обладает сингулярной структурой, а её параметры могут быть частично неизвестны, изменяться во времени или зависеть от внешних условий. Подобные особенности характерны для многих прикладных областей, таких как робототехника, оптимальное управление, экономика, крупномасштабные взаимосвязанные системы, электроэнергетика, биомедицинские системы и т.д. В таких условиях использование строгих аналитических методов затруднено или невозможно, что делает необходимым применение гибких, обучаемых моделей, способных адаптироваться к изменяющейся динамике и неполноте данных.

Одним из перспективных направлений является использование методов машинного обучения, в частности нейронных дифференциальных уравнений (Neural ODE), предложенных в 2018 году Чэньом и соавторами[]. Этот подход представляет собой новую архитектуру нейросетей, в которой непрерывная динамика системы моделируется с помощью параметризованного векторного поля и интегрируется численно. Neural ODE объединяет идеи глубокого обучения и классических методов решения обыкновенных дифференциальных уравнений, обеспечивая непрерывное во времени представление модели, возможность обратного дифференцирования и естественную интеграцию в контур управления. Благодаря этим свойствам он становится мощным инструментом для анализа и моделирования сложных динамических систем на основе наблюдаемых данных.

В рамках данного исследования рассматривается система управления с линейной динамикой, описываемая уравнением с сингулярной матрицей :

Основное внимание уделяется численным методам решения обыкновенных дифференциальных уравнений и их возможному использованию совместно с Neural ODE. Выбор численного интегратора напрямую влияет на устойчивость, точность и вычислительную эффективность обучения моделей на базе Neural ODE. Особенно это критично в задачах с сингулярной структурой, где малейшая ошибка аппроксимации может привести к некорректному управлению или срыву устойчивости.

Цель работы заключается в разработке и сравнении различных численных методов интегрирования при решении матричного уравнения Риккати, которое лежит в основе построения оптимального регулятора.

Новизна данного подхода состоит в применении Neural ODE к задаче управления системой с сингулярной динамикой и в комплексном анализе влияния различных численных интеграторов на скорость и качество аппроксимации решения уравнения Риккати. Практическая значимость заключается в создании универсального инструмента, способного использоваться в условиях нечеткой информации о таких объектах исследования, как автономные роботы, адаптивные экономические модели и интеллектуальные технологии управления.

# Задача оптимального управления для сингулярных линейных систем

## Теория оптимального управления линейных систем

Линейный квадратичный регулятор (LQR) — это оптимальная стратегия управления, используемая в теории управления и инженерии. Она направлена ​​на оптимальную работу динамической системы путем минимизации функции стоимости, которая обычно представляет собой компромисс между усилием управления и ошибкой состояния. LQR особенно эффективен для линейных систем, где динамика системы может быть описана линейными дифференциальными уравнениями.

Сингулярные системы представляют собой обобщение обычных линейных динамических систем. Их особенностью является наличие сингулярной матрицы в уравнении состояния:

|  | (1) |
| --- | --- |

где - вектор состояния системы;

- вектор управляющих воздействий;

- матрица системы (описывает внутреннюю динамику);

- матрица управления (описывает, как вход влияет на состояние);

- сингулярная матрица, то есть .

Сделано допущение []: решение существует и единственно, если для некоторого — это гарантирует что любой входной вектор синтезирует одну и только одну траекторию состояния

Такая модель является приближением многих физических систем при малых отклонениях от точки равновесия. Она может описывать механические, электрические, термодинамические или другие типы объектов.

Основная задача системы управления — обеспечить поведение объекта в соответствии с заданной целью. В рамках линейно-квадратичного подхода эта цель формулируется как минимизация некоторого интегрального функционала качества. Иными словами, мы стремимся найти такое управляющее воздействие , которое минимизирует заданную стоимость функционирования системы, при этом обеспечивая устойчивость и достижение цели управления.

Для оценки качества управления вводится следующий функционал []:

|  | (2) |
| --- | --- |

где - симметричная, положительно полуопределенная матрица, определяющая цену отклонения состояний;

- положительно определенная матрица, характеризующая стоимость управляющих усилий;

- симметричная положительно полуопределенная матрица

- фиксированные конечный момент времени;

- произволен, что позволяет оптимизировать траекторию на всем интервале;

Чем больше элементы матрицы , тем сильнее регулятор стремится минимизировать отклонение состояния системы от заданного значения, поскольку увеличение весовых коэффициентов усиливает "штраф" за ошибку в соответствующей переменной состояния.

Например, если в системе управления положением и скоростью увеличить элемент ​, отвечающий за позицию, регулятор будет жестче подавлять отклонения по координате, даже если это потребует более резких управляющих воздействий.

В то же время, чем больше элементы матрицы , тем менее агрессивным будет управление, поскольку регулятор начнет экономить управляющие сигналы, избегая больших значений управляющего воздействия. Это означает, что при высоких значениях система будет реагировать медленнее, но управление станет более плавным, что может быть критически важно для энергоэффективности и долговечности исполнительных устройств.

Для получения условий оптимальности используется гамильтонов формализм. Вводится функция Гамильтона []:

|  | (3) |
| --- | --- |

где - вектор сопряженных переменных (тензор Лагранжа);

- строгое решение задачи оптимизации, полученное на основе функционала

Необходимые условия оптимальности:

|  | (4) |
| --- | --- |
|  | (5) |
|  | (6) |

Предположим, что сопряженная переменная связана с состоянием через матрицу :

|  | (7) |
| --- | --- |

Тогда подставляя (4) в (6), после математических преобразований можно получить матричное уравнение Риккати, адаптированное под сингулярную структуру системы:

|  | (8) |
| --- | --- |

После получения решения обобщенного матричного уравнения Риккати (4) оптимальное воздействие в задаче LQR для системы с сингулярной матрицей формируется в виде обратной связи по состоянию:

|  | (9) |
| --- | --- |

где матрица усиления определяется выражением:

|  | (10) |
| --- | --- |

Поскольку матрица однозначно определяется через решение уравнения (4), а также известны параметры системы , то можно построить замкнутую систему. На каждом шаге времени текущий вектор состояния будет считываться с датчиков, после чего он умножается на матрицу , которая находится согласно (4) и (9). Результатом будет управляющее воздействие которое подается на исполнительные механизмы системы. Такая реализация обеспечит устойчивость системы при соблюдении условий корректности задачи LQR.

## Условия корректности задачи LQR для сингулярных систем

Матрица должна быть симметричной и положительно полуопределенной. Это обязательные свойства, вытекающие из условий оптимальности и уравнения Риккати. Они необходимы:

* чтобы функционал (2) действительно был минимизирован;
* чтобы обратная связь стабилизировала систему;
* чтобы управление не обращалось в бесконечность.

Если , то . В случае сингулярной системы имеет структуру:

|  | (11) |
| --- | --- |

Тогда в может быть «свободный» блок (например ), не влияющий на управление, например

На основе леммы 2 из [] следует условие существования и единственности решения уравнения Риккати. Если система не имеет бесконечных динамических мод, и матрица имеет блочную структуру:

|  | (12) |
| --- | --- |

то обобщенное уравнение Риккати имеет решение.

А количество динамических мод определяет матрица , имеющая вид:

|  | (13) |
| --- | --- |

Если невырождена, то система имеет только конечные моды, а если вырождена, то появляются бесконечных мод, где . Отсутствие бесконечных мод гарантирует, что система управляема стандартными методами, такими как LQR.

## Выводы

Одним из главных достоинств линейно-квадратичного регулятора является возможность поддерживать устойчивость системы, одновременно оптимизируя её производительность. Подбор матриц и оказывает влияние не только на управляющие воздействия, но и на устойчивость замкнутого контура. Грамотная настройка этих матриц позволяет добиться требуемых динамических характеристик, включая время переходного процесса, величину перерегулирования и статическую ошибку.

Ключевым этапом при построении LQR-регулятора является решение матричного уравнения Риккати. Найденное решение позволяет вычислить оптимальную матрицу усиления обратной связи , с помощью которой формируется управляющее воздействие по закону . Таким образом, само уравнение Риккати служит основой для синтеза оптимального управления, минимизирующего заданный функционал качества.

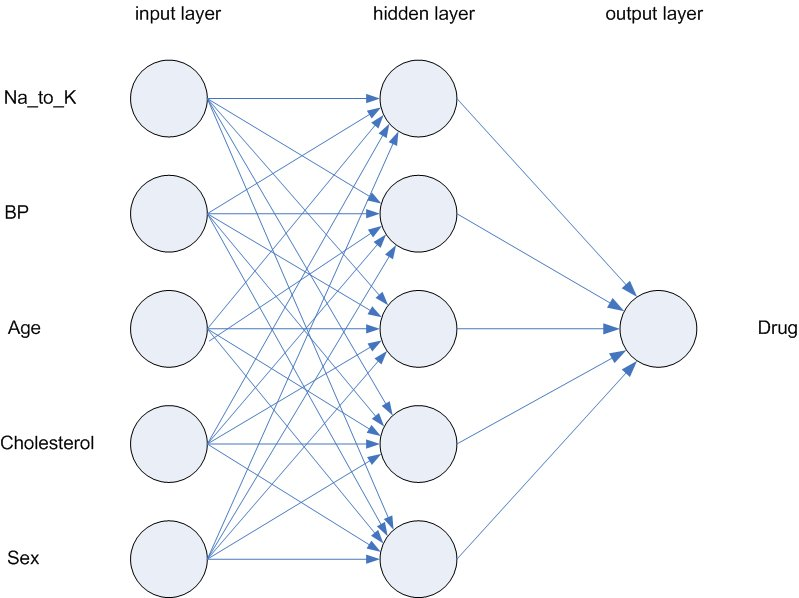
Управление на основе линейной обратной связи обладает высокой вычислительной эффективностью, так как на каждом шаге требуется лишь операция умножения текущего вектора состояния на матрицу усиления. Такая структура регулятора минимизирует нагрузку на вычислительные ресурсы и хорошо масштабируется для задач, работающих в режиме реального времени.

LQR находит широкое применение в различных сферах, таких как авиакосмическая отрасль, робототехника и автомобильная промышленность. В аэрокосмической сфере его используют в системах управления полётом для обеспечения стабильности и эффективности при разных режимах работы. В робототехнике с его помощью решают задачи управления движением и точного слежения за траекториями, что позволяет роботам и манипуляторам выполнять задачи с высокой точностью.

# нейронные оду для решения уравнения Риккати

## Описание нейросетевой модели

Нейронные сети представляют собой упрощенные модели работы нервной системы живых организмов. Базовые блоки называются нейронами и обычно сгруппированы в слои, как показано на рисунке.

Рисунок 1. Структура нейронной сети

Нейронная сеть использует упрощенную модель обработки информации человеческим мозгом. Нейросети работают, обсчитывая большое количество связанных между собой обрабатываемых элементов, которые представляют абстрактную версию нейронов.

Обрабатывающие блоки (нейроны) сгруппированы в слои. В типичной нейросети есть три части: нейроны входного слоя представляют входные поля, есть один или несколько скрытых слоев и есть выходной слой, содержащий один или несколько нейронов, представляющих поля назначения. Каждому соединению между нейронами назначается та или иная сила воздействия, или вес. Входные данные поступают в первый слой, далее значения распространяются по слоям от каждого нейрона данного слоя в каждый нейрон следующего слоя. Окончательный результат снимается с выходного слоя.

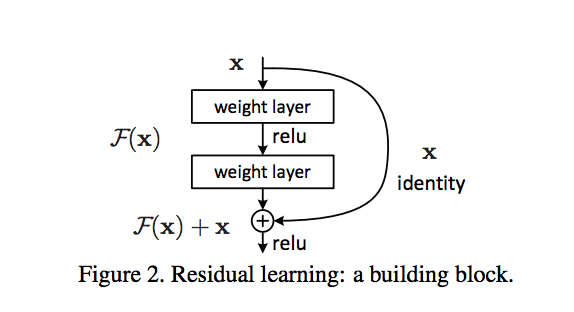
Нейронная сеть обучается путем просмотра записей; для каждой записи нейронная сеть генерирует предсказание и, если предсказание неверно, вносит поправки в веса. Процесс повторяется большое число раз, и точность предсказаний постепенно повышается, пока не срабатывает один из критериев остановки.

## Предыстория

В настоящее время нейронные сети применяются для решения широкого круга задач в области машинного обучения. С момента, когда глубокие нейронные сети продемонстрировали высокую эффективность в решении сложных задач, их архитектуры начали активно развиваться: увеличивалась глубина моделей и число параметров с целью повышения точности и выразительности.

Однако в скором времени выяснилось, что увеличение глубины нейросетей не всегда приводит к улучшению результатов: слишком глубокие модели перестают обучаться. Основной причиной этого явления являются затухающие градиенты — в процессе обратного распространения ошибки градиенты многократно умножаются, и их значения могут стремиться к нулю. Это приводит к тому, что веса нижних слоёв почти не обновляются, и обучение всей сети останавливается.

Решением данной проблемы стала архитектура ResNet. В ней предложили инновационный подход — использование остаточных связей (skip connections), которые передают вход предыдущего слоя напрямую на выход последующего, складывая его с преобразованным сигналом. Это позволяет градиентам в процессе обратного распространения ошибки (backpropagation) распространяться не только через нелинейные преобразования слоёв, но и по "короткому пути" — через саму остаточную связь. Таким образом, каждый слой в ResNet получает доступ как к преобразованному градиенту (через свёртки и активации), так и к неизменённому градиенту, что предотвращает его затухание даже в очень глубоких сетях.

Рисунок 2. Блок со skip-connection

В результате ResNet смог успешно обучать сети глубиной более 100 слоёв, устанавливая новые рекорды в задачах классификации изображений и других областях глубокого обучения.

Рекуррентная запись ResNet может быть описана следующим образом:

|  | (14) |
| --- | --- |

где - данные с предыдущего слоя, - слой нейросети, - текущие данные.

Она очень напоминает метода Эйлера — численный метод решения обыкновенных дифференциальных уравнений с заданными начальными условиями. Пусть задано некоторое дифференциальное уравнение с начальными условиями:

|  | (15) |
| --- | --- |

Нужно найти решение этого уравнения в конечный момент времени , то есть .

Чтобы найти решение методом Эйлера, нужно разбить отрезок времени на равные промежутки. Обозначим - шаг разбиения временного отрезка. Тогда - называются узлами. Очевидно, что чем меньше будет шаг разбиения, тем более точным окажется решение.

Решение ОДУ методом Эйлера находится по рекуррентной формуле

|  | (16) |
| --- | --- |

и она с точностью до повторяет формулу ResNet.

## Нейронные обыкновенные дифференциальные уравнения

Заметив такое сходство, авторы статьи [] предложили представить нейронную сеть как обыкновенное дифференциальное уравнение. Можно сказать, что если у вас есть очень много рекуррентных слоев с малым разбиением по времени , то по сути вы решаете систему ОДУ.

Neural ODE заменяет дискретную последовательность слоев нейросети на непрерывную динамику состояния, описываемую дифференциальным уравнением следующего вида:

|  | (17) |
| --- | --- |

где - скрытое состояние модели в момент времени , - нейронная сеть, - обучаемые параметры нейронной сети, - начальные условия (входные данные нейронной сети)

В отличие от традиционных нейросетей, в которых выход последнего слоя зависит от фиксированной глубины, в Neural ODE выход определяется как решение ОДУ на отрезке времени , а глубина сети становится переменной величиной, зависящей от выбранного численного метода и заданной точности.

Решение уравнения можно представить в интегральной форме:

|  | (18) |
| --- | --- |

На практике это решение приближается с использованием одного из численных методов интегрирования: Эйлера, Рунге-Кутты, Адамса и других.

Таким образом, модель определяет скрытое состояние как непрерывную функцию от времени, а сама нейросеть становится эквивалентом системы с континуальным количеством слоёв, все из которых используют одни и те же параметры .

Обучение моделей Neural ODE, как и в традиционных нейронных сетях, осуществляется с помощью метода обратного распространения ошибки (backpropagation). Однако из-за специфики архитектуры — непрерывного представления скрытого состояния через решение ОДУ — стандартный алгоритм обучения требует адаптации. Прямое сохранение всех промежуточных состояний на каждом временном шаге, необходимых для расчёта градиентов, может оказаться слишком ресурсоёмким: число шагов заранее неизвестно и может быть большим.

Для решения этой проблемы используется метод сопряжённых переменных (adjoint method). Вместо хранения всех промежуточных состояний, модель решает другую систему дифференциальных уравнений — так называемое adjoint-уравнение — в обратном направлении по времени. Это позволяет восстанавливать необходимые градиенты без сохранения траектории состояния, что значительно снижает потребление памяти. Кроме того, этот подход обеспечивает точное вычисление производных при соблюдении заданной точности численного интегрирования.

В процессе обучения параметры функции динамики обновляются при помощи стандартных оптимизационных алгоритмов (например, Adam или SGD), основываясь на градиентах, полученных через решение adjoint-системы. При этом качество обучения и устойчивость модели напрямую зависят от выбранного численного метода интегрирования: он должен быть не только точным, но и достаточно стабильным для вычисления производных. Поэтому выбор интегратора становится критически важным аспектом построения и обучения Neural ODE — и именно этот вопрос рассматривается в данной работе как основной предмет исследования.

## Методы численного интегрирования ОДУ

Заметим, что для численного решения задачи Коши совершенно необязательно использовать метод Эйлера. Хотя он является интуитивно понятным и простым в реализации, его точность оставляет желать лучшего: это метод первого порядка, и ошибка в нём линейно зависит от шага интегрирования. В реальных вычислениях, особенно при моделировании сложных динамик, таких как в Neural ODE, этого часто оказывается недостаточно. Существуют более точные и устойчивые методы численного интегрирования с фиксированным шагом, позволяющие существенно уменьшить глобальную погрешность при том же числе итераций. В следующих разделах будут рассмотрены наиболее распространённые из таких методов.

### Метод Рунге-Кутты

Формула для вычисления методом Рунге-Кутты четвертого порядка точности имеет следующий вид:

|  | (19) |
| --- | --- |

где — угловые коэффициенты касательных к графику решения в различных точках, вычисляемые по формулам

|  | (20) |
| --- | --- |

Метод Рунге-Кутты, как и методы Эйлера, является одношаговым, так как значение вычисляется на основе текущего значения .

Формула для метода Рунге-Кутты третьего порядка точности следующая:

|  | (21) |
| --- | --- |

где коэффициенты определяются согласно (22)

|  | (22) |
| --- | --- |

### Метод Адамса-Башфорта

В многошаговом методе Адамса-Башфорта третьего порядка точности для нахождения точки используются три предыдущие точки:

|  | (23) |
| --- | --- |

Для начала расчетов требуются четыре «разгонные» точки , которые можно получить любым из предложенных методов.

В многошаговом методе Адамса-Башфорта четвертого порядка точности для нахождения точки используются четыре предыдущие точки:

|  | (24) |
| --- | --- |

Для начала расчетов требуются четыре «разгонные» точки .

В многошаговом методе Адамса-Башфорта пятого порядка точности для нахождения точки используются пять предыдущих точек:

|  | (25) |
| --- | --- |

Для начала расчетов требуются пять «разгонных» точек .

Методы Адамса-Башфорта не позволяет изменять шаг в процессе расчетов. В отличие от метода Рунге-Кутты четвертого порядка в этих методах требуется вычислять только одно новое значение правой части системы вместо четырех. Высокая точность методов достигается при этом за счет учета информации о предыдущих точках. Напротив, в методе Рунге-Кутты, как и в других одношаговых методах, недостающую информацию о поведении правых частей системы получают в результате вычислений в специальным образом выбранных дополнительных точках.

### Метод Фельберга

В методе Фельберга пятого порядка точности для расчета точки используется формула:

|  | (26) |
| --- | --- |

где

|  | (27) |
| --- | --- |

В методе Фельберга четвертого порядка точности для расчета точки используется формула:

|  | (28) |
| --- | --- |

где коэффициенты определяются согласно (27)

### Метод Ингленда

В методе Ингленда пятого порядка точности для расчета точки используется формула:

|  | (29) |
| --- | --- |

где

|  | (30) |
| --- | --- |

В методе Ингленда четвертого порядка точности для расчета точки используется формула:

|  | (31) |
| --- | --- |

где коэффициенты определяются согласно (30)

### Метод Нюстрема

В многошаговых методах Нюстрема второго, третьего и четвертого порядка точности для нахождения точки используются две, три и четыре предыдущие точки соответственно:

|  | | (32) |
| --- | --- | --- |
|  | (33) | |
|  | (34) | |

Для начала расчетов по формулам (32) - (34) требуются две, три и четыре «разгонные» точки соответственно.

### Метод Милны

Многошаговый метод Милны четвертого порядка точности может быть реализован следующим способом:

|  | (35) |
| --- | --- |

Для начала расчетов по формуле (35) требуется четыре «разгонные» точки , которые могут быть найдены любым из предыдущих методов.

В методе Милны шестого порядка точности для расчета точки используется шесть предыдущих точек:

|  | (36) |
| --- | --- |

### Метод Хемминга

Многошаговый метод Хемминга четвертого порядка точности может быть реализован следующим способом, в котором для нахождения точки используются четыре предыдущие точки:

|  | (37) |
| --- | --- |

## Решение уравнения Риккати с помощью Neural ODE

Как мы выяснили, уравнение Риккати играет ключевую роль в теории оптимального управления в задаче построения линейно-квадратичного регулятора. Оно имеет вид

|  | (38) |
| --- | --- |

где — искомая симметричная матрица, описывающая функцию стоимости, а - заданные матрицы, характеризующие поведение системы, управления и штрафов. Решение этого уравнения позволяет получить матрицу усиления , с помощью которой формируется оптимальное управляющее воздействие .

Уравнение Риккати представляет собой систему матричных ОДУ, и его решение сводится к численному интегрированию во времени от начального условия

В контексте Neural ODE данное уравнение можно интерпретировать как модель, в которой функция динамики уже известна, то есть не обучается, а задана аналитически. Это позволяет использовать всю инфраструктуру Neural ODE (ODE solver, численные интеграторы, итерационная логика) для вычисления траектории . Такой подход удобен, потому что мы хотим сравнить различные численные методы, применяемые в рамках Neural ODE, на конкретной и строго определённой системе.

В данной работе рассматривается вариант, в котором выбирается равной единице. Это упрощает выражение и устраняет влияние масштабирующих коэффициентов на поведение системы, позволяя сосредоточиться исключительно на сравнении точности и стабильности численного решения. По сути, модель с уравнением Риккати в данном случае используется как тестовая задача, позволяющая объективно сравнивать поведение интеграторов, применяемых в Neural ODE, в условиях известной динамики и строгой постановки.

## Выводы

В данной главе была рассмотрена архитектура Neural ODE — нейросетевого подхода, в котором эволюция скрытого состояния задаётся в виде решения обыкновенного дифференциального уравнения. Такая модель представляет собой естественное продолжение глубоких нейросетей, заменяя дискретную последовательность слоёв на непрерывную динамику. В этом контексте временной интервал можно интерпретировать как непрерывную глубину нейросети, где каждый шаг интегрирования соответствует одному виртуальному слою.

Особое внимание было уделено численным методам интегрирования, поскольку в Neural ODE они играют ключевую роль: именно с их помощью вычисляется траектория состояния, а значит — и выход модели. Каждый численный метод, будь то Рунге-Кутта, Адамса, Милна или Хемминга, потенциально может быть использован внутри Neural ODE, если он обеспечивает необходимую точность и устойчивость. Правильный выбор интегратора позволяет балансировать между скоростью вычислений и качеством предсказания, что особенно важно для задач, чувствительных к вычислительным ресурсам.

В качестве примера конкретной задачи была рассмотрена система, описываемая уравнением Риккати, которое возникает при решении задачи линейно-квадратичного регулятора. Поскольку в этой задаче правая часть уравнения заранее известна и не требует обучения, её удобно использовать для сравнения численных методов в изолированной, контролируемой среде. Такое решение позволяет объективно оценить, насколько эффективен тот или иной метод, прежде чем применять его в обучаемых моделях Neural ODE.

Таким образом, Neural ODE — это принципиально новый подход, где нейросети и дифференциальные уравнения объединяются в единую математическую модель. В рамках данной работы акцент сделан на выборе и сравнении интеграторов, чтобы впоследствии использовать наиболее подходящие из них в практических задачах, включая оптимальное управление.

# Разработка по

## Разработка архитектуры системы программного обеспечения

Архитектура разработанной программы по сути является реализацией Neural ODE, в котором состояние системы обновляется во времени на основе численного решения ОДУ. В классическом варианте Neural ODE правая часть задаётся нейросетью и оптимизируется по данным, тогда как в данной реализации динамика заранее известна: используются фиксированные матрицы , что эквивалентно ситуации, когда сеть уже "обучена", а параметры модели известны. Таким образом, задача сводится не к обучению, а к точному и быстрому численному интегрированию системы с целью сравнения различных методов по эффективности.

Программа предоставляет набор численных методов (Рунге-Кутта, Адамс, Милна и др.), каждый из которых реализован как структура с функцией step() и подключается через шаблонный решатель Solver<Method>, основанный на абстрактном классе RiccatiSolver. Этот базовый класс задаёт универсальный интерфейс для решения уравнения и реализует общую итерационную логику, включая проверку сходимости, контроль ошибок и обработку начальных условий. Каждый конкретный метод переопределяет только шаг интегрирования через функцию update\_step(), что обеспечивает модульность и минимизирует дублирование кода.

Приложение для ОС Windows/Linux реализовано на языке C++ в виде набора взаимосвязанных модулей, каждый из которых отвечает за определённый этап вычислений: ввод данных, запуск метода, визуализация, сохранение результатов. Вся конфигурация управления передаётся через аргументы командной строки, а фабричная функция create\_solver() связывает пользовательский ввод с конкретной реализацией численного метода. Благодаря использованию шаблонов и абстракций новые численные методы могут быть добавлены без изменения основной логики, что делает архитектуру легко масштабируемой и пригодной для дальнейшего расширения — в том числе с целью интеграции в обучаемую Neural ODE-модель.

## Алгоритм численного решения ОДУ

Решение матричного уравнения Риккати реализовано в методе solve() абстрактного класса RiccatiSolver, от которого наследуется шаблонный класс Solver<Method>. Алгоритм построен по общему принципу численного интегрирования: состояние системы обновляется во времени шаг за шагом, пока не будет достигнута заданная точность или не исчерпано максимальное число итераций. Основные этапы алгоритма следующие:

1. Инициализация

Исходная матрица заполняется нулями, устанавливаются параметры моделирования: шаг , диапазон времени , допустимая погрешность и число потоков. Устанавливается текущее значение времени .

1. Подготовка начальных точек

Для многошаговых методов (таких, как Адамса, Милны и др.) с помощью метода acceleration\_points() вычисляется требуемое число предыдущих точек с использованием универсального метода Рунге-Кутты 4-го порядка. Эти значения сохраняются в структуру std::deque<Eigen::MatrixXd> и используются при первых итерациях.

1. Основной итерационный цикл

Алгоритм запускает цикл, в котором на каждой итерации:

* вызывается метод update\_step(P\_n, h, history) — реализация конкретного численного метода, определённая в Solver<Method>, которая возвращает новое значение матрицы
* рассчитывается ошибка между текущей и предыдущей матрицей, обычно в виде нормы .
* сохраняется значение ошибки при активированном флаге   
  --draw;
* при включённом пошаговом режиме (--manual) выполнение может быть приостановлено на каждом шаге;
* вызывается check\_nan() для проверки корректности матрицы
* значение времени и счётчик итераций увеличиваются.

1. Завершение

Цикл завершается при достижении необходимой точности или превышении максимального количества итераций. В завершении вызывается verify\_solution(P), которая подставляет полученное решение в уравнение Риккати и сохраняет невязку в файл для анализа.

1. Формирование результата

Метод solve() возвращает структуру Result, содержащую:

* финальную матрицу
* количество шагов,
* последнюю ошибку,
* при необходимости массив ошибок по шагам для построения графика.

Благодаря использованию шаблонов структура Solver<Method> остаётся неизменной при замене численного метода — меняется только поведение update\_step(). Это позволяет проводить сравнение методов на одной и той же задаче.

## Разработка модулей приложения

Программа разделена на несколько ключевых модулей приведенных ниже, обеспечивающих гибкость и масштабируемость.

### Модуль запуска (main.cpp)

Отвечает за

* инициализацию конфигурации из командной строки с помощью parse\_cli*;*
* загрузку входных матриц c помощью функции read\_matrix\_from\_file.Программа ожидает, что файлы с матрицами (например E.dat) находятся в папке data относительно запускаемого файла;
* выбор численного метода через create\_solver,
* запуск алгоритма решения;
* отображение хода интегрирования и сохранение результатов;

### Конфигурационный модуль (Config)

Структура Config содержит параметры моделирования:

* шаг интегрирования: double h*,* по умолчанию 0.01
* начальное и конечное время: double t0 и double t\_end*,* по умолчанию от 0 до 10
* метод интегрирования: String method*,* например «nystrom4»
* требуемая точность решения: double target\_error*,* по умолчанию 0.001
* максимальное количество итераций: int max\_steps*,* по умолчанию 200
* количество потоков: int threads*,* по умолчанию 1
* флаг отрисовки графиков: bool draw*,* по умолчанию false
* флаг пошаговой отладки: bool manual*,* по умолачанию false

Эта структура передается во все ключевые функции и обеспечивает централизованное управление параметрами.

### Абстрактный решатель (RiccatiSolver)

Класс RiccatiSolver реализует основу для всех конкретных методов численного интегрирования уравнения Риккати. Это абстрактный базовый класс, от которого наследуются шаблонные специализированные решатели Solver<Method>, реализующие конкретную стратегию вычислений. Он содержит:

* riccati\_equation(P) Матричное уравнение Риккати: return E\*P\*A + A^T \*P\*E + Q - E\*B\*R\* B^T \*P\*E
* solve(Config cfg) Основной метод решения. Реализует итерационный цикл по времени с вызовом update\_step().
* verify\_solution(P) Подставляет найденное решение в исходное уравнение и сохраняет остаток в файл. Это позволяет проверить, насколько точно выполняется уравнение при полученной матрице. Если значения матрицы далеки от нуля, это значит что матрица найдена не правильно.
* acceleration\_points() Для многошаговых методов (например, Адамса, Милны) требуются несколько разгонных точек. Эти точки вычисляются с помощью метода Рунге-Кутты 4-го порядка. Хранятся в виде двусторонней очереди.
* update\_step() Чисто виртуальная функция. Конкретный численный метод должен реализовать эту функцию — она выполняет один шаг интегрирования.

### Шаблонный класс (Solver<Method>)

Шаблонный класс, который наследуется от абстрактного базового класса RiccatiSolver. Он предназначен для конкретной реализации численного метода решения уравнения Риккати, передаваемого через параметр шаблона Method (например, Inglend4, Adams5). Этот класс реализует вирутальный метод update\_step()*.* Внутри update\_step()вызывается статический метод step()*.* Каждый метод решения ОДУ реализуется как структура с одним методом step()*.*

Пример:

|  |
| --- |
| struct RungeKutta4 {  static Eigen::MatrixXd step( RiccatiSolver& solver,  const Eigen::MatrixXd& P,   double h,  const std::deque<Eigen::MatrixXd>& ) {  Eigen::MatrixXd k1, k2, k3, k4;  k1.noalias() = solver.riccati\_equation(P);  k2.noalias() = solver.riccati\_equation(P + (0.5 \* h) \* k1);  k3.noalias() = solver.riccati\_equation(P + (0.5 \* h) \* k2);  k4.noalias() = solver.riccati\_equation(P + h \* k3);  return P + (h / 6.0) \* (k1 + 2 \* k2 + 2 \* k3 + k4);  }  }; |

Чтобы добавить новый метод (например, Рунге-Кутты 6 порядка), нужно просто создать структуру в methods.hppи добавить условие в фабрику. Тем самым больше ничего править в коде не требуется.

### Фабрика решателей (fabric.hpp)

Файл fabric.hppсодержит реализацию фабричного метода, задача которого на основе имени выбранного численного метода создать соответствующий объект решателя. Фабрика create\_solver()является связующим элементом между пользовательским вводом и архитектурой решателя, она создаёт объект Solver<Method> в зависимости от строки, переданной в аргументах (runge4, adams5 и др.).

Фабрика это порождающий шаблон проектирования, определяющий общий интерфейс создания объектов в родительском классе и позволяющий изменять создаваемые объекты в дочерних классах.

Шаблон позволяет классу делегировать создание объектов подклассам. Используется, когда:

* Классу заранее неизвестно, объекты каких подклассов ему нужно создать.
* Обязанности делегируются подклассу, а знания о том, какой подкласс принимает эти обязанности, локализованы.
* Создаваемые объекты родительского класса специализируются подклассами.

Пример :

|  |
| --- |
| std::unique\_ptr<RiccatiSolver> create\_solver(...) {  if (name == "runge4")  return std::make\_unique<Solver<RungeKutta4>>(...);  else if (name == "adams4")  return std::make\_unique<Solver<Adams4>>(...);  // ...  } |

Вне зависимости от метода, возвращается std::unique\_ptr<RiccatiSolver>, что позволяет использовать один и тот же код вызова. Она позволяет добавлять новые методы, просто расширяя список в fabric.hpp добавлением строки вида

|  |
| --- |
| else if (name == "milna6")  return std::make\_unique<Solver<Milna6>>(...); |

### Методы интегрирования (methods.hpp)

Файл methods.hpp содержит реализации конкретных численных методов, используемых для пошагового решения матричного уравнения Риккати. Каждый метод оформлен в виде отдельной структуры с единственным статическим методом step, который выполняет один шаг интегрирования по времени. Такой подход делает их легко заменяемыми в шаблонном классеSolver<Method>.

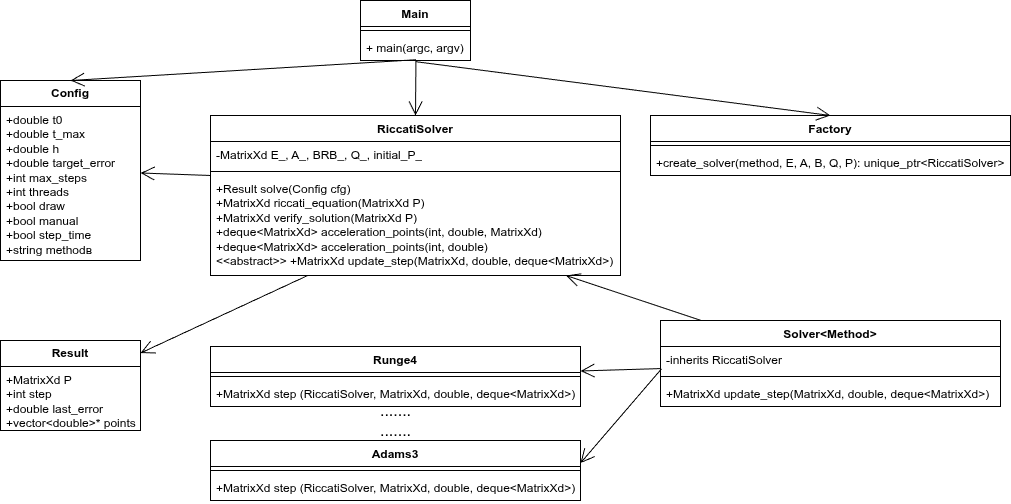
|  |
| --- |
| struct Adams3 {  static Eigen::MatrixXd step( RiccatiSolver& solver,  const Eigen::MatrixXd& P,  double h,  const std::deque<Eigen::MatrixXd>& prev)  {  return P + (h / 12.0) \*  ((23 \* solver.riccati\_equation(P)) -  (16 \* solver.riccati\_equation(prev[1])) +  (5 \* solver.riccati\_equation(prev[2])));  }  }; |

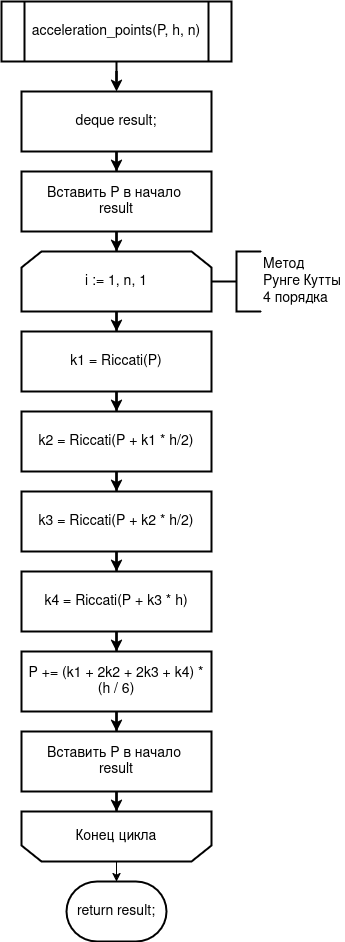
### Вспомогательные функции (utils.cpp/hpp)

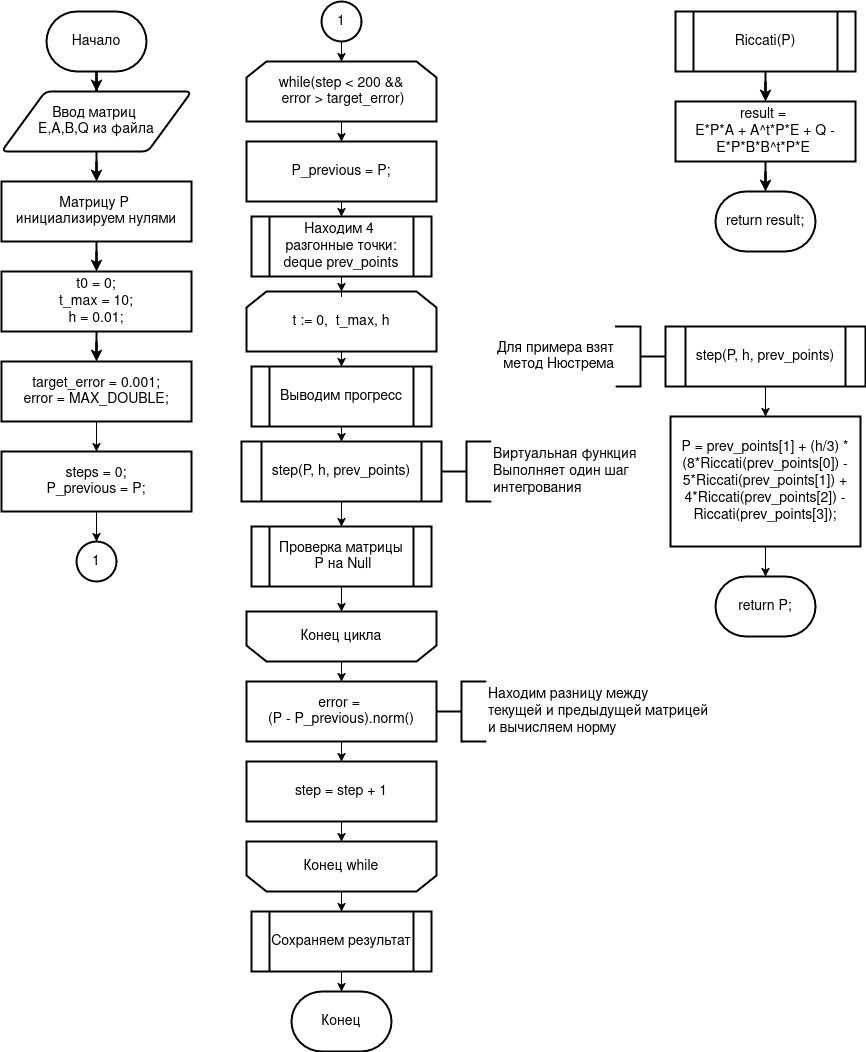
* Функция read\_matrix\_from\_file() загружает матрицу из текстового файла, построчно считывая числовые значения и проверяя корректность формата каждого элемента. При успешной обработке она преобразует данные в объект Eigen::MatrixXd, обеспечивая строгую валидацию структуры и значений входного файла.
* Функция draw\_graph() предназначена для визуализации изменения ошибки численного решения на каждом шаге интегрирования. При наличии флага --draw она передаёт данные об ошибке во внешний процесс gnuplot через стандартный ввод, формируя график и автоматически сохраняет его в PNG-файл по пути results/png/<имя\_метода>.png.
* Функция progress() отображает в консоли текущий шаг интегрирования, значение ошибки, время и фрагмент матрицы решения, обновляя вывод каждые две секунды или при пошаговом режиме. Это позволяет пользователю отслеживать ход вычислений в реальном времени и при необходимости выполнять контроль на каждом этапе.
* Функция show\_results() сохраняет параметры моделирования, итоговую ошибку, количество шагов и финальную матрицу в файл results/output.txt. Кроме того, она выводит краткую сводку результатов в консоль, обеспечивая как сохранение, так и визуальное представление вычислений.
* Функция check\_nan() выполняет проверку матрицы на наличие некорректных значений, таких как NaN (Not a Number), которые могут возникнуть в результате численной нестабильности или деления на ноль. При обнаружении недопустимого значения программа немедленно прерывает выполнение и выбрасывает исключение с указанием текущего шага и времени.

## Диаграммы и блок-схемы

Рисунок 3. UML диаграмма классов



Рисунок 4. Блок-схема функции, находящей разгонные точки для многошаговых методов

Рисунок 5. Блок-схема решателя

## Используемый инструментарий

В качестве языка программирования для реализации данного дипломного проекта был выбран С++, поскольку он обеспечивает высокую производительность за счёт компилируемости и оптимизации на уровне машинного кода, поддерживает объектно-ориентированное и обобщённое программирование для создания гибкой модульной архитектуры, а также предоставляет богатую стандартную библиотеку и инструменты для работы с математическими вычислениями, что критически важно для эффективного решения матричных уравнений, таких как уравнение Риккати.

Программа использует внешние библиотеки:

* Eigen — высокопроизводительная C++ библиотека для линейной алгебры, поддерживающая операции с матрицами, векторами, численными решателями и алгоритмами линейной алгебры. Поддерживает арифметику матриц и векторов.
* CLI11 — библиотека для обработки аргументов командной строки в C++. Она предоставляет простой и интуитивно понятный интерфейс для парсинга командной строки, генерации справки (help) и валидации входных данных.
* gnuplot — утилита для построения графиков.

## Тестирование

### Выбор стратегии тестирования

Для выбора оптимальной стратегии тестирования учитывались особенности разработанного программного продукта. Было принято решение использовать функциональное тестирование по методологии "чёрного ящика"

Функциональное тестирование направлено на проверку соответствия программного обеспечения заявленным требованиям, то есть его способности выполнять необходимые пользователям задачи в заданных условиях. Оно фокусируется на том, что делает система, а не на том, как она это реализует.

Метод "чёрного ящика" предполагает тестирование без анализа внутренней структуры кода. Тестировщик проверяет только входные и выходные данные, оценивая корректность работы системы исключительно по её внешнему поведению.

### Аппаратно-программная платформа для тестирования

Программное обеспечение разрабатывалось и тестировалось на стенде со следующей конфигурацией:

* Операционная система: Arch Linux (ядро версии 6.14.7)
* Процессор: Intel Core i7-10700K
* Оперативная память: 16 ГБ
* Компилятор: g++ (GNU C++ Compiler, версия 13.2.1)
* Эмулятор терминала: Kitty
* Периферийные устройства:

Клавиатура

### Результаты тестирования

Запуск программы без аргументов:

|  |
| --- |
| ❯ ./solver  Riccati Equation Solver  Usage: [OPTIONS]  OPTIONS:  -h, --help Print this help and exit  --method TEXT REQUIRED  Метод решения (runge3,runge4,  adams3, adams4, adams5,  milna4, milna6,  nystrom2, nystrom3, nystrom4,  felberg4, felberg5,  inglend4, inglend5, hemming4)  --h FLOAT Шаг интегрирования >= 0  --t0 FLOAT Начальное время  (по умолчанию 0)  --t\_end FLOAT Конечное время  (по умолчанию 10)  --error FLOAT Требуемая погрешность  (по умолчанию 0.001)  --max\_steps INT Максимум шагов  (по умолчанию 200)  --threads INT Количество потоков  (по умолчанию 1)  Для небольших матриц большое  кол-во потоков негативно  повлияют на скорость.  --draw Рисовать график ошибки  (использует gnuplot)  --manual Пошаговое выполнение  вычислений  --step\_time Отображать время выполнения  одного шага |

Попытка запустить программу без входных матриц:

|  |
| --- |
| ❯ ./solver --method nystrom4 --h 0.1 --draw  Не удалось открыть файл data/E.dat |

Вывод прогресса программы во время вычислений:

|  |
| --- |
| Метод: nystrom4 | Требуемая ошибка: 0.001 | Ошибка: 0.04395  h: 0.1 | Итерация : 4 | t : 9.2 из 10  P(первые 8x8 элементов):  0.4244 0.159  0.159 0 |

Тестирование на корректность входных данных. Пример для  
матрицы :

|  |
| --- |
| ❮ ./solver --method nystrom4 --h 0.1  Обнаружен нечисловой токен: a-1 |

Результат работы программы:

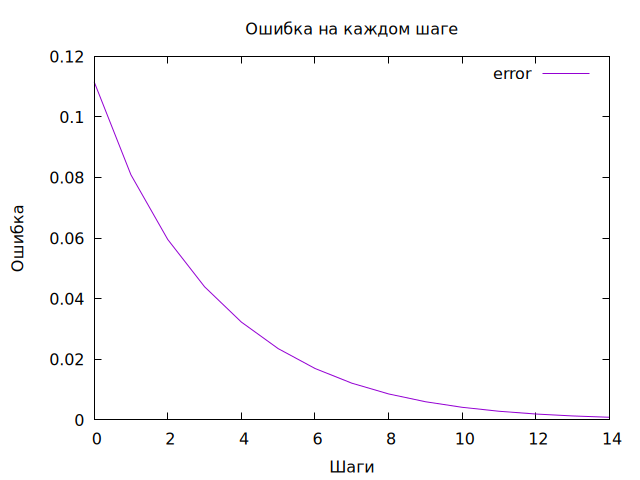
|  |
| --- |
| Исходные данные: nystrom4; t = 0 ; t\_end = 10 ; h = 0.1 ; error\_value = 0.001  Последняя ошибка = 0.000871733 ; Итерация = 15  Mатрица P записана в output.txt  Время работы программы: 0 сек. 6 мс. |

Полученные матрицы:

|  |  |
| --- | --- |
| output.txt | verified.txt |
| 0.4725506 0.2343252  0.2343252 0.0000000 | -0.0000095 0.0039001  0.0039001 0.0000000 |

Поскольку матрица из файла verified.txt оказалась близка к нулю, матрица из файла output.txt найдена верно.

Отрисованный график при переданном параметре --draw

Рисунок 6. Пример графика динамики ошибки

## Выводы

В данной главе была подробно рассмотрена архитектура и реализация программного обеспечения для численного решения матричного уравнения Риккати. Программа построена по модульному принципу, она обеспечивает возможность расширения, повторного использования компонентов и простоту интеграции новых численных методов. Использование шаблонных классов и абстрактных интерфейсов позволило реализовать универсальный механизм интегрирования, не зависящий от конкретного метода.

Особое внимание было уделено реализации набора численных методов (Runge-Kutta, Adams, Milne и др.), подключаемых через фабричный механизм. Это сделало возможным автоматизированное сравнение методов по точности, скорости и устойчивости. Таким образом, разработанное программное обеспечение представляет собой удобную платформу для оценки эффективности численных интеграторов, что особенно важно при выборе метода интегрирования для использования в Neural ODE. Поскольку в нашей реализации правая часть уравнения фиксирована, программа имитирует поведение Neural ODE на этапе после обучения, когда параметры уже заданы, а задача сводится к точному решению уравнения. Это делает программу полезной как инструмент для подбора и тестирования численных методов, которые в будущем могут использоваться в нейросетевых моделях с непрерывной динамикой, таких как Neural ODE.

# Исследование работы методов

Целью данного эксперимента является сравнение различных численных методов интегрирования, которые могут быть использованы в архитектуре Neural ODE, на примере решения уравнения Риккати, возникающего в задаче линейно-квадратичного управления (LQR). В отличие от стандартного применения Neural ODE, где функция динамики аппроксимируется обучаемой нейросетью, в данной работе рассматривается случай, когда правая часть системы известна заранее. Это позволяет исключить влияние неопределённостей, связанных с обучением, и сфокусироваться исключительно на точности, устойчивости и производительности численных методов.

В качестве тестовой задачи выбрано уравнение Риккати, поскольку оно хорошо изучено, имеет чёткую математическую постановку и представляет собой реальный пример из теории управления. Такое уравнение задаёт динамику симметричной матрицы , от которой напрямую зависит управляющее воздействие в системе. Решение этого уравнения требует численного интегрирования матричной системы, что делает его удобной основой для сравнения интеграторов.

Эксперименты проводятся на наборах матриц различных размеров — от маленьких (2×2) до больших (100×100) — с использованием фиксированного шага интегрирования. Оцениваются следующие параметры: точность (на основе нормы разности между шагами), число итераций, время выполнения и устойчивость метода (наличие/отсутствие сбоев, ошибок или расходимости).

## Особенности методов и нюансы реализации

Для сравнения были выбраны как одношаговые, так и многошаговые численные методы интегрирования. В частности, рассматривались методы Рунге-Кутты 3-го и 4-го порядка, методы Адамса-Башфорта 4-го и 5-го порядка, а также схемы Нюстрема, Милна, Фельберга, Ингленда и Хемминга. Каждый из этих методов обладает своими особенностями, влияющими на точность, устойчивость и скорость вычислений при использовании в задачах, подобных Neural ODE.

Одношаговые методы, такие как Рунге-Кутты, вычисляют новое состояние только на основе текущего значения и производной, что делает их простыми в реализации. Многошаговые методы, наоборот, используют информацию из нескольких предыдущих шагов, что может дать прирост точности, но требует дополнительных условий для инициализации. В частности, для запуска метода Адамса-Башфорта 5-го порядка необходимо иметь пять предшествующих точек. Поэтому такие методы требуют так называемых «разгонных шагов», которые обычно вычисляются другим, стабильным одношаговым методом (например, Рунге-Куттой).

В процессе экспериментов было замечено, что поведение методов Адамса существенно зависит от точности и свежести предыдущих точек. Если в буфере находятся значения вычисленные в начале итерации, метод Адамса приводит к расходимости на обычных шагах (0.1 — 0.001). Метод становится стабильным только при микроскопическом шаге , но тогда время возрастает на порядки и становится практически неприемлемым. Решить эту проблему можно сохраняя предыдущие точки в каждый момент времени, а не один раз в начале итерации. Однако такой подход такой прием плохо согласуется с «слой за слоем» логикой Neural ODE: буфер превращается в скрытое внутреннее состояние, не имеющее естественной интерпретации в виде дополнительного слоя сети и усложняющее схему обратного прохода. Тем не менее, чтобы оценить потенциал многошаговых схем, в работе приведены результаты и для версий Адамса с динамически обновляемыми точками; при анализе нужно учитывать, что их производительность достигается ценой отхода от канонической структуры Neural ODE.

В реализации сравниваемых методов шаг интегрирования оставался фиксированным, без использования адаптивных стратегий. Это было сделано для того, чтобы обеспечить честное сравнение условий и исключить влияние внутренних механизмов регулировки точности. Все методы запускались с одинаковыми параметрами: фиксированным шагом, одинаковым временным интервалом и ограничением на максимальное число итераций.

## Результаты экспериментов

Входные данные для всех замеров:

* Требуемая ошибка 0.001 (норма разницы между матрицами и  
   предыдущей)
* Начальное время = 0
* Конечное время = 10
* Ограничение по итерациям max\_steps = 200

В ходе экспериментов было замечено, что поведение методов для матриц меньших размеров (2×2, 4×4, 6×6, 10×10 и 40×40) демонстрирует аналогичные закономерности сходимости, в дальнейшем анализе мы сосредоточимся на результатах для матрицы 60×60 как наиболее показательном примере, не приводя избыточных графиков для вышеперечисленных размерностей.

### Матрицы 60х60

Все одношаговые методы и методы Адамса нашли матрицу за 47 итераций при = 0.02, график ошибки у них идентичный. Результаты занесены в таблицу 1. В таблице минус указывает на неустойчивость метода при заданном шаге: происходит расходимость, и значения матрицы превышают допустимый диапазон типа double.

Таблица 1. э

|  |  |  |  |
| --- | --- | --- | --- |
| Метод \ Шаг | 0.1 | 0.02 | 0.01 |
| Рунге Кутта 3 порядка | 8 сек. 081 мс. | 37 сек. 293 мс. |  |
| Рунге Кутта 4 порядка | 10 сек 058 мс. | 52 сек. 613 мс. |  |
| Адамс-Башфорт 3 порядка | - | 33 сек. 474 мс. |  |
| Адамс-Башфорт 4 порядка | - | 39 сек. 992 мс. |  |
| Адамс-Башфорт 5 порядка | - | - | 101 сек. 72 мс. |
| Фельберг 4 порядка | - | 66 сек. 538 мс. |  |
| Фельберг 5 порядка | 15 сек. 238 мс. | 79 сек. 580 мс. |  |
| Ингленд 4 порядка | 10 сек. 045 мс. | 49 сек. 519 мс. |  |
| Ингленд 5 порядка | - | 77 сек. 655 мс. |  |

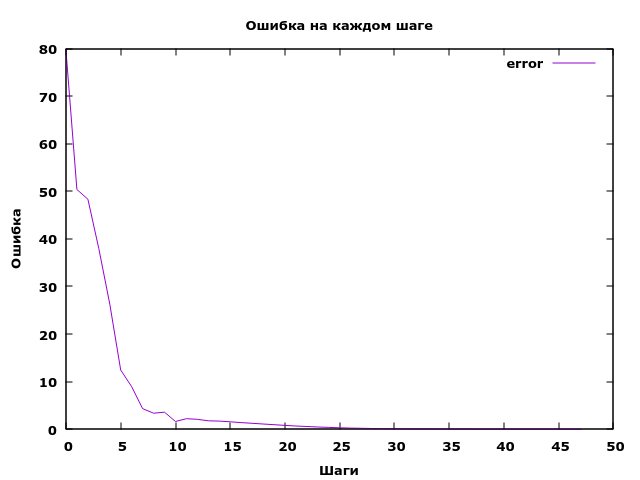


Рисунок 7. График ошибки для методов закончивших за 47 итераций

Многошаговые методы при лимите в 200 итераций не успевают найти адекватное решение, значение ошибки не достигает заданного. Поэтому ограничение по итерациям можно убирать при использовании методов.

Также обнаружилась важная особенность многошаговых методов: в течение одной итерации метод может найти решение матрицы уже при   
 = 1. Однако вычисления продолжаются до = 10, хотя матрица уже не меняется. Происходят бесполезные матричные перемножения, которые лишь тратят время.

Таблица 2. Многошаговые методы с шагом = 0.02

|  |  |  |
| --- | --- | --- |
| Методы \ Ограничения | 200 итераций, = 10 | Без лимита по итерациям,   = 0.5 |
| Нюстрем 2 порядка | 52 сек. | 108 сек. (5982 итераций) |
| Нюстрем 3 порядка | 123 сек. | 221 сек. (5554 итераций) |
| Нюстрем 4 порядка | 184 сек. | 267 сек. (5183 итераций) |
| Милна 4 порядка | 134 сек. | 141 сек. (3429 итераций) |
| Милна 6 порядка | 232 сек. | 162 сек. (2440 итераций) |
| Хемминг 4 порядка | 184 сек. | 237 сек. (3958 итераций) |

Для эффективной работы многошаговых методов можно подбирать оптимальное экспериментально:

Опытным путем выяснил, что минимальное значение может меняться в зависимости от метода. Например, у Нюстрема 2 порядка, Милны 4 порядка, Хемминга 4 порядка = 0.03, а у Милны 6 порядка = 0.05. (все это при шаге = 0.02)

Также выяснилось, что у некоторых методов при уменьшении после некоторого значения начинает увеличиватся количество шагов. Например, у метода Нюстрема 2 порядка число шагов при уменьшении никак не изменилось, а у Хемминга число шагов возросло с 3958 при = 0.5 до 4757 при = 0.03.

Поскольку поведение многошаговых методов во многом схожее, я не буду приводить графики для каждого из них. Вместо этого покажу типичный пример на методе Нюстрема второго порядка — этого достаточно, чтобы проиллюстрировать общую картину (рисунки )

|  |  |
| --- | --- |
| Рисунок 8. График ошибки для метода Нюстрема 2 порядка с лимитом в 200 итераций | Рисунок 9. График ошибки для метода Нюстрема 2 порядка без лимита на итерации |

### Матрицы 100x100

Шаг для всех методов = 0.1

У одношаговых методов ошибка колебалась в районе ~ 0.5 – 0.9, так и не достигнув требуемой ошибки в 0.001, поэтому они исчерпали 200 итераций. При этом все методы, кроме Адамса-Башфорта 5 порядка, смогли найти матрицу.

Многошаговые методы получили требуемую ошибку не достигнув лимита по итерациям. Это количество записано в таблицу рядом с временем выполнения. Если итерации не указаны, это значит что метод достиг лимита итераций.

Таблица 3. какоето название

|  |  |
| --- | --- |
| Метод | Время |
| Рунге Кутта 3 порядка | 142 сек. |
| Рунге Кутта 4 порядка | 193 сек. |
| Адамс-Башфорт 3 порядка | 130 сек. |
| Адамс-Башфорт 4 порядка | 169 сек. |
| Адамс-Башфорт 5 порядка | 2305 сек. (Матрицу не нашел) |
| Фельберг 4 порядка | 240 сек. |
| Фельберг 5 порядка | 293 сек. |
| Ингленд 4 порядка | 192 сек. |
| Ингленд 5 порядка | 290 сек. |
| Нюстрем 2 порядка | 36 сек. (146 итераций) |
| Нюстрем 3 порядка | 68 сек. (111 итераций) |
| Нюстрем 4 порядка | 78 сек. (91 итерация) |
| Милна 4 порядка | 80 сек. (127 итераций) |
| Милна 6 порядка | 210 сек. (186 итераций) |
| Хемминг 4 порядка | 32 сек. (39 итераций) |

|  |  |
| --- | --- |
| Рисунок 10. График для одношаговых методов на примере Рунге Кутты 4 порядка | Рисунок 11. График ошибки для метода Адамса-Башфорта 5 порядка |
| Рисунок 12. График ошибки для многошаговых методов на примере Нюстрема 2 порядка | |

### Матрицы 40x40

Все одношаговые методы и методы Адамса нашли матрицу за 47 итераций при = 0.01, график ошибки у них идентичный.

Таблица 4. э

|  |  |  |
| --- | --- | --- |
| Метод \ Шаг | 0.1 | 0.01 |
| Рунге Кутта 3 порядка | 2 сек. 414 мс. | 24 сек. |
| Рунге Кутта 4 порядка | 3 сек. 088 мс. | 31 сек. |
| Адамс-Башфорт 3 порядка | - | 22 сек. |
| Адамс-Башфорт 4 порядка | - | 28 сек. |
| Адамс-Башфорт 5 порядка | - | 35 сек. |
| Фельберг 4 порядка | 4 сек. 107 мс. | 46 сек. |
| Фельберг 5 порядка | 4 сек. 943 мс. | 50 сек. |
| Ингленд 4 порядка | 3 сек. 255 мс. | 33 сек. |
| Ингленд 5 порядка | 4 сек. 861 мс. | 51 сек. |

Таблица 5. Многошаговые методы с шагом = 0.01

|  |  |  |
| --- | --- | --- |
| Методы \ Ограничения | 200 итераций,  = 10 | Без лимита по итерациям,   = 0.5 |
| Нюстрем 2 порядка | 42 сек. | 97 сек. (8077 итераций) |
| Нюстрем 3 порядка | 107 сек. | 217 сек. (7498 итераций) |
| Нюстрем 4 порядка | 156 сек. | 279 сек. (6997 итераций) |
| Милна 4 порядка | 116 сек. | 155 сек. (4832 итераций) |
| Милна 6 порядка | 205 сек. | 193 сек. (3483 итераций) |
| Хемминг 4 порядка | 161 сек. | 241 сек. (5517 итераций) |

### Матрицы 10x10

Все одношаговые методы и методы Адамса нашли матрицу за 9 итераций при = 0.01, график ошибки у них идентичный.

Таблица 6. э

|  |  |  |
| --- | --- | --- |
| Метод \ Шаг | 0.1 | 0.01 |
| Рунге Кутта 3 порядка | - | 273 мс. |
| Рунге Кутта 4 порядка | 33 мс. (8 итераций) | 365 мс. |
| Адамс-Башфорт 3 порядка | - | 274 мс. |
| Адамс-Башфорт 4 порядка | - | 363 мс. |
| Адамс-Башфорт 5 порядка | - | 454 мс. |
| Фельберг 4 порядка | 38 мс. (200 итераций) | 464 мс. |
| Фельберг 5 порядка | 49 мс. (8 итераций) | 551 мс. |
| Ингленд 4 порядка | 32 мс. (8 итераций) | 367 мс. |
| Ингленд 5 порядка | 55 мс. (9 итераций) | 558 мс. |

На матрице 10×10 наблюдалось исключение: метод Фельберга 4-го порядка с шагом = 0.1 начал колебаться и не смог корректно завершить интегрирование. Видимо, при определенных сочетаниях параметров системы и шага возникает такая неустойчивость. Если поставить = 0.01, то метод начинает вести себя аналогично другим.

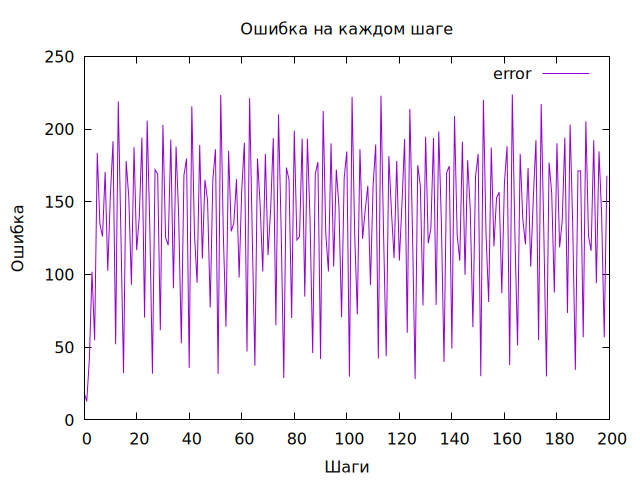
Рисунок 13. График ошибки Фельберга 4-го порядка (10×10, = 0.1)

Таблица 7. Многошаговые методы с шагом = 0.01

|  |  |  |
| --- | --- | --- |
| Методы \ Ограничения | 200 итераций,  = 10 | Без лимита по итерациям,   = 0.5 |
| Нюстрем 2 порядка | 2 сек. 024 мс. | 0 сек. 709 мс. (1195 итераций) |
| Нюстрем 3 порядка | 6 сек. 122 мс. | 1 сек. 675 мс. (1011 итераций) |
| Нюстрем 4 порядка | 8 сек. 128 мс. | 1 сек. 951 мс. (897 итераций) |
| Милна 4 порядка | 6 сек. 084 мс. | 1 сек. 360 мс. (789 итераций) |
| Милна 6 порядка | 10 сек. 161 мс. | 1 сек. 780 мс. (646 итераций) |
| Хемминг 4 порядка | 8 сек. 097 мс. | 1 сек. 042 мс. (479 итераций) |

### Матрицы 6х6

Все одношаговые методы и методы Адамса нашли матрицу за 8 итераций при = 0.01, график ошибки у них идентичный.

Таблица 8. э

|  |  |  |
| --- | --- | --- |
| Метод \ Шаг | 0.1 | 0.01 |
| Рунге Кутта 3 порядка | 7 мс. | 78 мс. |
| Рунге Кутта 4 порядка | 10 мс. | 106 мс. |
| Адамс-Башфорт 3 порядка | - | 79 мс. |
| Адамс-Башфорт 4 порядка | - | 106 мс. |
| Адамс-Башфорт 5 порядка | - | 133 мс. |
| Фельберг 4 порядка | 13 мс. | 134 мс. |
| Фельберг 5 порядка | 16 мс. | 160 мс. |
| Ингленд 4 порядка | 10 мс. | 106 мс. |
| Ингленд 5 порядка | 16 мс. | 160 мс. |

Таблица 9. Многошаговые методы с шагом = 0.01

|  |  |  |
| --- | --- | --- |
| Методы \ Ограничения | 200 итераций,  = 10 | Без лимита по итерациям,   = 0.5 |
| Нюстрем 2 порядка | 0 сек. 659 мс. | 276 мс. (1216 итераций) |
| Нюстрем 3 порядка | 1 сек. 983 мс. | 554 мс. (1033 итераций) |
| Нюстрем 4 порядка | 2 сек. 632 мс. | 693 мс. (917 итераций) |
| Милна 4 порядка | 1 сек. 987 мс. | 438 мс. (798 итераций) |
| Милна 6 порядка | 3 сек. 298 мс. | 589 мс. (652 итераций) |
| Хемминг 4 порядка | 2 сек. 640 мс. | 352 мс. (494 итераций) |

### Матрицы 4х4

У одношаговых методов и методов Адамса график ошибки очень похожий.

Таблица 10. Одношаговые методы

|  |  |  |
| --- | --- | --- |
| Метод \ Шаг | 0.1 | 0.01 |
| Рунге Кутта 3 порядка | 3 мс. (6 итераций) | 37 мс. (8 итераций) |
| Рунге Кутта 4 порядка | 4 мс. (7 итераций) | 51 мс. (8 итераций) |
| Адамс-Башфорт 3 порядка | - | 32 мс. (7 итераций) |
| Адамс-Башфорт 4 порядка | - | 43 мс. (7 итераций) |
| Адамс-Башфорт 5 порядка | - | 54 мс. (7 итераций) |
| Фельберг 4 порядка | 5 мс. (7 итераций) | 64 мс. (8 итераций) |
| Фельберг 5 порядка | 6 мс. (7 итераций) | 77 мс. (8 итераций) |
| Ингленд 4 порядка | 4 мс. (7 итераций) | 50 мс. (8 итераций) |
| Ингленд 5 порядка | 6 мс. (7 итераций) | 77 мс. (8 итераций) |

Таблица 11. Многошаговые методы с шагом = 0.01

|  |  |  |
| --- | --- | --- |
| Методы \ Ограничения | 200 итераций, = 10 | Без лимита по итерациям,   = 0.5 |
| Нюстрем 2 порядка | 0 сек. 314 мс. | 60мс. (663 итераций) |
| Нюстрем 3 порядка | 0 сек. 923 мс. | 128 мс. (519 итераций) |
| Нюстрем 4 порядка | 1 сек. 246 мс. | 152 мс. (460 итераций) |
| Милна 4 порядка | 0 сек. 935 мс. | 123 мс. (484 итераций) |
| Милна 6 порядка | 1 сек. 544 мс. | 197 мс. (471 итераций) |
| Хемминг 4 порядка | 1 сек. 229 мс. | 88 мс. (266 итераций) |

### Матрицы 2х2

Для матриц размером 2×2 все численные методы завершали работу менее чем за 1 мс, поэтому основное внимание было уделено числу итераций. Все методы запускались с фиксированным шагом  = 0.1. Следует отметить, что метод Адамса 5-го порядка при таком шаге не смог стартовать из-за численной неустойчивости.

Остальные методы успешно справились с задачей: все одношаговые методы, а также методы Адамса 3-го и 4-го порядков завершили интегрирование всего за 2 итерации.

Многошаговые схемы (такие как Милна, Хемминга, Нюстрема) потребовали больше шагов — в среднем около 13, с разбросом примерно ±5 итераций.

## Выводы

В рамках проведённых вычислительных экспериментов был выполнен сравнительный анализ одношаговых и многошаговых численных методов интегрирования, применяемых к задаче решения уравнения Риккати. Основная цель состояла в оценке точности, устойчивости и скорости этих методов на системах различной размерности, с фиксированным шагом интегрирования. Такое сравнение важно для дальнейшего применения выбранных схем в архитектурах Neural ODE, особенно в контексте задач управления, где динамика системы строго задана.

Методы Рунге-Кутты продемонстрировали стабильную и надёжную работу на всём диапазоне размеров матриц. Они оказались устойчивыми даже при сравнительно большом шаге (например,  = 0.1), что позволяет значительно сократить количество итераций и общее время интегрирования без ущерба для точности. Это делает их универсальным выбором как для малых, так и для крупных задач.

Одношаговые методы в целом показали предсказуемое поведение и устойчивую сходимость. Однако на матрицах большого размера (например, 100×100) была замечена интересная особенность: при достижении лимита по числу итераций решение уже находилось, но погрешность (норма разности между текущим и предыдущим шагом) продолжала колебаться, не достигая строго заданного порога. Это связано с тем, что при работе с большими матрицами даже небольшие локальные колебания в отдельных элементах приводят к ощутимой норме по всей матрице. В таких случаях, возможно, потребуется перенастройка критерия остановки, например, с использованием относительной ошибки или порогов, зависящих от размерности системы.

Многошаговые методы (в частности, Милна, Нюстрема, Хемминга) продемонстрировали потенциал для более быстрого достижения точности, особенно на крупных матрицах. Благодаря накоплению информации с предыдущих шагов они сглаживают локальные колебания и быстрее «успокаиваются». Также было замечено, что при стандартном значении ​ некоторые методы выполняются заметно дольше, однако при сокращении временного интервала они выполняются быстрее, сохраняя корректность работы. Это открывает возможность гибкой настройки времени интегрирования в зависимости от требований к скорости и точности решения.

В целом, исследования, описанные в данной главе показали, что:

* Рунге-Кутта остаётся универсальным и надёжным методом, особенно при ограничениях по вычислительным ресурсам
* Для задач с большими матрицами важно не только выбирать метод, но и адаптировать критерии оценки ошибки под масштаб системы
* Для многошаговых методов требуется подбирать оптимальное , чтобы ускорить время вычисления

Полученные результаты могут быть использованы для дальнейшего проектирования Neural ODE-моделей с заданной правой частью, где точность и устойчивость численного решения напрямую влияют на поведение всей системы.

# ЗАКЛЮЧЕНИЕ

В данной работе была рассмотрена архитектура Neural ODE (нейросетевые обыкновенные дифференциальные уравнения) как современный подход к моделированию динамических систем. В отличие от классических нейросетей с фиксированным числом слоёв, технология Neural ODE позволяет задавать эволюцию скрытого состояния в виде непрерывной функции времени, решая систему дифференциальных уравнений с помощью численного интегрирования. Это открывает возможности для более гибкого и интерпретируемого моделирования, особенно в задачах управления, где структура динамики известна заранее.

В качестве основной тестовой задачи было выбрано матричное уравнение Риккати, возникающее при решении задачи линейно-квадратичного регулятора (LQR). Данное уравнение обладает чётко определённой аналитической структурой, но требует численного решения, что делает его удобным объектом для анализа поведения различных интеграторов. Поскольку правая часть системы в этой задаче фиксирована, основное внимание было сосредоточено не на обучении модели, а на сравнении точности, устойчивости и скорости различных численных методов при решении ОДУ.

В рамках реализации была разработана модульная архитектура на языке C++ с поддержкой шаблонных решателей, фабричного метода подключения новых численных схем и универсального интерфейса запуска. Это позволило провести масштабный эксперимент с использованием одношаговых и многошаговых методов интегрирования, включая схемы Рунге-Кутты, Адамса, Милна, Нюстрема, Хемминга и других.

Результаты экспериментов показали, что методы Рунге-Кутты, особенно 4-го порядка, обеспечивают устойчивую работу на всём диапазоне размерностей и позволяют использовать крупные шаги, снижая общее время вычислений.

Таким образом, в ходе работы была проведена комплексная оценка численных методов интегрирования для задач, лежащих на стыке численного анализа и нейросетевых моделей. Несмотря на то, что Neural ODE изначально возникли как архитектурный приём в глубоком обучении, по своей сути они возвращают внимание к фундаментальной задаче — точному и устойчивому решению обыкновенных дифференциальных уравнений. В данной работе показано, что именно выбор численного метода, его параметров и понимание особенностей поведения системы оказывают ключевое влияние на итоговый результат.

Значимость численного интегрирования особенно проявляется в задачах управления, где даже малые погрешности в расчёте могут привести к нестабильности всей системы. Уравнение Риккати, выбранное в качестве тестовой задачи, широко применяется в технических и экономических приложениях, а потому является отличной площадкой для тестирования численных схем. Полученные результаты и наблюдения могут быть полезны не только в контексте Neural ODE, но и при построении классических регуляторов, симуляторов и математических моделей динамики.

Таким образом, работа в равной степени затрагивает как современные направления в машинном обучении, так и опирается на проверенные временем методы численного анализа. Это делает её потенциально полезной как для практикующих инженеров, так и для исследователей, разрабатывающих новые гибридные подходы на пересечении математики, информатики и прикладной науки.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Квакернаак, Х., Сиван, Р.. Линейные оптимальные системы управления. // Х. Квакернаак — Москва: Изд-во Мир, 1977. — 656 c.
2. Пантелеев А.В., Якимова А.С., Босов А.В. Обыкновенные дифференциальные уравнения в примерах и задачах: Учебное пособие // Пантелеев А.В. — Москва: Изд-во МАИ, 2000. — 380с.
3. Chen, R.T.Q., Rubanova, Y., Bettencourt, J., Duvenaud, D. Neural Ordinary Differential Equations // Advances in Neural Information Processing Systems (NeurIPS). – 2018. – Vol. 31. – P. 6571–6583.
4. CLI11 парсер аргументов командной строки для C++ // [Электронный ресурс] : github.com. – URL: https://cliutils.github.io/CLI11/
5. Eigen линейная алгебра на C++ // [Электронный ресурс]: eigen.tuxfamily.org – URL: https://eigen.tuxfamily.org/index.php?title=Main\_Page
6. Factory Method Pattern // [Электронный ресурс]: habr.ru. — URL: https://habr.com/ru/articles/556512/
7. Shafiee, M., Amani, S. Optimal control for a class of singular systems using neural network // Iranian Journal of Science & Technology, Transaction B, Engineering. – 2005. – Vol. 29, No. B1. – P. 34–48.

# ПРИЛОЖЕНИЕ А код программы

**main.cpp**

#include <iostream>

#include "include/fabric.hpp"

#include "include/utils.hpp"

int main(int argc, char\* argv[]) {

try {

// Параметры интегрирования из параметров

Config cfg = parse\_cli(argc, argv);

// включаем многопоток

Eigen::setNbThreads(cfg.threads);

Eigen::MatrixXd E = read\_matrix\_from\_file("data/E.dat");

Eigen::MatrixXd A = read\_matrix\_from\_file("data/A.dat");

Eigen::MatrixXd B = read\_matrix\_from\_file("data/B.dat");

Eigen::MatrixXd Q = read\_matrix\_from\_file("data/Q.dat");

Eigen::MatrixXd initial\_P = Eigen::MatrixXd::Zero(E.rows(), E.cols());

// создает solver с переданным методом

auto solver = create\_solver(cfg.method, E, A, B, Q, initial\_P);

// Замеры времени

auto begin = std::chrono::system\_clock::now();

// Решаем

Result result = solver->solve(cfg);

auto end = std::chrono::system\_clock::now();

auto duration =

std::chrono::duration\_cast<std::chrono::milliseconds>(end - begin);

// Подставляем найденную матрицу в уравнение риккати -> ответ записываем

// в файл

solver->verify\_solution(result.P);

// результаты находятся в папке results

show\_results(cfg, result.last\_error, result.step, duration, result.P);

// Рисуем график если передали агрумент draw при запуске программы

if (cfg.draw) draw\_graph(result.points, cfg);

} catch (std::exception& e) {

std::cout << e.what() << '\n';

}

return 0;

}

**solver.cpp**

#include "include/solver.hpp"

#include <Eigen/Dense>

#include <deque>

#include <filesystem>

#include <fstream>

#include <iostream>

#include "include/utils.hpp"

// Наше матричное уравнение Риккати

inline Eigen::MatrixXd RiccatiSolver::riccati\_equation(

const Eigen::MatrixXd& P) {

return E\_ \* P \* A\_ + A\_.transpose() \* P \* E\_ + Q\_ - E\_ \* P \* BRB\_ \* P \* E\_;

}

int RiccatiSolver::get\_acceleration\_points(const std::string& method) {

if (method == "runge3" || method == "runge4" || method == "felberg4" ||

method == "felberg5" || method == "inglend4" || method == "inglend5") {

return 0;

}

if (method == "adams3") return 3;

if (method == "adams4") return 4;

if (method == "adams5") return 5;

if (method == "milna4") return 4;

if (method == "milna6") return 6;

if (method == "nystrom2") return 2;

if (method == "nystrom3") return 3;

if (method == "nystrom4") return 4;

if (method == "hemming4") return 4;

throw std::runtime\_error("(acc\_points)Неизвестный метод интегрирования: " +

method);

}

Result RiccatiSolver::solve(Config cfg) {

Eigen::MatrixXd P = initial\_P\_;

Eigen::MatrixXd P\_previous = P;

int step = 0;

double error = std::numeric\_limits<double>::max();

// если draw == true, то сохраняем точки для графика

std::vector<double>\* points =

cfg.draw ? new std::vector<double>() : nullptr;

int acc\_points = get\_acceleration\_points(cfg.method);

std::chrono::time\_point<std::chrono::system\_clock> begin, end;

// без этого метод Адамса не заработает

bool flag = false;

if (cfg.method.find("adams") != std::string::npos) flag = true;

while (step < cfg.max\_steps && error > cfg.target\_error) {

std::deque<Eigen::MatrixXd> prev\_points =

acceleration\_points(acc\_points, cfg.h, P);

if (cfg.step\_time) begin = std::chrono::system\_clock::now();

P\_previous = P;

for (double t = cfg.t0; t < cfg.t\_max; t += cfg.h) {

progress(cfg, error, step, t, P);

P = update\_step(P, cfg.h, prev\_points);

check\_nan(P, step, t);

if (flag) {

prev\_points.pop\_back();

prev\_points.push\_front(P);

}

}

error = (P - P\_previous).norm();

// значение ошибки на каждом шагу для графика

if (cfg.draw) points->push\_back(error);

if (cfg.step\_time) {

end = std::chrono::system\_clock::now();

auto duration =

std::chrono::duration\_cast<std::chrono::milliseconds>(end -

begin);

std::cout << duration.count() << "ms\n";

}

step++;

}

Result result{P, step, error, points};

return result;

}

// Найденную матрицу подставляем в уравнение Риккати и выводим в файл

Eigen::MatrixXd RiccatiSolver::verify\_solution(const Eigen::MatrixXd& P) {

namespace fs = std::filesystem;

Eigen::MatrixXd P\_verified = riccati\_equation(P);

fs::path folder\_path = "results";

fs::path P\_verified\_file\_path = folder\_path / "verified.txt";

if (!fs::exists(folder\_path)) fs::create\_directory(folder\_path);

std::ofstream verified(P\_verified\_file\_path);

if (verified.is\_open()) {

verified << std::fixed << std::setprecision(7);

verified << P\_verified << '\n';

verified.close();

} else {

std::cout << "Unable to open file\n";

}

return riccati\_equation(P);

}

// С помощью метода Рунге-Кутты получаем начальные точки для некоторых

// методов

std::deque<Eigen::MatrixXd> RiccatiSolver::acceleration\_points(

int count, double h, Eigen::MatrixXd& P\_initial) {

std::deque<Eigen::MatrixXd> result;

Eigen::MatrixXd k1, k2, k3, k4;

for (int i = 0; i < count; i++) {

k1 = h \* riccati\_equation(P\_initial);

k2 = h \* riccati\_equation(P\_initial + (k1 / 2.0));

k3 = h \* riccati\_equation(P\_initial + (k2 / 2.0));

k4 = h \* riccati\_equation(P\_initial + (k3));

P\_initial += ((k1 + 2.0 \* k2 + 2.0 \* k3 + k4) / 36.0);

result.push\_front(P\_initial);

}

return result;

}

std::deque<Eigen::MatrixXd> RiccatiSolver::acceleration\_points(int count,

double h) {

return acceleration\_points(count, h, initial\_P\_);

}

**include/solver.hpp**

#ifndef SOLVER\_HPP

#define SOLVER\_HPP

#include <Eigen/Dense>

#include <deque>

#include "utils.hpp"

// Абстрактный базовый класс для решателей уравнения Риккати

class RiccatiSolver {

public:

RiccatiSolver(Eigen::MatrixXd& E, Eigen::MatrixXd& A, Eigen::MatrixXd& B,

Eigen::MatrixXd& Q, Eigen::MatrixXd& initial\_P) {

this->E\_ = E;

this->A\_ = A;

this->BRB\_ = B \* B.transpose();

this->Q\_ = Q;

this->initial\_P\_ = initial\_P;

}

Result solve(Config cfg);

int get\_acceleration\_points(const std::string& method);

// Наше матричное уравнение Риккати

virtual Eigen::MatrixXd riccati\_equation(const Eigen::MatrixXd& P);

// Найденную матрицу подставляем в уравнение Риккати и выводим в файл

virtual Eigen::MatrixXd verify\_solution(const Eigen::MatrixXd& P);

// С помощью метода Рунге-Кутты получаем начальные точки для некоторых

// методов

virtual std::deque<Eigen::MatrixXd> acceleration\_points(

int count, double h, Eigen::MatrixXd& P\_initial);

virtual std::deque<Eigen::MatrixXd> acceleration\_points(int count,

double h);

virtual ~RiccatiSolver() {}

protected:

Eigen::MatrixXd E\_, A\_, BRB\_, Q\_, initial\_P\_;

// виртуальная функция, которую все методы реализуют для решения уравнения

virtual Eigen::MatrixXd update\_step(

const Eigen::MatrixXd& P, double h,

const std::deque<Eigen::MatrixXd>& prev\_points) = 0;

};

#endif

**include/methods.hpp**

#ifndef METHODS\_HPP

#define METHODS\_HPP

#include "solver.hpp"

template <typename Method>

class Solver : public RiccatiSolver {

public:

Solver(Eigen::MatrixXd& E, Eigen::MatrixXd& A, Eigen::MatrixXd& B,

Eigen::MatrixXd& Q, Eigen::MatrixXd& initial\_P)

: RiccatiSolver(E, A, B, Q, initial\_P) {}

private:

Eigen::MatrixXd update\_step(

const Eigen::MatrixXd& P, double h,

const std::deque<Eigen::MatrixXd>& prev) override {

return Method::step(\*this, P, h, prev);

};

};

struct RungeKutta3 {

static Eigen::MatrixXd step(RiccatiSolver& solver, const Eigen::MatrixXd& P,

double h, const std::deque<Eigen::MatrixXd>&) {

Eigen::MatrixXd k1, k2, k3;

k1.noalias() = solver.riccati\_equation(P);

k2.noalias() = solver.riccati\_equation(P + (0.5 \* h) \* k1);

k3.noalias() = solver.riccati\_equation(P - h \* k1 + 2 \* h \* k2);

return P + (h / 6.0) \* (k1 + 4 \* k2 + k3);

}

};

struct RungeKutta4 {

static Eigen::MatrixXd step(RiccatiSolver& solver, const Eigen::MatrixXd& P,

double h, const std::deque<Eigen::MatrixXd>&) {

Eigen::MatrixXd k1, k2, k3, k4;

k1.noalias() = solver.riccati\_equation(P);

k2.noalias() = solver.riccati\_equation(P + (0.5 \* h) \* k1);

k3.noalias() = solver.riccati\_equation(P + (0.5 \* h) \* k2);

k4.noalias() = solver.riccati\_equation(P + h \* k3);

return P + (h / 6.0) \* (k1 + 2 \* k2 + 2 \* k3 + k4);

}

};

struct Adams3 {

static Eigen::MatrixXd step(RiccatiSolver& solver, const Eigen::MatrixXd& P,

double h,

const std::deque<Eigen::MatrixXd>& prev) {

return P + (h / 12.0) \* ((23 \* solver.riccati\_equation(P)) -

(16 \* solver.riccati\_equation(prev[1])) +

(5 \* solver.riccati\_equation(prev[2])));

}

};

struct Adams4 {

static Eigen::MatrixXd step(RiccatiSolver& solver, const Eigen::MatrixXd& P,

double h,

const std::deque<Eigen::MatrixXd>& prev) {

return P + (h / 24.0) \* ((55 \* solver.riccati\_equation(P)) -

(59 \* solver.riccati\_equation(prev[1])) +

(37 \* solver.riccati\_equation(prev[2])) -

(9 \* solver.riccati\_equation(prev[3])));

}

};

struct Adams5 {

static Eigen::MatrixXd step(RiccatiSolver& solver, const Eigen::MatrixXd& P,

double h,

const std::deque<Eigen::MatrixXd>& prev) {

return P + (h / 720.0) \* ((1901 \* solver.riccati\_equation(P)) -

(2774 \* solver.riccati\_equation(prev[1])) +

(2616 \* solver.riccati\_equation(prev[2])) -

(1274 \* solver.riccati\_equation(prev[3])) +

(251 \* solver.riccati\_equation(prev[4])));

}

};

struct Felberg4 {

static Eigen::MatrixXd step(RiccatiSolver& solver, const Eigen::MatrixXd& P,

double h, const std::deque<Eigen::MatrixXd>&) {

Eigen::MatrixXd k1, k2, k3, k4, k5;

k1.noalias() = solver.riccati\_equation(P);

k2.noalias() = solver.riccati\_equation(P + (k1 \* (h / 4.0)));

k3.noalias() = solver.riccati\_equation(P + (3 \* h / 32.0) \* k1 +

(9 \* h / 32.0) \* k2);

k4.noalias() = solver.riccati\_equation(P + (1932 \* h / 2197.0) \* k1 -

(7200 \* h / 2197.0) \* k2 +

(7296 \* h / 2197.0) \* k3);

k5.noalias() = solver.riccati\_equation(

P + (439 \* h / 216.0) \* k1 - (8 \* h) \* k2 +

(3680 \* h / 513.0) \* k3 - (845 \* h / 4104.0) \* k4);

return P + h \* ((25 \* k1 / 216.0) + (1408 \* k3 / 2565.0) +

(2197 \* k4 / 4101.0) - (k5 / 5.0));

}

};

struct Felberg5 {

static Eigen::MatrixXd step(RiccatiSolver& solver, const Eigen::MatrixXd& P,

double h, const std::deque<Eigen::MatrixXd>&) {

Eigen::MatrixXd k1, k2, k3, k4, k5, k6;

k1.noalias() = solver.riccati\_equation(P);

k2.noalias() = solver.riccati\_equation(P + (k1 \* (h / 4.0)));

k3.noalias() = solver.riccati\_equation(P + (3 \* h / 32.0) \* k1 +

(9 \* h / 32.0) \* k2);

k4.noalias() = solver.riccati\_equation(P + (1932 \* h / 2197.0) \* k1 -

(7200 \* h / 2197.0) \* k2 +

(7296 \* h / 2197.0) \* k3);

k5.noalias() = solver.riccati\_equation(

P + (439 \* h / 216.0) \* k1 - (8 \* h) \* k2 +

(3680 \* h / 513.0) \* k3 - (845 \* h / 4104.0) \* k4);

k6.noalias() = solver.riccati\_equation(

P - (8 \* h / 27.0) \* k1 + 2 \* h \* k2 - (3544 \* h / 2565.0) \* k3 +

(1859 \* h / 4101.0) \* k4 - (11 \* h / 40.0) \* k5);

return P +

h \* ((16 / 135.0) \* k1 + (6656 / 12825.0) \* k3 +

(28561 / 56430.0) \* k4 - (9 / 50.0) \* k5 + (2 / 55.0) \* k6);

}

};

struct Inglend4 {

static Eigen::MatrixXd step(RiccatiSolver& solver, const Eigen::MatrixXd& P,

double h, const std::deque<Eigen::MatrixXd>&) {

Eigen::MatrixXd k1, k2, k3, k4;

k1.noalias() = solver.riccati\_equation(P);

k2.noalias() = solver.riccati\_equation(P + (k1 \* (h / 2.0)));

k3.noalias() = solver.riccati\_equation(P + (h / 4.0) \* (k1 + k2));

k4.noalias() = solver.riccati\_equation(P + h \* (2 \* k3 - k2));

return P + (h / 6.0) \* (k1 + 4 \* k3 + k4);

}

};

struct Inglend5 {

static Eigen::MatrixXd step(RiccatiSolver& solver, const Eigen::MatrixXd& P,

double h, const std::deque<Eigen::MatrixXd>&) {

Eigen::MatrixXd k1, k2, k3, k4, k5, k6;

k1.noalias() = solver.riccati\_equation(P);

k2.noalias() = solver.riccati\_equation(P + (k1 \* (h / 2.0)));

k3.noalias() = solver.riccati\_equation(P + (h / 4.0) \* (k1 + k2));

k4.noalias() = solver.riccati\_equation(P + h \* (2 \* k3 - k2));

k5.noalias() =

solver.riccati\_equation(P + (h / 27.0) \* (7 \* k1 + 10 \* k2 + k4));

k6.noalias() = solver.riccati\_equation(

P +

(h / 625.0) \* (28 \* k1 - 125 \* k2 + 546 \* k3 + 54 \* k4 - 378 \* k5));

return P + (h / 336.0) \* (14 \* k1 + 35 \* k4 + 162 \* k5 + 125 \* k6);

}

};

struct Nystrom2 {

static Eigen::MatrixXd step(RiccatiSolver& solver, const Eigen::MatrixXd& P,

double h,

const std::deque<Eigen::MatrixXd>& prev) {

return prev[1] + 2 \* h \* solver.riccati\_equation(P);

}

};

struct Nystrom3 {

static Eigen::MatrixXd step(RiccatiSolver& solver, const Eigen::MatrixXd& P,

double h,

const std::deque<Eigen::MatrixXd>& prev) {

return prev[1] + (h / 3.0) \* (7 \* solver.riccati\_equation(P) -

2 \* solver.riccati\_equation(prev[1]) +

solver.riccati\_equation(prev[2]));

}

};

struct Nystrom4 {

static Eigen::MatrixXd step(RiccatiSolver& solver, const Eigen::MatrixXd& P,

double h,

const std::deque<Eigen::MatrixXd>& prev) {

return prev[1] + (h / 3.0) \* (8 \* solver.riccati\_equation(P) -

5 \* solver.riccati\_equation(prev[1]) +

4 \* solver.riccati\_equation(prev[2]) -

solver.riccati\_equation(prev[3]));

}

};

struct Milna4 {

static Eigen::MatrixXd step(RiccatiSolver& solver, const Eigen::MatrixXd& P,

double h,

const std::deque<Eigen::MatrixXd>& prev) {

return prev[3] +

((4 \* h / 3.0) \* (2 \* solver.riccati\_equation(P) -

solver.riccati\_equation(prev[1]) +

2 \* solver.riccati\_equation(prev[2])));

}

};

struct Milna6 {

static Eigen::MatrixXd step(RiccatiSolver& solver, const Eigen::MatrixXd& P,

double h,

const std::deque<Eigen::MatrixXd>& prev) {

return prev[5] +

(3 \* h / 10.0) \* (11 \* solver.riccati\_equation(P) -

14 \* solver.riccati\_equation(prev[1]) +

26 \* solver.riccati\_equation(prev[2]) -

14 \* solver.riccati\_equation(prev[3]) +

11 \* solver.riccati\_equation(prev[4]));

}

};

struct Hemming4 {

static Eigen::MatrixXd step(RiccatiSolver& solver, const Eigen::MatrixXd& P,

double h,

const std::deque<Eigen::MatrixXd>& prev) {

return (0.5 \* (P + prev[1])) +

(h / 48.0) \* (119 \* solver.riccati\_equation(P) -

99 \* solver.riccati\_equation(prev[1]) +

69 \* solver.riccati\_equation(prev[2]) -

17 \* solver.riccati\_equation(prev[3]));

}

};

#endif

**include/fabric.hpp**

#ifndef FABRIC\_HPP

#define FABRIC\_HPP

#include <Eigen/Dense>

#include <memory>

#include "methods.hpp"

inline std::unique\_ptr<RiccatiSolver> create\_solver(

std::string& name, Eigen::MatrixXd& E, Eigen::MatrixXd& A,

Eigen::MatrixXd& B, Eigen::MatrixXd& Q, Eigen::MatrixXd& P0) {

if (name == "runge3")

return std::make\_unique<Solver<RungeKutta3>>(E, A, B, Q, P0);

if (name == "runge4")

return std::make\_unique<Solver<RungeKutta4>>(E, A, B, Q, P0);

if (name == "adams3")

return std::make\_unique<Solver<Adams3>>(E, A, B, Q, P0);

if (name == "adams4")

return std::make\_unique<Solver<Adams4>>(E, A, B, Q, P0);

if (name == "adams5")

return std::make\_unique<Solver<Adams5>>(E, A, B, Q, P0);

if (name == "milna4")

return std::make\_unique<Solver<Milna4>>(E, A, B, Q, P0);

if (name == "milna6")

return std::make\_unique<Solver<Milna6>>(E, A, B, Q, P0);

if (name == "nystrom2")

return std::make\_unique<Solver<Nystrom2>>(E, A, B, Q, P0);

if (name == "nystrom3")

return std::make\_unique<Solver<Nystrom3>>(E, A, B, Q, P0);

if (name == "nystrom4")

return std::make\_unique<Solver<Nystrom4>>(E, A, B, Q, P0);

if (name == "hemming4")

return std::make\_unique<Solver<Hemming4>>(E, A, B, Q, P0);

if (name == "felberg4")

return std::make\_unique<Solver<Felberg4>>(E, A, B, Q, P0);

if (name == "felberg5")

return std::make\_unique<Solver<Felberg5>>(E, A, B, Q, P0);

if (name == "inglend4")

return std::make\_unique<Solver<Inglend4>>(E, A, B, Q, P0);

if (name == "inglend5")

return std::make\_unique<Solver<Inglend5>>(E, A, B, Q, P0);

throw std::runtime\_error("Неизвестный метод интегрирования: " + name);

}

#endif

**utils.cpp**

#include "include/utils.hpp"

#include <CLI/CLI.hpp>

#include <Eigen/Dense>

#include <chrono>

#include <filesystem>

#include <fstream>

#include <iostream>

#include <string>

namespace fs = std::filesystem;

// Ввод одной матрицы в файл

Eigen::MatrixXd read\_matrix\_from\_file(const std::string& filepath) {

std::ifstream file(filepath);

if (!file.is\_open()) {

throw std::runtime\_error("Не удалось открыть файл " + filepath);

}

std::vector<std::vector<double>> data;

std::string line;

size\_t cols = 0; // Количество столбцов (определяется по первой строке)

while (std::getline(file, line)) {

// Пропускаем пустые строки

if (line.empty()) continue;

std::istringstream iss(line);

std::vector<double> row;

std::string token;

while (iss >> token) {

// Проверяем, что число состоит только из цифр, знака и точек

bool is\_valid = !token.empty();

bool has\_digit = false;

size\_t sign\_pos = (token[0] == '-' || token[0] == '+') ? 1 : 0;

size\_t dot\_count = 0;

for (size\_t i = sign\_pos; i < token.size(); ++i) {

if (token[i] == '.') {

dot\_count++;

// Не больше одной точки в числе

if (dot\_count > 1) {

is\_valid = false;

break;

}

} else if (!isdigit(token[i])) {

is\_valid = false;

break;

} else {

has\_digit = true;

}

}

if (!is\_valid || !has\_digit) {

throw std::runtime\_error("Обнаружен лишний символ: " + token);

}

try {

double value = std::stod(token);

row.push\_back(value);

} catch (...) {

throw std::runtime\_error("Ошибка обработки: " + token);

}

}

if (row.empty()) {

continue; // Пропускаем строки без чисел

}

// Проверяем согласованность количества столбцов

if (cols == 0) {

cols = row.size();

} else if (row.size() != cols) {

throw std::runtime\_error("Несовпадение количества столбцов в строке: " + line);

}

data.push\_back(row);

}

file.close();

if (data.empty()) {

throw std::runtime\_error("Файл " + filepath + " не содержит числовых данных.");

}

// Преобразуем в Eigen::MatrixXd

Eigen::MatrixXd mat(data.size(), cols);

for (size\_t i = 0; i < data.size(); ++i) {

for (size\_t j = 0; j < data[i].size(); ++j) {

mat(static\_cast<long>(i), static\_cast<long>(j)) = data[i][j];

}

}

return mat;

}

// Вывод прогресса раз в 2 секунды

void progress(Config cfg, double error, int step, double t,

Eigen::MatrixXd& P) {

static auto start\_time = std::chrono::system\_clock::now();

auto current\_time = std::chrono::system\_clock::now();

auto elapsed\_time = std::chrono::duration\_cast<std::chrono::seconds>(

current\_time - start\_time);

if (cfg.manual || elapsed\_time.count() >= 2) {

std::cout << "\033[H\033[2J";

std::cout << "Метод: " << cfg.method

<< " | Требуемая ошибка: " << cfg.target\_error

<< " | Ошибка: " << error << "\t\r\nh: " << cfg.h

<< " | Шаг : " << step << " | t : " << t << " из "

<< cfg.t\_max << "\t\t\n";

std::cout << "P(первые 8x8 элементов):\n";

int rows\_to\_show = std::min(static\_cast<int>(P.rows()), 8);

int cols\_to\_show = std::min(static\_cast<int>(P.cols()), 8);

for (int i = 0; i < rows\_to\_show; ++i) {

for (int j = 0; j < cols\_to\_show; ++j) {

std::cout << std::setw(10) << std::setprecision(4) << P(i, j)

<< " ";

}

std::cout << "\n";

}

start\_time = current\_time;

}

int ch;

while (cfg.manual) {

ch = std::cin.get();

if (ch == '\n') break;

}

}

// записывает результаты в файл и выводит в терминал

void show\_results(Config cfg, double error, double steps,

std::chrono::milliseconds elapsed\_time, Eigen::MatrixXd& P) {

fs::path folder\_path = "results";

fs::path result\_file\_path = folder\_path / "output.txt";

if (!fs::exists(folder\_path)) fs::create\_directory(folder\_path);

std::ofstream out(result\_file\_path);

if (out.is\_open()) {

out << cfg.method << " ; t = " << cfg.t0 << " ; t\_end = " << cfg.t\_max

<< " ; h = " << cfg.h << " ; error\_value = " << cfg.target\_error

<< "\n"

<< "error = " << error << " ; last\_step = " << steps

<< "\ntime: " << elapsed\_time.count() / 1000 << " sec. "

<< elapsed\_time.count() % 1000 << " ms."

<< "\n\n";

out << std::fixed << std::setprecision(4);

out << P << "\n";

out.close();

} else {

std::cout << "Unable to open file\n";

}

std::cout << "\033[2J\033[H\nИсходные данные: ";

std::cout << cfg.method << "; t = " << cfg.t0 << " ; t\_end = " << cfg.t\_max

<< " ; h = " << cfg.h << " ; error\_value = " << cfg.target\_error

<< "\n";

std::cout << "Последняя ошибка = " << error << " ; Шаг = " << steps << "\n";

std::cout << "\033[32mMатрица P записана в output.txt\033[0m\n";

std::cout << "Время работы программы: " << elapsed\_time.count() / 1000

<< " сек. " << elapsed\_time.count() % 1000 << " мс.\n";

}

// Рисует график если флаг draw == true

void draw\_graph(std::vector<double>\* error, Config cfg) {

FILE\* gnuplot = popen("gnuplot -persist", "w");

if (!gnuplot) {

std::cerr << "gnuplot не удалось запустить\n";

return;

}

fs::path folder = "results/png";

if (!fs::exists(folder)) fs::create\_directory(folder);

fprintf(gnuplot, "set terminal pngcairo\n");

fprintf(gnuplot, "set output 'results/png/%s.png'\n", cfg.method.c\_str());

fprintf(gnuplot, "set title \"Ошибка на каждом шаге\"\n");

fprintf(gnuplot, "set xlabel 'Шаги'\n");

fprintf(gnuplot, "set ylabel 'Ошибка'\n");

fprintf(gnuplot, "plot '-' with lines title 'error'\n");

for (size\_t i = 0; i < error->size(); i++) {

fprintf(gnuplot, "%ld %f\n", i, error->at(i));

}

fprintf(gnuplot, "e\n");

fprintf(gnuplot, "set output\n");

pclose(gnuplot);

}

void check\_nan(const Eigen::MatrixXd& P, int step, double t) {

if (std::isnan(P(0, 0))) {

std::string message =

"Матрица P заполнилась -nan\nШаг : " + std::to\_string(step) +

" | t : " + std::to\_string(t) + "\n";

throw std::runtime\_error(message);

}

}

Config parse\_cli(int argc, char\*\* argv) {

Config opts;

CLI::App app{"Riccati Equation Solver"};

app.add\_option("--method", opts.method,

"Метод решения (runge3, runge4,"

"adams3, adams4, adams5, "

"milna4, milna6, "

"nystrom2, nystrom3, nystrom4, "

"felberg4, felberg5, "

"inglend4, inglend5, "

"hemming4)")

->required();

app.add\_option("--h", opts.h, "Шаг интегрирования >= 0");

app.add\_option("--t0", opts.t0, "Начальное время (по умолчанию 0)");

app.add\_option("--t\_max", opts.t\_max, "Конечное время (по умолчанию 10)");

app.add\_option("--error", opts.target\_error,

"Требуемая погрешность (по умолчанию 0.001)");

app.add\_option("--max\_steps", opts.max\_steps,

"Максимум шагов (по умолчанию 200)");

app.add\_option("--threads", opts.threads,

"Количество потоков (по умолчанию 1)\n"

"Для небольших матриц большое кол-во потоков негативно "

"повлияют на скорость.");

app.add\_flag("--draw", opts.draw,

"Рисовать график ошибки (использует gnuplot)");

app.add\_flag("--manual", opts.manual, "Пошаговое выполнение вычислений");

app.add\_flag("--step\_time", opts.step\_time,

"Отображать время выполнения одного шага");

if (argc < 2) {

std::cout << app.help() << "\n";

exit(0);

}

try {

app.parse(argc, argv);

} catch (const CLI::ParseError& e) {

app.exit(e);

exit(1);

}

return opts;

}

**include/utils.hpp**

#ifndef UTILS\_HPP

#define UTILS\_HPP

#include <Eigen/Dense>

#include <chrono>

#include <string>

#include <vector>

struct Config {

double t0 = 0.0;

double t\_max = 10.0;

double h = 0.01;

double target\_error = 0.001;

int max\_steps = 200;

int threads = 1;

bool draw = false;

bool manual = false;

bool step\_time = false;

std::string method = "runge4";

};

// Структура для хранения результата

struct Result {

Eigen::MatrixXd P;

int step;

double last\_error;

std::vector<double>\* points;

};

Config parse\_cli(int argc, char\* argv[]);

// Чтение матрицы из файла

Eigen::MatrixXd read\_matrix\_from\_file(const std::string& filepath);

// Вывод прогресса вычислений

void progress(Config cfg, double error, int step, double t, Eigen::MatrixXd& P);

// Вывод результатов вычислений

void show\_results(Config cfg, double error, double steps,

std::chrono::milliseconds elapsed\_time, Eigen::MatrixXd& P);

// Построение графика ошибки (если разрешено)

void draw\_graph(std::vector<double>\* error, Config cfg);

// Проверка на NaN в матрице

void check\_nan(const Eigen::MatrixXd& P, int step, double t);

#endif

**CmakeLists.txt**

cmake\_minimum\_required(VERSION 3.10)

project(RiccatiSolver)

set(CMAKE\_CXX\_STANDARD 17)

set(CMAKE\_EXPORT\_COMPILE\_COMMANDS ON)

add\_executable(solver main.cpp utils.cpp solver.cpp)

# Добавляем флаг оптимизации

if(NOT CMAKE\_BUILD\_TYPE)

set(CMAKE\_BUILD\_TYPE Release CACHE STRING "Build type" FORCE)

endif()

set(CMAKE\_CXX\_FLAGS\_RELEASE "${CMAKE\_CXX\_FLAGS\_RELEASE} -O3")

find\_package(OpenMP QUIET)

if(OpenMP\_CXX\_FOUND)

set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} ${OpenMP\_CXX\_FLAGS}")

else()

message(STATUS "OpenMP not found. Parallelization disabled.")

endif()

# Ищем библиотеку Eigen3, если Eigen не найден, загружаем его с помощью FetchContent

find\_package(Eigen3 QUIET)

if(NOT Eigen3\_FOUND)

message(STATUS "Eigen not found in system. Downloading Eigen...")

include(FetchContent)

FetchContent\_Declare(

Eigen

GIT\_REPOSITORY https://gitlab.com/libeigen/eigen.git

GIT\_TAG 3.4.0

)

FetchContent\_MakeAvailable(Eigen)

else()

message(STATUS "Eigen found in system. Using installed version.")

endif()

#CLI11

find\_package(CLI11 QUIET)

if(NOT CLI11\_FOUND)

message(STATUS "CLI11 not found in system. Downloading CLI11...")

include(FetchContent)

FetchContent\_Declare(

CLI11

GIT\_REPOSITORY https://github.com/CLIUtils/CLI11.git

GIT\_TAG v2.5.0

)

FetchContent\_MakeAvailable(CLI11)

target\_link\_libraries(solver PUBLIC CLI11::CLI11)

else()

message(STATUS "CLI11 found in system. Using installed version.")

endif()

message(STATUS "Eigen include dir: ${EIGEN3\_INCLUDE\_DIR}")

target\_link\_libraries(solver PUBLIC Eigen3::Eigen CLI11::CLI11)

**build.sh**

#!/bin/sh

cmake -S ./1` -B ./.build

cmake --build ./.build --parallel $(nproc)

cp .build/solver ./