**СОДЕРЖАНИЕ**

[СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ 3](#__RefHeading___Toc1785_3851806917)

[ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ 4](#__RefHeading___Toc1787_3851806917)

[ВВЕДЕНИЕ 5](#__RefHeading___Toc1789_3851806917)

[1 Задача оптимального управления для сингулярных линейных систем 7](#__RefHeading___Toc1793_3851806917)

[1.1 Теория оптимального управления линейных систем 7](#__RefHeading___Toc8379_387698889)

[1.2 Условия корректности задачи LQR для сингулярных систем 10](#__RefHeading___Toc9758_3653515131)

[1.3 Выводы 11](#__RefHeading___Toc9760_3653515131)

[2 нейронные оду для решения уравнения Риккати 12](#__RefHeading___Toc1795_3851806917)

[2.1 Описание 12](#__RefHeading___Toc6644_387698889)

[2.2 Описание методов численного интегрирования ОДУ 12](#__RefHeading___Toc1243_619138733)

[2.2.1 Метод Рунге-Кутты 12](#__RefHeading___Toc659_463851313)

[2.2.2 Метод Адамса-Башфорта 12](#__RefHeading___Toc663_463851313)

[2.2.3 Метод Фельберга 13](#__RefHeading___Toc665_463851313)

[2.2.4 Метод Ингленда 14](#__RefHeading___Toc1809_3851806917_%25252)

[2.2.5 Метод Нюстрема 14](#__RefHeading___Toc1809_3851806917_%25251)

[2.2.6 Метод Милны 15](#__RefHeading___Toc1809_3851806917_%25253)

[2.2.7 Метод Хемминга 15](#__RefHeading___Toc1809_3851806917_%25254)

[2.3 Выводы 15](#__RefHeading___Toc1805_3851806917)

[3 архитектура и реализация программы 16](#__RefHeading___Toc1807_3851806917)

[3.1 Проектирование и архитектура программной системы 16](#__RefHeading___Toc1809_3851806917)

[3.2 UML? 20](#__RefHeading___Toc1811_3851806917)

[3.3 Инструментарий 20](#__RefHeading___Toc1813_3851806917)

[3.4 Планирование 20](#__RefHeading___Toc1815_3851806917)

[3.5 Выводы 20](#__RefHeading___Toc1817_3851806917)

[4 исследование скорости работы методов 21](#__RefHeading___Toc1819_3851806917)

[4.2 Выводы 21](#__RefHeading___Toc669_463851313)

[ЗАКЛЮЧЕНИЕ 22](#__RefHeading___Toc1827_3851806917)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 23](#__RefHeading___Toc1829_3851806917)

[ПРИЛОЖЕНИЕ А 24](#__RefHeading___Toc1831_3851806917)

# СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

LQR — линейно-квадратичный регулятор

ОДУ — обыкновенное дифференциальное уравнение

CLI — интерфейс командной строки

# ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

Уравнение Рикатти — обыкновенное дифференциальное уравнение первого порядка, правая часть которого является многочленом второй степени.

Динамическая мода — это характеристика линейной динамической системы, связанная с её собственными значениями и собственными векторами, определяющая форму и скорость изменения состояния системы во времени. В контексте LQR динамические моды описывают поведение замкнутой системы после применения оптимального управления.

# 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 E:

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 неопределённости или частичной информации о системе — в таких областях, 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 цену функционирования системы, при этом обеспечивая устойчивость и достижение цели управления.

Для оценки качества управления вводится следующий функционал []:

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

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

- положительно определенная матрица, характеризующая стоимость управляющих усилий;

- симметричная положительно полуопределенная матрица

- фиксированные конечный момент времени;

- свободен, 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) |

Предполагая, что сопряженная переменная связана с состоянием через матрицу :

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

Тогда подставляя в (5) и (6), после математических преобразований можно получить матричное уравнение Риккати, адаптированное под сингулярную структуру системы:

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

После получения решения обобщенного матричного уравнения Риккати (8), оптимальное воздействие в задаче LQR для системы с сингулярной матрицей формируется в виде состояния-обратной связи:

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

где матрица усиления определяется выражением:

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

Поскольку матрица однозначно определяется через решение уравнения (8), а также известны параметры системы известны, 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 (8) и (9). Результатом будет управляющее воздействие которое подается на исполнительные механизмы системы. Такая реализация обеспечит устойчивость системы при соблюдении уловий 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 невырожденна, то система имеет только конечные моды, а если вырождена, то появляются бесконечных мод, где . Отстутствие 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.

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 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 полетом 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

## Описание методов численного интегрирования ОДУ

### Метод Рунге-Кутты

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

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

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

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

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:

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

где коэффициенты определяются согласно (17)

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

### Метод Адамса-Башфорта

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

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

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

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

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

Для начала расчетов требуются четыре «разгонные» точки .

В многошаговом методе Адамса-Башформа пятого порядка точности для нахождения точки используется пять предыдущих точек:

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

Для начала расчетов требуются пять «разгонных» точек .

Методы Адамса-Башформа 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 ситсемы 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:

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

где

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

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

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

где коэффициенты определяются согласно (22)

### Метод Ингленда

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

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

где

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

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

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

где коэффициенты определяются согласно (25)

### Метод Нюстрема

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

|  | | (27) |
| --- | --- | --- |
|  | (28) | |
|  | (29) | |

Для начала расчетов по формулам (27) - (29) требуются две, три и четыре «разгонные» точки соответственно.

### Метод Милны

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

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

Для начала расчетов по формуле (30) требуется четыре «разгонные» точки , которые могут быть найдены любым из предыдущих методов.

В методе Милна шестого порядка точности для расчета точки используется шесть предыдущих точкек:

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

### Метод Хемминга

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

|  | (32) |
| --- | --- |

## Выводы

# архитектура и реализация программы

## Проектирование и архитектура программной системы

Разработка программного обеспечения для решения матричного уравнения Риккати потребовала построения гибкой и расширяемой архитектуры, поддерживающей использование различных методов интегрирования и обеспечивающей управление параметрами моделирования. В данной работе реализован модульный подход на языке программирования C++ с использованием объектно-ориентированного проектирования.

В основу архитектуры легли следующие принципы:

* Разделение ответственности: каждая часть программы выполняет строго определённую роль — чтение входных данных, решение уравнения, визуализация и т.д.
* Расширяемость: с помощью шаблонов и абстракций возможно добавление новых численных методов без модификации основной логики.
* Повторное использование кода: за счёт обобщённых решений и применения шаблонных классов.

Программа разделена на несколько ключевых модулей, обеспечивающих гибкость и масштабируемость:

1. Модуль запуска (*main.cpp*)

Отвечает за

* инициализицию 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 запускного файла;
* выбирается численный метод через ***create\_solver*,**
* запускает алгоритм решения;
* отображение хода интегрирования и сохранение результатов;

1. 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\_max,* 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.

1. 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:

* реализацию уравнения Риккати
* алгоритм решения *solve(Config)* в котором циклически обновляется матрица
* абстрактный метод *update\_step*, реализуемый в потомках — он выполняет один шаг интегрирования
* алгоритм подстановки найденной матрицы в уравнение Риккати для верификации

1. Шаблонный класс (*Sovler<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). Каждый метод реализуется как структура с одной функцией *step.*

1. Фабрика решателей (*fabric.hpp*)

Файл *fabric.hpp* содержит 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 численного ядра, она создаёт объект *Solver<Method>* в зависимости от строки, переданной в аргументах (runge4, adams5 и др.).

Она позволяет добавлять новые методы, просто расширяя список в fabric.hpp

1. 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 классе *Solver<Method>.*

1. Вспомогательные функции (*utils.cpp/hpp*)

* Загрузка матрицы из файлов
* Построение графика ошибки с помощью gnuplot
* Отображение прогресса в консоль
* Запись результатов вычислений в файл
* Проверка матрицы на содержание некорректных (бесконечно больших) значений

**Модуль запуска**

**Конфигурационный модуль**

**Абстрактный решатель**

*RiccatiSolver* определяет общие данные и операции, которые необходимы для любого метода решения уравнения Риккати:

Хранение исходных матриц системы

Реализация самого уравнения Риккати ()

Алгоритм пошагового решения (циклическое обновление матрицы P)

Абстрактный метод *update\_step()* — реализуется в каждом методе (Runge, Adams и др.)

Методы верификации, визуализации и инициализации

Описание методов:

* *riccati\_equation(P)* Реализует правую часть матричного уравнения Риккати
* *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
* *acceleration\_points()* Для многошаговых методов (например, Адамса, Милна) требуется 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**

**Фабрика решателей**

**Методы интегрирования**

**Вспомогательные функции**

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 ошибок (вызывается через popen())

Поток выполнения программы

1. Пользователь запускает программу с нужными параметрами:

|  |
| --- |
| ./solver --method runge4 --h 0.01 --t\_max 10 --error 0.001 --draw |

1. Конфигурация сохраняется в Config.
2. Загружаются матрицы data/E.dat и т.д.
3. Создаётся решатель соответствующего метода (Solver<RungeKutta4>).
4. Вычисляется решение уравнения Риккати.
5. Проверяется корректность решения.
6. Результаты сохраняются, при необходимости строится график ошибки.

Принципы проектирования:

* 1. Объектно-ориентированных подход:
  2. Шаблон проектирования «Фабрика»
  3. Разделение интерфейса и реализации
  4. Использование STL и Eigen:

Взаимодействие модулей

* 1. Последовательность работы
  2. Поток данных
  3. Обратная связь

Обработка ошибок

* 1. Валидация на этапе ввода
  2. Контроль вычислений
  3. Исключения

Гибкость архитектуры

* 1. Легкое добавление новых методов
  2. Настройка через CLI

## UML?

## Инструментарий

## Планирование

## Выводы

# исследование скорости работы методов

## Выводы

+

# ЗАКЛЮЧЕНИЕ

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. 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.
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. Квакернаак, Х., Сиван, Р.. Линейные оптимальные системы управления. / Х. Квакернаак — Москва: Изд-во Мир, 1977. — 656 c.
4. 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.
5. Eigen / линейная алгебра на C++ [Электронный ресурс]. – URL: https://eigen.tuxfamily.org/ (дата обращения: 14.05.2025).
6. CLI11 / парсер аргументов командной строки для C++ [Электронный ресурс] : официальный репозиторий. – URL: https://github.com/CLIUtils/CLI11 (дата обращения: 14.05.2025).

# ПРИЛОЖЕНИЕ А

Код приложения

**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;

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

std::istringstream iss(line);

std::vector<double> row;

double value;

while (iss >> value) row.push\_back(value);

if (!row.empty()) data.push\_back(row);

}

file.close();

if (data.empty()) {

throw std::runtime\_error("Файл " + filepath +

" пустой или содержит неверные данные.");

}

Eigen::MatrixXd mat(data.size(), data[0].size());

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

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

mat((long)i, (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 src/ -B ./.build

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

cp .build/solver ./