|  |  |  |
| --- | --- | --- |
|  | | |
| **Лабораторная работа 2** | | |
| по дисциплине «Разработка программного обеспечения для моделирования физических процессов» | | |
| Выполнил |  |  |
| студент гр.5130904/10101 |  | Абраамян А. М. |
| Руководитель |  | Воскобойников С. П. |
|  |  |  |

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_202\_\_ г.

**Оглавление**

Оглавление

[Постановка задачи 3](#__RefHeading___Toc1377_811955146)

[Дискретная модель 3](#__RefHeading___Toc1379_811955146)

[Коэффициенты 5](#__RefHeading___Toc1381_811955146)

[Решение системы ОДУ 6](#__RefHeading___Toc1383_811955146)

[Явный метод ломаных Эйлера 6](#__RefHeading___Toc1264_4143828341)

[Неявный метод ломаных Эйлера 7](#__RefHeading___Toc1266_4143828341)

[Метод неявного Эйлера с тридиагональной матрицей 7](#__RefHeading___Toc6329_4088301765)

[Обзор метода неявного Эйлера 7](#__RefHeading___Toc6331_4088301765)

[Тридиагональная матрица 8](#__RefHeading___Toc6333_4088301765)

[Метод Томаса 8](#__RefHeading___Toc6335_4088301765)

[Метод Томаса для неявного метода Эйлера 9](#__RefHeading___Toc6337_4088301765)

[Жёсткость системы 10](#__RefHeading___Toc1268_4143828341)

[Тестирование 10](#__RefHeading___Toc1387_811955146)

[Пример 1 10](#__RefHeading___Toc1389_811955146)

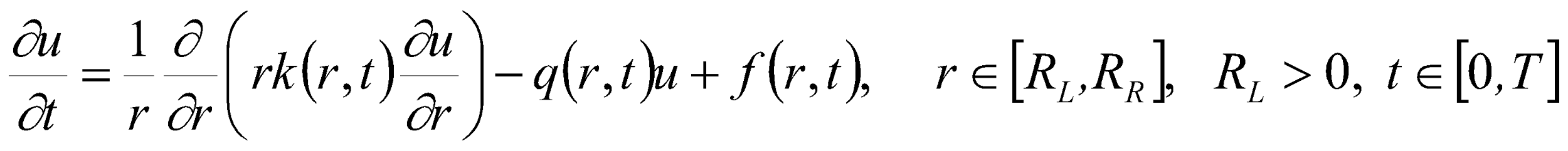
[Пример 2 12](#__RefHeading___Toc1391_811955146)

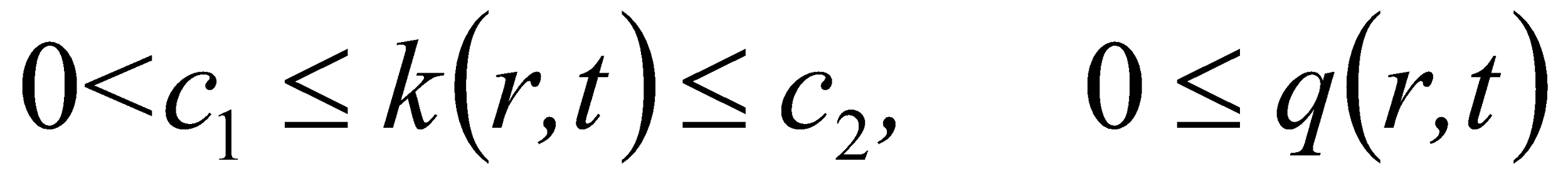
[Вывод 13](#__RefHeading___Toc1393_811955146)

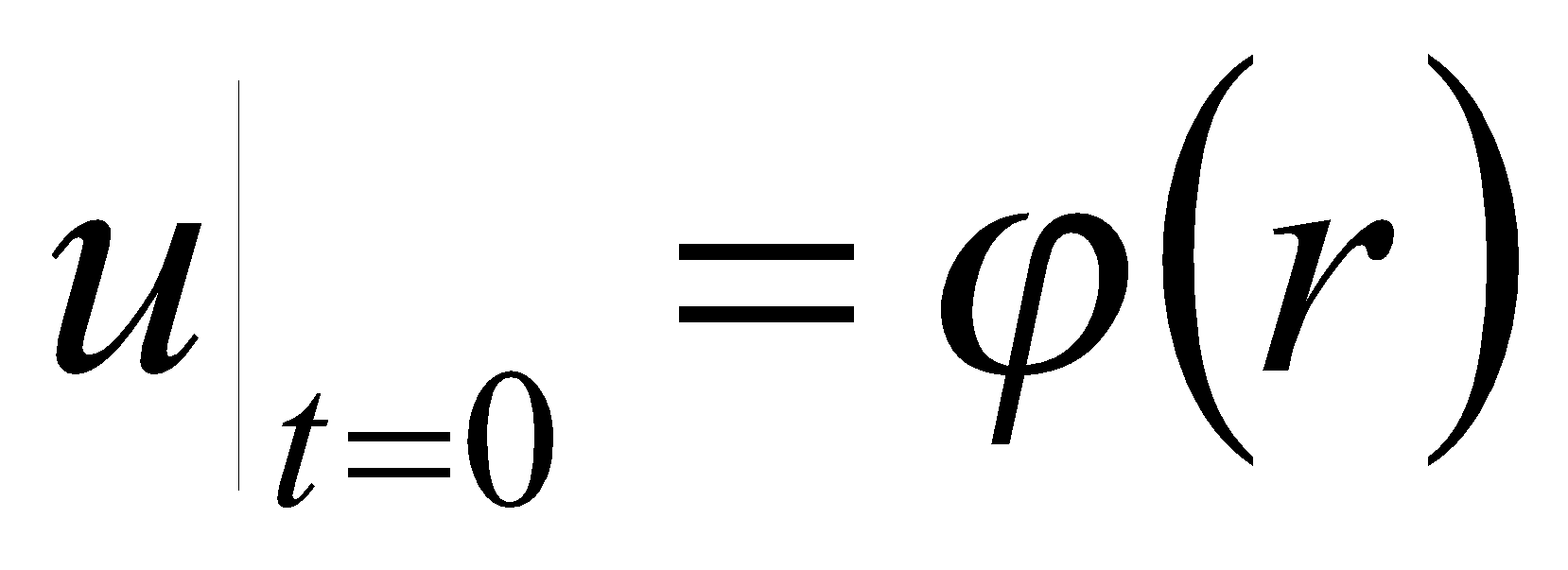
[Код 13](#__RefHeading___Toc1395_811955146)

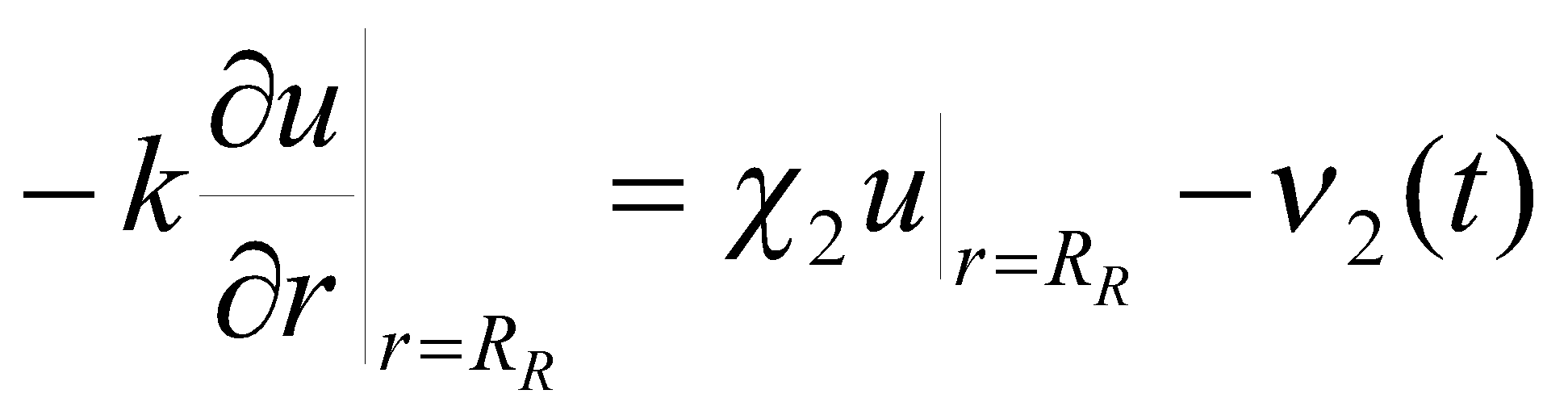
# Постановка задачи

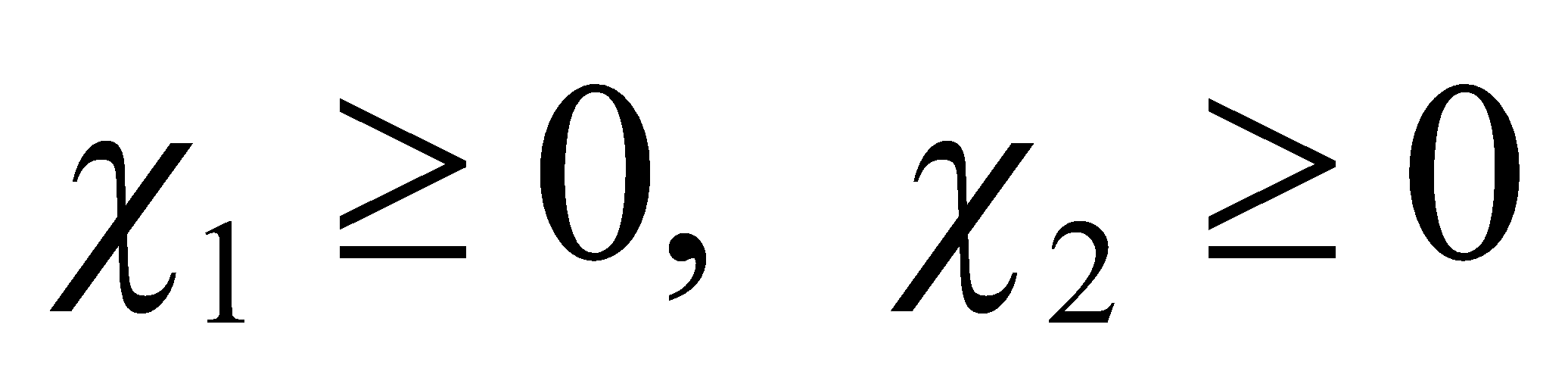
Используя интегро-интерполяционный метод (метод баланса), разработать программу для моделирования нестационарного распределения температуры в полом цилиндре, описываемого математической моделью вида:





начальным условием вида и граничными условиями вида:





Для построения и тестирования модели будет использоваться язык C++.

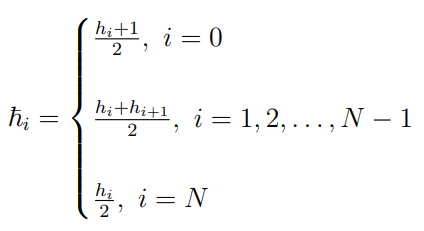
# Дискретная модель

Введём обозначения:

*N -* число разбиений интервала [Rl, Rr]

*hi* = *ri − ri−*1

*ri−1/2*=(*ri + ri−*1)*/2*



Домножим уравнение на r:

**Проинтегрируем уравнение для промежутка, не включая границы:**

*i = 1,2,...,N-1*

Интегро-дифференциальное тождество:

По формуле центральных разностей:

По формуле **средних** прямоугольников:

Разностная схема:

*i = 1,2,...,N-1*

Аппроксимация граничного условия справа:

*i = N*

Теперь используем формулу **правых** прямоугольников для аппроксимации интеграла:

Используя наше граничное условие справа:

*i =N*

Приведём подобные слагаемые для разностных схем:

*i = 1,2,...,N-1*

*i = N*

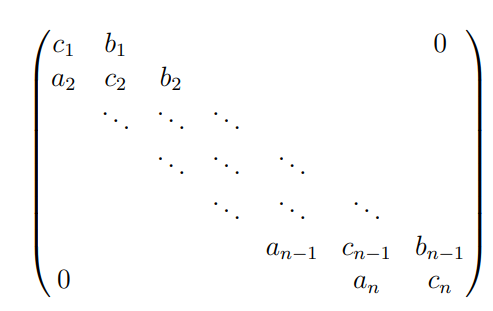
Поскольку нам дано граничное условие слева первого рода, то мы поступим следующим образом. Используем его для уравнение i = 1. Буквально возьмем и подставим вместо . Таким образом наш коэффициент просто перейдет просто перейдет в коэффициент как константа, а матрица станет на одну строчку короче чем была прежде.

# Коэффициенты

Теперь данную систему можно представить в виде:

(1)

Где *A* - трёхдиагональная матрица вида:



Коэффициенты соответственно следующие:

Для коэффициентов у нас не будет.

Решить систему численным методом - найти вектор *v* и сравнить его с точным решением *u.*

# Решение системы ОДУ

Чтобы решить систему дифференциальных уравнений, введём дискретизацию по времени и проинтегрируем:



Добавим начальное условие и получим систему:

## Явный метод ломаных Эйлера

Аппроксимируем интеграл по формуле левых прямоугольников , получаем:

## Неявный метод ломаных Эйлера

Аппроксимируем интеграл по формуле правых прямоугольников , получаем:



Это привычная запись неявного метода Эйлера. Однако надо понимать, что вычисление обратной матрицы очень трудоемкая операция, которую можно легко избежать зная что матрица А трехдиагональная

# Метод неявного Эйлера с тридиагональной матрицей

Метод неявного Эйлера (или метод "Backward Euler") используется для решения дифференциальных уравнений, особенно жестких систем. Это неявный метод, который требует решения системы линейных уравнений для получения решения на каждом шаге.

## Обзор метода неявного Эйлера

Для дифференциального уравнения вида:

=Au+b

где:

* u(t) — вектор состояния системы в момент времени t,
* A — матрица, описывающая систему,
* b — вектор внешних сил или источников.

Метод неявного Эйлера обновляет решение на каждом шаге по формуле:

u(t+Δt)=u(t)+Δt⋅(Au(t+Δt)+b(t+Δt))

Переписываем это уравнение, чтобы выразить решение для следующего шага:

где:

* I — единичная матрица,
* Δt — шаг по времени.

Получается система линейных уравнений, которую нужно решить для вектора u(t+Δt).

## Тридиагональная матрица

Если матрица A является тридиагональной, это означает, что все элементы матрицы вне главной диагонали, а также двух соседних диагоналей (над и под главной), равны нулю. Тридиагональная матрица имеет вид:

Система линейных уравнений для метода неявного Эйлера становится:

(I−ΔtA)u(t+Δt)=u(t)+Δtb(t+Δt)

Это линейная система с тридиагональной матрицей, которую можно решить с использованием метода Томаса.

## Метод Томаса

Метод Томаса — это специализированная форма метода Гаусса, оптимизированная для тридиагональных матриц. Система линейных уравнений для тридиагональной матрицы A имеет вид:

Ax=d

где A — тридиагональная матрица, x — вектор неизвестных, и d — вектор правых частей.

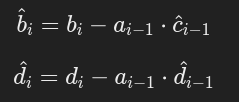
Система уравнений:

...

### Шаги метода Томаса

#### 1. Прямой ход (Forward Elimination)

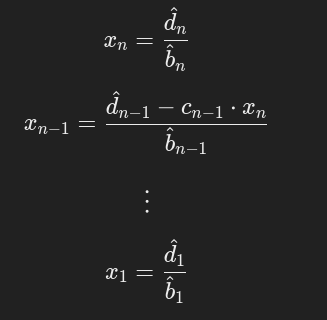
Необходимо модифицировать матрицу системы, чтобы избавиться от поддиагональных элементов:

​

где — это модифицированные элементы главной диагонали, а — обновленные элементы вектора правых частей.

#### 2. Обратный ход (Back Substitution)

После того как система станет верхней треугольной, мы можем решить её методом обратного хода:



## Метод Томаса для неявного метода Эйлера

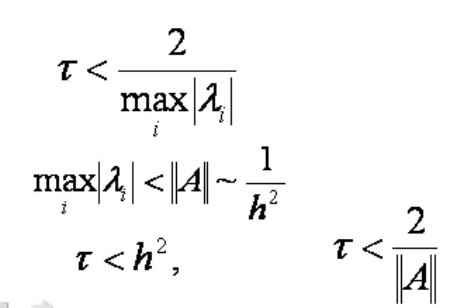
Когда матрица A тридиагональная, для решения системы линейных уравнений для метода неявного Эйлера, мы используем алгоритм Томаса. Шаги следующие:

1. Формируем тридиагональную матрицу .
2. Решаем систему с использованием метода Томаса.

# Жёсткость системы

При решении явным методом нужно учитывать, что система может быть жёсткой. В этом случае накладывается ограничение на шаг интегрирования 𝛕.

Это ограничение зависит от обусловленности матрицы, которое зависит от шага разбиение интервала по *r.*

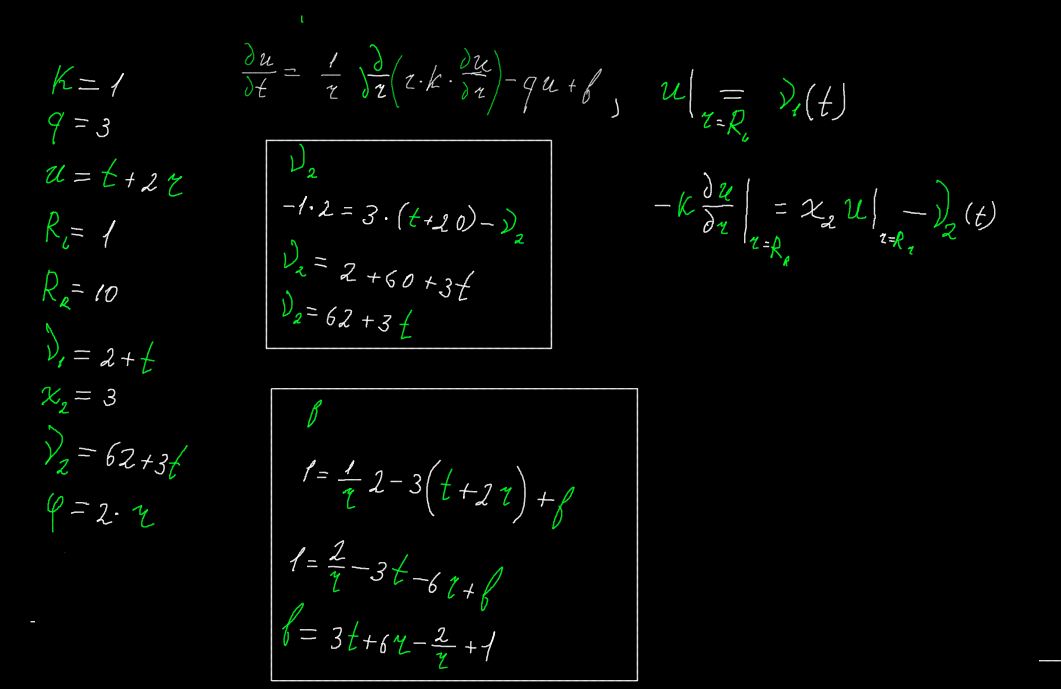


При превышении этого ограничения погрешность станет накапливаться с очень большой скоростью (решение неустойчиво).

# Тестирование

## Пример 1

Первый тест будет выглядеть как на рисунке. Предварительно ожидается что компьютер без проблем справится с поставленной задачей. Зададим .



В результате программы получаем следующие столбцы:

**Method Name**  Название метода

**R Steps Count** Количество разбиений по R

**R Step Size** Получившийся шаг по R

**T Steps Count** Количество разбиений по T

**T Step Size**  Получившийся шаг по T

**Sum Inaccuracy** Суммарная ошибка

**Max Inaccuracy** Максимальная ошибка

**Sum Inaccuracy (2nd Column)** Суммарная погрешность второй колонки

**Max Inaccuracy (2nd Column)** Максимальная погрешность второй колонки

**Sum Inaccuracy (Middle Col)** Суммарная погрешность средней колонки

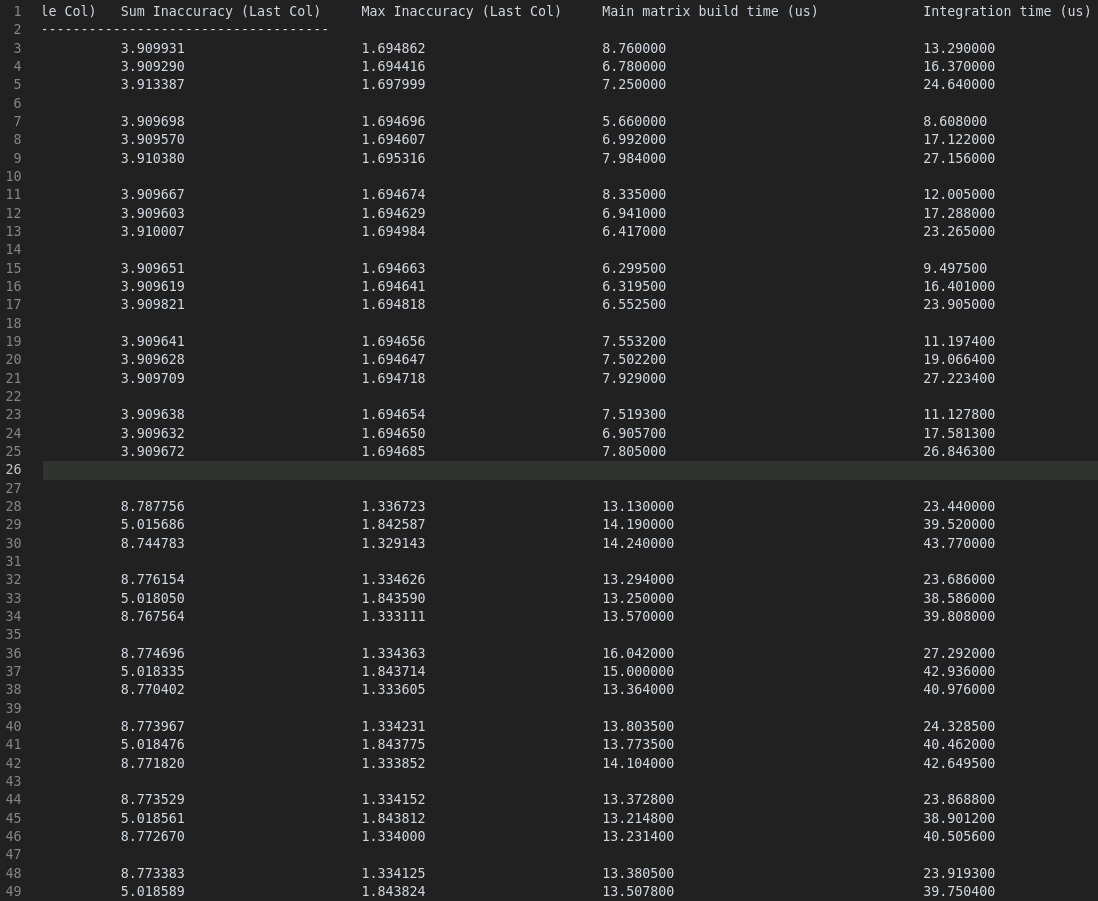
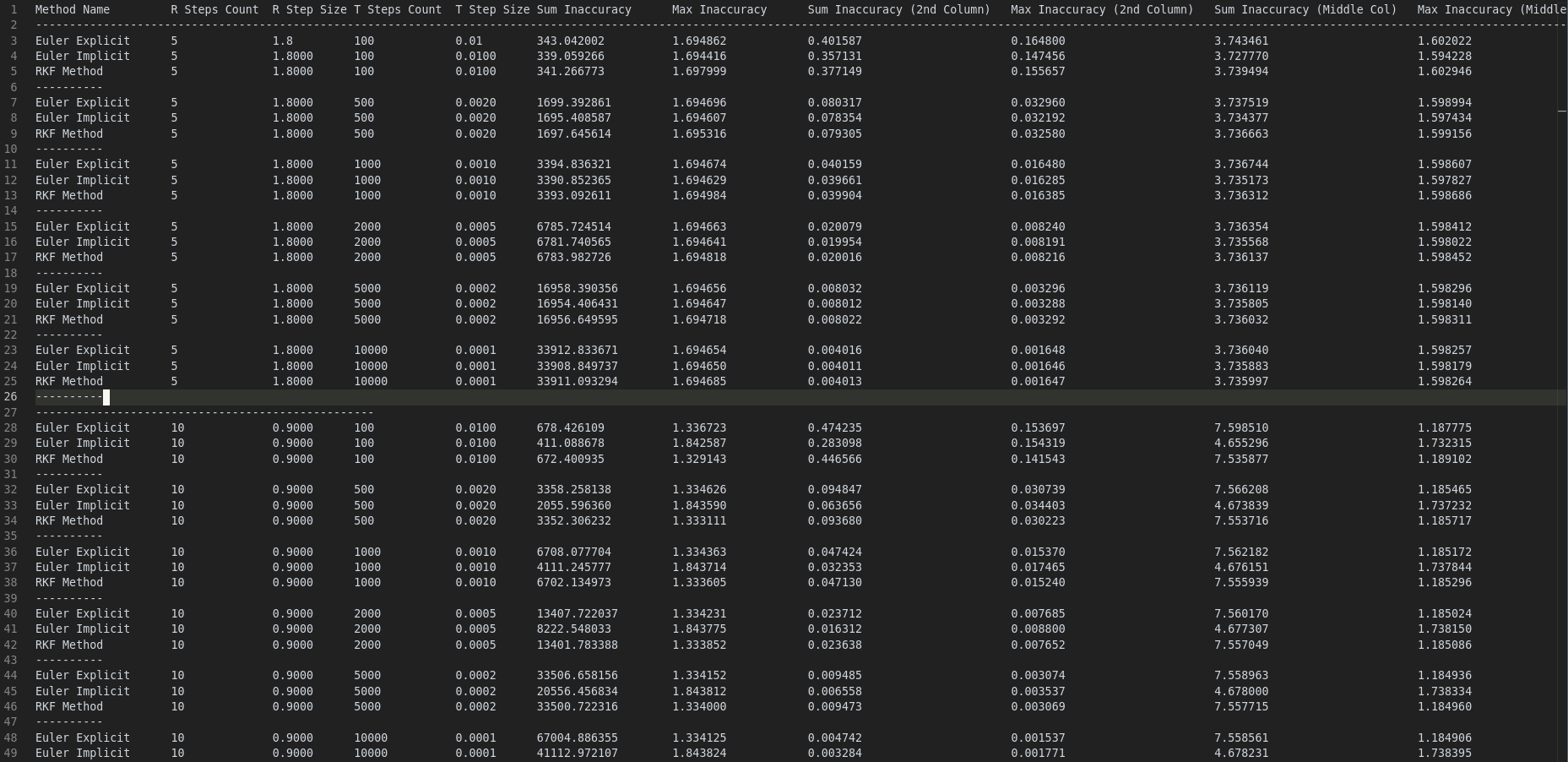
**Max Inaccuracy (Middle Col)** Максимальная погрешность средней колонки

**Sum Inaccuracy (Last Col)** Суммарная погрешность последней колонки

**Max Inaccuracy (Last Col)** Максимальная погрешность последней колонки

**Main matrix build time (us)** Среднее время построения матрицы

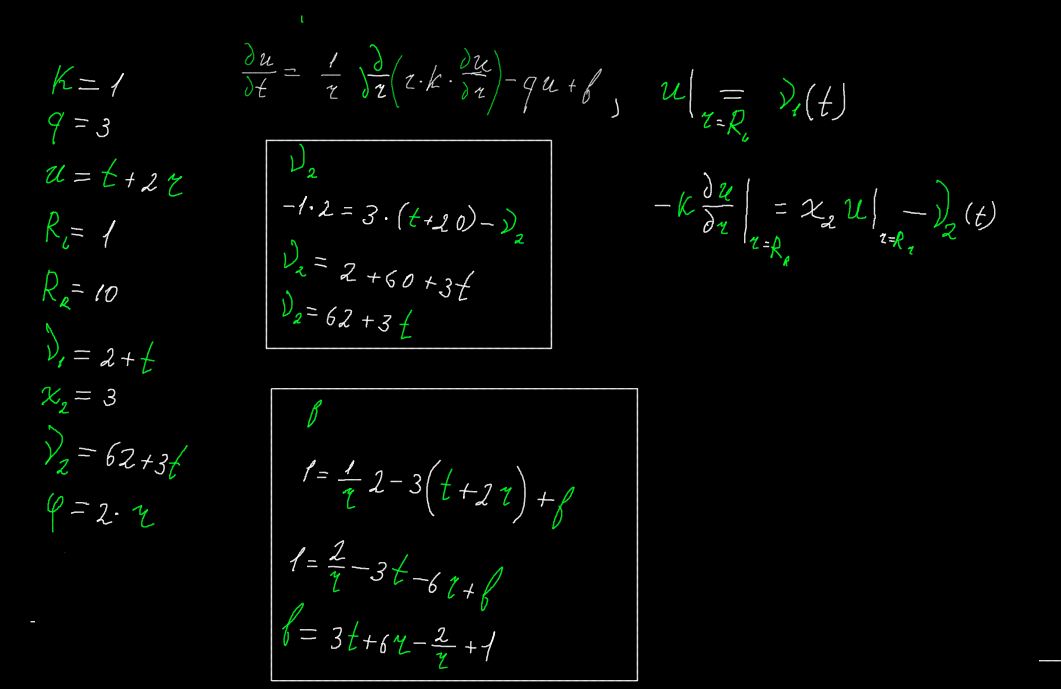
**Integration time (us)** Среднее время интегрирования ОДНОГО ШАГА



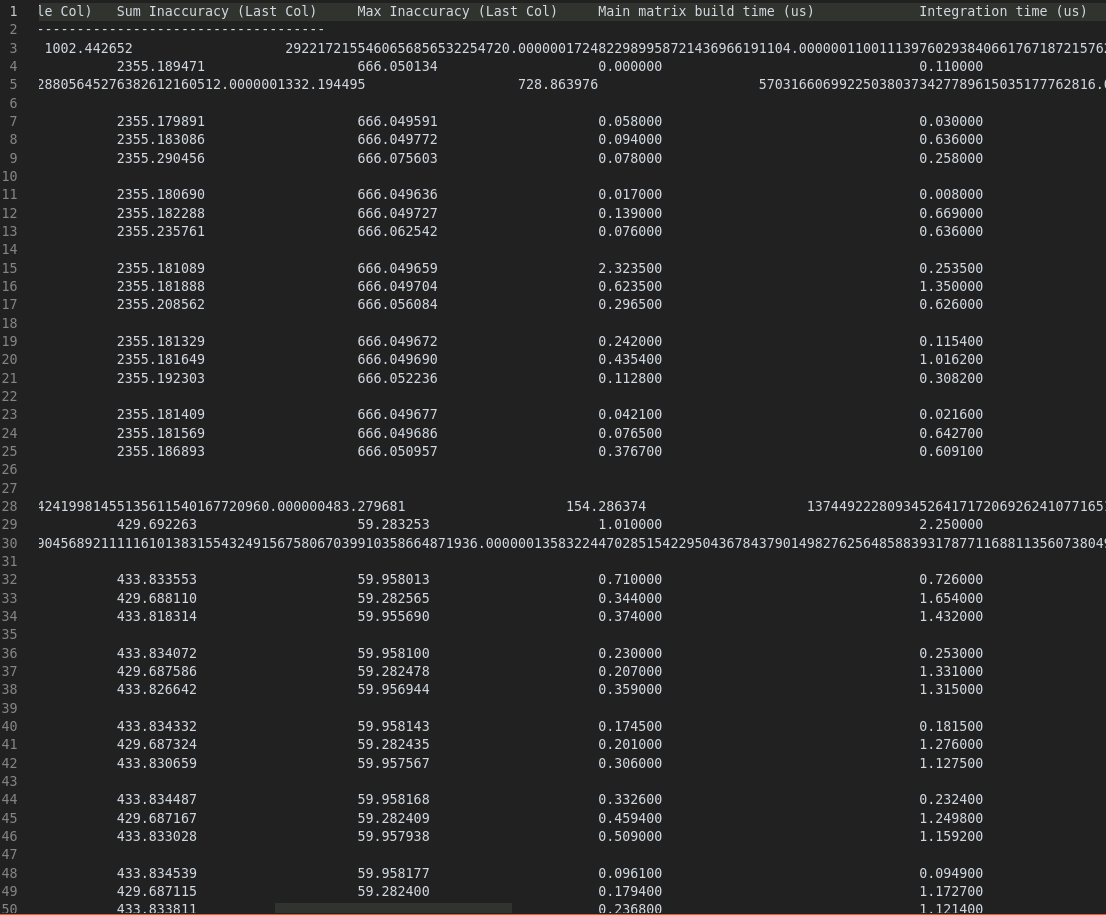
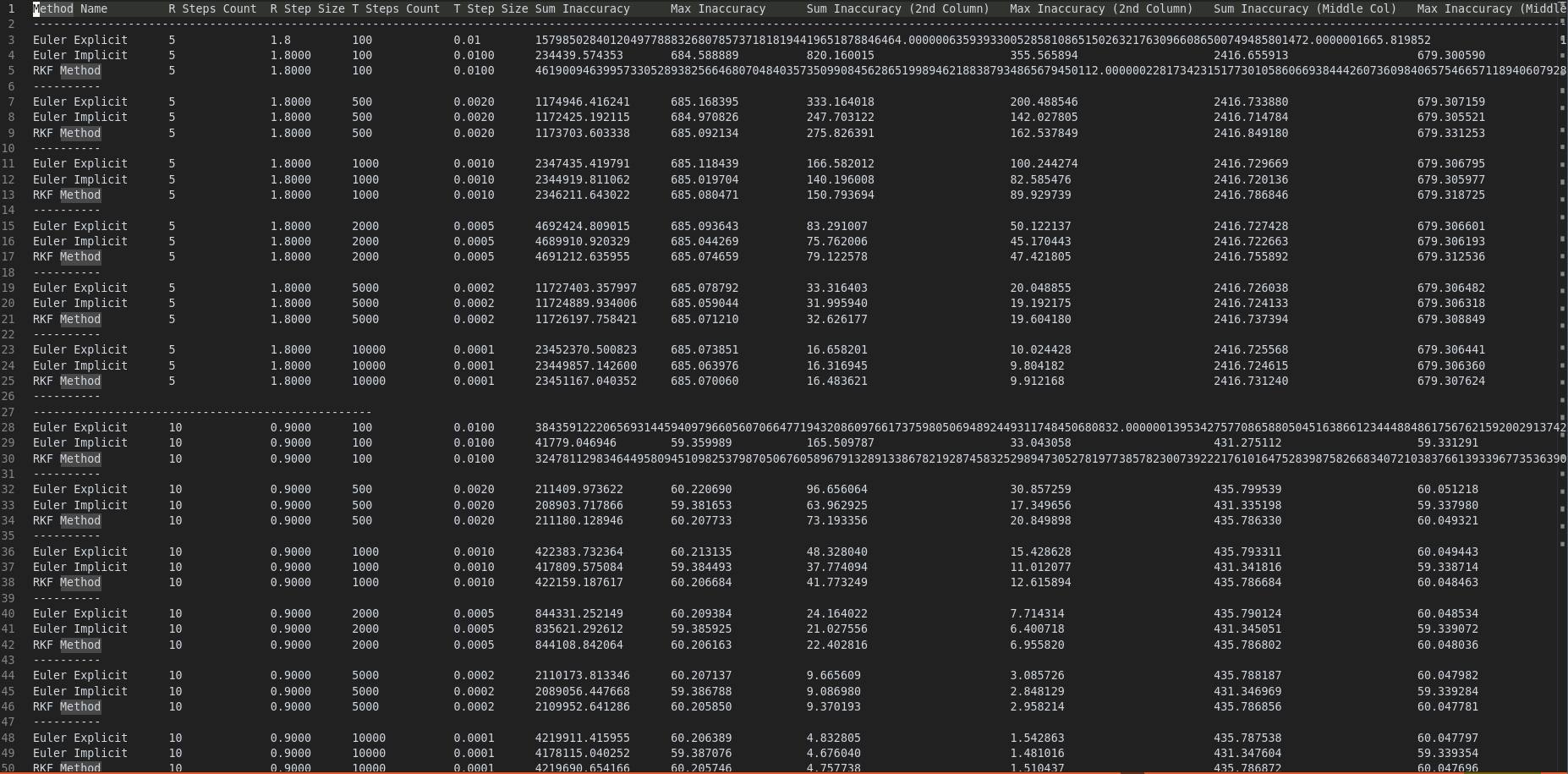
Глядя на результаты можно сделать вывод, что метод Рунге-Кутты показал наименьшую погрешность, однако надо сказать, что Явный метод Эйлера отличался от Рунге-Кутты максимум на 0.001-0.003. Неявный метод Эйлера выдал несколько большую погрешность, примерно в 1.2 раза большую чем предыдущие два. Это всё равно неплохой результат, поскольку он позволяет считать с очень большой скоростью.

Целиком с таблцией можно ознакомится по ссылке: https://github.com/Hryapusek/rust-tridiagonal-matrix-vector/blob/cp3-try-correct-version/lab2/result.txt

## Пример 2



Второй тест выглядит несколько серьезнее первого, и забегая наперед скажу, что программа не смогла уложиться в разумные значения погрешностей.



Лучшее чего удалось добиться — это уменьшение погрешности до значения 1.5 с помощью неявного метода Эйлера при разбиении **R** интервала на 10 частей и интервала **Т** на 10000 частей.

В целом такая точность может быть допустимой если нужны лишь приблизительные вычисления.

Целиком с таблцией можно ознакомится по ссылке: https://github.com/Hryapusek/rust-tridiagonal-matrix-vector/blob/cp3-try-correct-version/lab2/result2.txt

# Вывод

Освоили необычный способ численного решения дифференциального уравнения второго порядка с различными граничными условиями, увидели зависимости результата от различных комбинаций настраиваемых параметров и освоили неявный метод Эйлера в комбинации с трёхдиагональным методом.

Исходный код доступен по ссылке: https://github.com/Hryapusek/rust-tridiagonal-matrix-vector/tree/cp3-try-correct-version

# Код

#pragma once

#include <interface/i\_euler\_explicit\_method.hpp>

#include <Eigen/Dense>

class DefaultEulerExplicitMethod : public IEulerExplicitMethod

{

public:

auto integrate(

Eigen::VectorX<Number\_t> const& start\_v,

Eigen::MatrixX<Number\_t> const& A,

Eigen::SparseVector<Number\_t> const& g,

std::vector<Number\_t> const& points

) -> Eigen::SparseMatrix<Number\_t> override;

auto name() -> std::string override { return "Euler Explicit"; }

};

#pragma once

#include <interface/i\_euler\_implicit\_method.hpp>

#include <Eigen/Dense>

class DefaultEulerImplicitMethod : public IEulerImplicitMethod

{

public:

auto integrate(

Eigen::VectorX<Number\_t> const& start\_v,

Eigen::MatrixX<Number\_t> const& A,

Eigen::SparseVector<Number\_t> const& g,

std::vector<Number\_t> const& points

) -> Eigen::SparseMatrix<Number\_t> override;

auto name() -> std::string override { return "Euler Implicit"; }

};

#pragma once

#include <memory>

#include <Eigen/Dense>

#include <interface/i\_main\_matrix\_calculator.hpp>

#include <input\_parameters.hpp>

class DefaultMainMatrixCalculator : public IMainMatrixCalculator

{

public:

explicit DefaultMainMatrixCalculator(

std::shared\_ptr<InputParameters> params,

std::vector<Number\_t> r\_points

)

: params\_(params)

, r\_points\_(std::move(r\_points))

{}

auto calc\_a(size\_t r\_index, Number\_t t) const -> Number\_t override;

auto calc\_b(size\_t r\_index, Number\_t t) const -> Number\_t override;

auto calc\_c(size\_t r\_index, Number\_t t) const -> Number\_t override;

auto calc\_g(size\_t r\_index, Number\_t t) const -> Number\_t override;

auto r\_points() const -> std::vector<Number\_t> const& { return r\_points\_; }

auto params() const -> std::shared\_ptr<InputParameters> const& { return params\_; }

protected:

std::vector<Number\_t> r\_points\_;

std::shared\_ptr<InputParameters> params\_;

};

#pragma once

#include <interface/i\_base\_integrate.hpp>

#include <Eigen/Dense>

class RKFMethod : public IBaseIntegrate

{

public:

auto integrate(

Eigen::VectorX<Number\_t> const& start\_v,

Eigen::MatrixX<Number\_t> const& A,

Eigen::SparseVector<Number\_t> const& g,

std::vector<Number\_t> const& points

) -> Eigen::SparseMatrix<Number\_t> override;

auto name() -> std::string override { return "RKF Method"; }

};

#pragma once

#include <string>

#include <defines.hpp>

#include <Eigen/Sparse>

class IBaseIntegrate

{

public:

virtual auto integrate(

Eigen::VectorX<Number\_t> const& start\_v,

Eigen::MatrixX<Number\_t> const& A,

Eigen::SparseVector<Number\_t> const& g,

std::vector<Number\_t> const& points

) -> Eigen::SparseMatrix<Number\_t> = 0;

virtual auto name() -> std::string { return "Unknown"; }

};

#pragma once

#include <interface/i\_base\_integrate.hpp>

class IEulerExplicitMethod : public IBaseIntegrate

{};

#pragma once

#include <interface/i\_base\_integrate.hpp>

class IEulerImplicitMethod : public IBaseIntegrate

{};

#pragma once

#include <cstdio>

using Number\_t = double;

class IMainMatrixCalculator

{

public:

using Number\_t = double;

virtual auto calc\_a(size\_t r\_index, Number\_t t) const -> Number\_t = 0;

virtual auto calc\_b(size\_t r\_index, Number\_t t) const -> Number\_t = 0;

virtual auto calc\_c(size\_t r\_index, Number\_t t) const -> Number\_t = 0;

virtual auto calc\_g(size\_t r\_index, Number\_t t) const -> Number\_t = 0;

};

#pragma once

#include <functional>

using Number\_t = double;

// First argument is r, second is t

using R\_T\_Function\_type = std::function<double(double, double)>;

using T\_Function\_type = std::function<double(double)>;

using R\_Function\_type = std::function<double(double)>;

#pragma once

#include <defines.hpp>

struct InputParameters {

Number\_t Rl;

Number\_t Rr; // [Rl, Rr]

Number\_t T; // [0, T]

// First type condition

T\_Function\_type v1; // u(rL) = v1(t)

// Third type condition

Number\_t hi2; // -k \* du/dr = hi2 \* u(rR) - v2(t)

R\_Function\_type phi;

T\_Function\_type v2;

// Just input functions

R\_T\_Function\_type k;

R\_T\_Function\_type q;

R\_T\_Function\_type f;

};

#include <vector>

#include <cstdio>

#include <contract/contract.hpp>

#include <defines.hpp>

auto split\_interval(const Number\_t& left, const Number\_t& right, size\_t num\_intervals) -> std::vector<Number\_t>;

// Calculate the length of an interval `index-1` to `index`

auto calc\_h(const std::vector<Number\_t>& intervals, size\_t index) -> Number\_t;

// Calculate the cross h of an interval

auto calc\_cross\_h(const std::vector<Number\_t>& intervals, size\_t index) -> Number\_t;

/// @return middle point between `index` and `index - 1`

auto middle\_point(const std::vector<Number\_t>& intervals, size\_t index) -> Number\_t;

#include <default\_impl/euler\_explicit\_method.hpp>

#include <Eigen/Dense>

#include <contract/contract.hpp>

auto DefaultEulerExplicitMethod::integrate(

Eigen::VectorX<Number\_t> const& start\_v,

Eigen::MatrixX<Number\_t> const& A,

Eigen::SparseVector<Number\_t> const& g,

std::vector<Number\_t> const& points

) -> Eigen::SparseMatrix<Number\_t>

{

auto result = Eigen::MatrixX<Number\_t>(

A.rows(),

points.size()

); // Adjust columns based on intervals size

contract(fun)

{

precondition(A.rows() == A.cols(), "A must be a square matrix");

precondition(A.rows() == start\_v.rows(), "A and start\_v must have the same number of rows");

precondition(A.rows() == g.rows(), "A and g must have the same number of rows");

postcondition(

result.cols() == points.size(),

"result must have the same number of columns as intervals"

);

};

result.col(0) = start\_v.sparseView();

auto E = Eigen::SparseMatrix<Number\_t>(A.rows(), A.rows());

E.setIdentity();

auto H = points.at(1) - points.at(0); // Step size

for(size\_t i = 1; i < 2; ++i) { // Loop over intervals, not A.cols()

// for(size\_t i = 1; i < points.size(); ++i) {

result.col(i) = (E + H \* A) \* result.col(i - 1) + H \* g; // Euler update

}

return result.sparseView();

}

#include <default\_impl/euler\_implicit\_method.hpp>

#include <vector>

#include <stdexcept>

#include <contract/contract.hpp>

#include <Eigen/Dense>

#include <Eigen/Sparse>

// Helper function to solve a tridiagonal system using the Thomas algorithm

void solve\_tridiagonal(

const Eigen::VectorXd& a, // Lower diagonal (size n-1)

const Eigen::VectorXd& b, // Main diagonal (size n)

const Eigen::VectorXd& c, // Upper diagonal (size n-1)

const Eigen::VectorXd& d, // Right-hand side (size n)

Eigen::VectorXd& x // Solution (size n)

)

{

size\_t n = b.size();

Eigen::VectorXd cp(n); // Temporary vector for forward substitution

Eigen::VectorXd dp(n); // Temporary vector for the right-hand side adjustment

// Forward elimination

cp(0) = c(0) / b(0);

dp(0) = d(0) / b(0);

for (size\_t i = 1; i < n - 1; ++i) { // Loop from 1 to n-2 for the main part

double m = 1.0 / (b(i) - a(i - 1) \* cp(i - 1)); // a(i-1) for lower diagonal

cp(i) = c(i) \* m;

dp(i) = (d(i) - a(i - 1) \* dp(i - 1)) \* m;

}

// Last step for the forward elimination

dp(n - 1) = (d(n - 1) - a(n - 2) \* dp(n - 2)) / (b(n - 1) - a(n - 2) \* cp(n - 2));

// Backward substitution

x(n - 1) = dp(n - 1);

for (size\_t i = n - 2; i != static\_cast<size\_t>(-1); --i) {

x(i) = dp(i) - cp(i) \* x(i + 1);

}

}

auto DefaultEulerImplicitMethod::integrate(

Eigen::VectorX<Number\_t> const& start\_v,

Eigen::MatrixX<Number\_t> const& A,

Eigen::SparseVector<Number\_t> const& g,

std::vector<Number\_t> const& points

) -> Eigen::SparseMatrix<Number\_t>

{

Eigen::MatrixX<Number\_t> result(

A.rows(), points.size()); // Adjust the columns to match intervals

contract(fun)

{

precondition(A.rows() == A.cols(), "A must be a square matrix");

precondition(A.rows() == start\_v.rows(), "A and start\_v must have the same number of rows");

precondition(A.rows() == g.rows(), "A and g must have the same number of rows");

postcondition(

result.cols() == points.size(),

"result must have the same number of columns as intervals"

);

};

result.col(0) = start\_v.sparseView();

auto E = Eigen::SparseMatrix<Number\_t>(A.rows(), A.rows());

E.setIdentity();

for (size\_t i = 1; i < points.size(); ++i) { // Iterate over intervals

auto H = points.at(i) - points.at(i - 1); // Step size

Eigen::SparseMatrix<Number\_t> M = E - H \* A; // Matrix for the implicit step

// Prepare the right-hand side of the equation

Eigen::SparseVector<Number\_t> rhs = result.col(i - 1) + H \* g;

// We will now extract the tridiagonal parts of M to solve the system

size\_t n = M.rows();

// Extract diagonals from sparse matrix M

Eigen::VectorXd a(n - 1); // Lower diagonal

Eigen::VectorXd b(n); // Main diagonal

Eigen::VectorXd c(n - 1); // Upper diagonal

// Fill diagonals by iterating through the sparse matrix

for (int k = 0; k < n; ++k) {

b(k) = M.coeff(k, k); // Main diagonal

if (k > 0) {

a(k - 1) = M.coeff(k, k - 1); // Lower diagonal

}

if (k < n - 1) {

c(k) = M.coeff(k, k + 1); // Upper diagonal

}

}

// Prepare the right-hand side (rhs) vector

Eigen::VectorXd rhs\_full(n);

for (size\_t k = 0; k < n; ++k) {

rhs\_full(k) = rhs.coeff(k); // Copy rhs values

}

// Now solve the tridiagonal system

Eigen::VectorXd solution(n);

solve\_tridiagonal(a, b, c, rhs\_full, solution);

// Update the result for this timestep

result.col(i) = solution.sparseView();

}

return result.sparseView();

}

#include <default\_impl/main\_matrix\_calculator.hpp>

#include <cassert>

#include <contract/contract.hpp>

#include <interval\_splitter.hpp>

auto DefaultMainMatrixCalculator::calc\_a(size\_t r\_index, Number\_t t) const -> Number\_t

{

contract(fun)

{

precondition(

r\_index != 0,

"You should not calculate anything for index == 0 - you already have v function"

);

precondition(r\_index != 1, "You should not calculate a for index == 1");

precondition(r\_index < r\_points\_.size(), "index out of range");

};

auto mid\_point = middle\_point(r\_points\_, r\_index);

auto k\_value = params\_->k(mid\_point, t);

auto h\_value = calc\_h(r\_points\_, r\_index);

return mid\_point \* k\_value / h\_value;

}

auto DefaultMainMatrixCalculator::calc\_b(size\_t r\_index, Number\_t t) const -> Number\_t

{

contract(fun)

{

precondition(

r\_index != 0,

"You should not calculate anything for index == 0 - you already have v function"

);

precondition(

r\_index != r\_points\_.size() - 1,

"You should not calculate b for index == intervals\_.size() - 1"

);

precondition(r\_index < r\_points\_.size(), "index out of range");

};

auto up = middle\_point(r\_points\_, r\_index + 1)

\* params\_->k(middle\_point(r\_points\_, r\_index + 1), t);

auto down = calc\_h(r\_points\_, r\_index + 1);

return up / down;

}

/\* clang-format off \*/

auto DefaultMainMatrixCalculator::calc\_c(size\_t r\_index, Number\_t t) const -> Number\_t

{

contract(fun) {

precondition(r\_index != 0, "You should not calculate anything for index == 0 - you already have v function");

precondition(r\_index < r\_points\_.size(), "index out of range");

};

if (r\_index == r\_points\_.size() - 1) {

return - middle\_point(r\_points\_, r\_index) \* params\_->k(middle\_point(r\_points\_, r\_index), t)

/ calc\_h(r\_points\_, r\_index)

- params\_->hi2

- params\_->q(r\_points\_[r\_index], t)

\* calc\_cross\_h(r\_points\_, r\_index);

} else {

auto m\_point\_index = middle\_point(r\_points\_, r\_index);

auto m\_point\_index\_plus\_1 = middle\_point(r\_points\_, r\_index + 1);

auto k\_index = params\_->k(m\_point\_index, t);

auto k\_index\_plus\_1 = params\_->k(m\_point\_index\_plus\_1, t);

auto h\_index\_plus\_1 = calc\_h(r\_points\_, r\_index + 1);

auto cross\_h\_index = calc\_cross\_h(r\_points\_, r\_index);

auto r = r\_points\_[r\_index];

auto q\_index = params\_->q(r\_index, t);

return -m\_point\_index \* k\_index / h\_index\_plus\_1

-m\_point\_index\_plus\_1 \* k\_index\_plus\_1 / h\_index\_plus\_1

-q\_index \* cross\_h\_index;

}

assert(false);

}

/\* clang-format on \*/

/\* clang-format off \*/

auto DefaultMainMatrixCalculator::calc\_g(size\_t r\_index, Number\_t t) const -> Number\_t {

contract(fun) {

precondition(r\_index != 0, "You should not calculate anything for index == 0 - you already have v function");

};

if (r\_index == 1) {

return params\_->f(r\_points\_[r\_index], t)

\* calc\_cross\_h(r\_points\_, r\_index-1)

+

middle\_point(r\_points\_, r\_index)

\* params\_->k(middle\_point(r\_points\_, r\_index), t)

/ calc\_h(r\_points\_, r\_index)

\* params\_->v1(t);

} else if (r\_index == r\_points\_.size() - 1) {

return params\_->f(r\_points\_[r\_index], t) \* calc\_cross\_h(r\_points\_, r\_index)

+ params\_->v2(t);

} else {

return params\_->f(r\_points\_[r\_index], t) \* calc\_cross\_h(r\_points\_, r\_index);

}

assert(false);

}

/\* clang-format on \*/

#include <default\_impl/rkf\_method.hpp>

#include <Eigen/Sparse>

auto RKFMethod::integrate(

Eigen::VectorX<Number\_t> const& start\_v,

Eigen::MatrixX<Number\_t> const& A,

Eigen::SparseVector<Number\_t> const& g,

std::vector<Number\_t> const& points

) -> Eigen::SparseMatrix<Number\_t>

{

auto result = Eigen::MatrixX<Number\_t>(A.rows(), A.cols());

result.col(0) = start\_v.sparseView();

for(size\_t i = 1; i < points.size(); ++i) {

auto H = points.at(i) - points.at(i - 1);

Eigen::MatrixX<Number\_t> u = result.col(i - 1);

Eigen::MatrixX<Number\_t> k1 = (H \* (A \* u + g));

Eigen::MatrixX<Number\_t> k2 = (H \* (A \* (u + 0.5 \* k1) + g));

Eigen::MatrixX<Number\_t> k3 = (H \* (A \* (u + 0.5 \* k2) + g));

Eigen::MatrixX<Number\_t> k4 = (H \* (A \* (u + k3) + g));

result.col(i) = u + (k1 + 2 \* k2 + 2 \* k3 + k4) / 6;

}

return result.sparseView();

}

#include <interval\_splitter.hpp>

auto split\_interval(Number\_t const& left, Number\_t const& right, size\_t num\_intervals) -> std::vector<Number\_t>

{

std::vector<Number\_t> intervals;

contract(fun)

{

precondition(num\_intervals > 0, "invalid number of intervals");

};

auto interval\_size = (right - left) / num\_intervals;

for(size\_t i = 0; i < num\_intervals; ++i) {

intervals.push\_back(left + interval\_size \* i);

}

intervals.push\_back(right);

return intervals;

}

// Calculate the length of an interval `index-1` to `index`

auto calc\_h(std::vector<Number\_t> const& points, size\_t index) -> Number\_t

{

contract(fun)

{

precondition(index < points.size(), "index out of range");

precondition(index > 0, "h can not be calculated for the first interval");

};

return (points.at(index) - points.at(index - 1));

}

// Calculate the cross h of an interval

auto calc\_cross\_h(std::vector<Number\_t> const& points, size\_t index) -> Number\_t

{

if(index == 0) {

return calc\_h(points, 1) / 2;

}

else if(index == points.size() - 1) {

return calc\_h(points, index) / 2;

}

else {

return (calc\_h(points, index) + calc\_h(points, index + 1)) / 2;

}

}

/// @return middle point between `index` and `index - 1`

auto middle\_point(std::vector<Number\_t> const& points, size\_t index) -> Number\_t

{

return (points.at(index) + points.at(index - 1)) / 2;

}

#include <iostream>

#include <iomanip> // For std::setw, std::fixed, std::setprecision, etc.

#include <chrono>

#include <Eigen/Dense>

#include <Eigen/Sparse>

#include <defines.hpp>

#include <input\_parameters.hpp>

#include <default\_impl/main\_matrix\_calculator.hpp>

#include <interval\_splitter.hpp>

#include <default\_impl/euler\_explicit\_method.hpp>

#include <default\_impl/euler\_implicit\_method.hpp>

#include <default\_impl/rkf\_method.hpp>

auto build\_main\_matrix(

DefaultMainMatrixCalculator const& calc,

double t,

Eigen::MatrixX<Number\_t>& main\_matrix

) -> Eigen::MatrixX<Number\_t>

{

auto const size = calc.r\_points().size() - 1;

contract(fun) { precondition(main\_matrix.rows() == size and main\_matrix.cols() == size); };

for(size\_t row = 0; row < size; ++row) {

main\_matrix(row, row) = calc.calc\_c(row + 1, t);

if(row + 1 < size) {

main\_matrix(row, row + 1) = calc.calc\_b(row + 1, t);

}

if(row > 0) {

main\_matrix(row, row - 1) = calc.calc\_a(row + 1, t);

}

}

return main\_matrix;

}

auto build\_g\_vector(DefaultMainMatrixCalculator const& calc, Number\_t t)

-> Eigen::SparseVector<Number\_t>

{

auto g = Eigen::SparseVector<Number\_t>(calc.r\_points().size() - 1);

for(size\_t row = 0; row < calc.r\_points().size() - 1; ++row) {

g.insert(row) = calc.calc\_g(row + 1, t);

}

return g;

}

auto integrate(

DefaultMainMatrixCalculator const& calc,

std::vector<Number\_t> const& r\_points,

std::vector<Number\_t> const& t\_points,

Eigen::VectorX<Number\_t> const& start\_v,

IBaseIntegrate& method

) -> std::tuple<Eigen::MatrixX<Number\_t>, double, double>

{

auto result = Eigen::MatrixXd(Eigen::MatrixXd::Zero(r\_points.size() - 1, t\_points.size()));

result.col(0) = start\_v;

std::cout << std::setprecision(4);

Eigen::MatrixX<Number\_t> main\_matrix(calc.r\_points().size() - 1, calc.r\_points().size() - 1);

Eigen::VectorX<Number\_t> g;

double total\_main\_matrix\_time = 0;

double total\_integrate\_time = 0;

for(size\_t i = 1; i < t\_points.size(); ++i) {

std::cerr << "Percentage done: " << (double)i / t\_points.size() \* 100 << " ";

std::cerr.flush();

auto start = std::chrono::high\_resolution\_clock::now();

if(dynamic\_cast<IEulerImplicitMethod\*>(&method)) {

build\_main\_matrix(calc, t\_points.at(i), main\_matrix);

g = build\_g\_vector(calc, t\_points.at(i));

}

else {

build\_main\_matrix(calc, t\_points.at(i - 1), main\_matrix);

g = build\_g\_vector(calc, t\_points.at(i - 1));

}

auto main\_matrix\_time = std::chrono::duration\_cast<std::chrono::microseconds>(

std::chrono::high\_resolution\_clock::now() - start

)

.count();

total\_main\_matrix\_time += main\_matrix\_time;

start = std::chrono::high\_resolution\_clock::now();

auto points = std::vector<Number\_t>(t\_points.cbegin() + i - 1, t\_points.cbegin() + i + 1);

auto integrated\_result =

method.integrate(result.col(i - 1), main\_matrix.sparseView(), g.sparseView(), points);

auto integrate\_time = std::chrono::duration\_cast<std::chrono::microseconds>(

std::chrono::high\_resolution\_clock::now() - start

)

.count();

total\_integrate\_time += integrate\_time;

result.col(i) = integrated\_result.col(1);

std::cerr << "Time for main matrix: " << main\_matrix\_time

<< "us, Time for integration: " << integrate\_time << "us\r";

std::cerr.flush();

}

std::cerr << "\n";

return std::make\_tuple(

result,

total\_main\_matrix\_time / (t\_points.size() - 1),

total\_integrate\_time / (t\_points.size() - 1)

);

}

void print\_table\_header()

{

// Set column widths for consistent formatting

std::cout << std::setw(20) << std::left << "Method Name" << std::setw(15) << "R Steps Count"

<< std::setw(12) << "R Step Size" << std::setw(15) << "T Steps Count" << std::setw(12)

<< "T Step Size" << std::setw(20) << "Sum Inaccuracy" << std::setw(20)

<< "Max Inaccuracy" << std::setw(30) << "Sum Inaccuracy (2nd Column)" << std::setw(30)

<< "Max Inaccuracy (2nd Column)" << std::setw(30) << "Sum Inaccuracy (Middle Col)"

<< std::setw(30) << "Max Inaccuracy (Middle Col)" << std::setw(30)

<< "Sum Inaccuracy (Last Col)" << std::setw(30) << "Max Inaccuracy (Last Col)"

<< std::setw(40) << "Main matrix build time (us)" << std::setw(40)

<< "Integration time (us)" << std::endl;

// Print a separator line

std::cout << std::string(260, '-') << std::endl;

}

void print\_row(

R\_T\_Function\_type original\_func,

std::vector<Number\_t> const& r\_points,

std::vector<Number\_t> const& t\_points,

Eigen::VectorX<Number\_t> const& start\_v,

Eigen::MatrixXd const& result,

IBaseIntegrate& method,

double build\_main\_matrix\_time,

double integration\_time

)

{

size\_t t\_steps\_count = t\_points.size() - 1;

size\_t r\_steps\_count = r\_points.size() - 1;

Number\_t t\_step\_size = (t\_points.back() - t\_points.front()) / t\_steps\_count;

Number\_t r\_step\_size = (r\_points.back() - r\_points.front()) / r\_steps\_count;

// Initialize inaccuracies for columns and total

Number\_t sum\_inaccuracy\_total = 0, max\_inaccuracy\_total = 0;

Number\_t sum\_inaccuracy\_2nd\_col = 0, max\_inaccuracy\_2nd\_col = 0;

Number\_t sum\_inaccuracy\_middle\_col = 0, max\_inaccuracy\_middle\_col = 0;

Number\_t sum\_inaccuracy\_last\_col = 0, max\_inaccuracy\_last\_col = 0;

Number\_t sum\_inaccuracy\_each\_cell = 0;

// Iterate over all cells of the result matrix

for(size\_t i = 1; i < r\_points.size(); ++i) { // Skip the first row (r = 0)

for(size\_t j = 0; j < t\_points.size(); ++j) {

Number\_t r = r\_points[i];

Number\_t t = t\_points[j];

Number\_t actual\_value = original\_func(r, t);

Number\_t result\_value = result(i - 1, j); // Note that we offset by 1 for r = 0

Number\_t inaccuracy = std::abs(actual\_value - result\_value);

sum\_inaccuracy\_each\_cell += inaccuracy;

// Sum and Max inaccuracies for the whole matrix

sum\_inaccuracy\_total += inaccuracy;

max\_inaccuracy\_total = std::max(max\_inaccuracy\_total, inaccuracy);

// Column-wise inaccuracies

if(j == 1) { // Second column

sum\_inaccuracy\_2nd\_col += inaccuracy;

max\_inaccuracy\_2nd\_col = std::max(max\_inaccuracy\_2nd\_col, inaccuracy);

}

if(j == t\_points.size() / 2) { // Middle column

sum\_inaccuracy\_middle\_col += inaccuracy;

max\_inaccuracy\_middle\_col = std::max(max\_inaccuracy\_middle\_col, inaccuracy);

}

if(j == t\_points.size() - 1) { // Last column

sum\_inaccuracy\_last\_col += inaccuracy;

max\_inaccuracy\_last\_col = std::max(max\_inaccuracy\_last\_col, inaccuracy);

}

}

}

// Print the row in the desired format

std::cout << std::setw(20) << std::left << method.name() // Method name

<< std::setw(15) << r\_steps\_count // R Steps Count

<< std::setw(12) << r\_step\_size // R Step Size

<< std::setw(15) << t\_steps\_count // T Steps Count

<< std::setw(12) << t\_step\_size // T Step Size

<< std::setw(20) << std::fixed << std::setprecision(6)

<< sum\_inaccuracy\_total // Sum Inaccuracy (Total)

<< std::setw(20) << max\_inaccuracy\_total // Max Inaccuracy (Total)

<< std::setw(30) << sum\_inaccuracy\_2nd\_col // Sum Inaccuracy (2nd Column)

<< std::setw(30) << max\_inaccuracy\_2nd\_col // Max Inaccuracy (2nd Column)

<< std::setw(30) << sum\_inaccuracy\_middle\_col // Sum Inaccuracy (Middle Column)

<< std::setw(30) << max\_inaccuracy\_middle\_col // Max Inaccuracy (Middle Column)

<< std::setw(30) << sum\_inaccuracy\_last\_col // Sum Inaccuracy (Last Column)

<< std::setw(30) << max\_inaccuracy\_last\_col // Max Inaccuracy (Last Column)

<< std::setw(40) << build\_main\_matrix\_time << std::setw(40) << integration\_time

<< std::endl;

}

void do\_all(std::shared\_ptr<InputParameters> params, R\_T\_Function\_type expected\_func)

{

print\_table\_header(); // Print header

static constexpr auto t\_interval\_counts = {100, 500, 1'000, 2'000, 5'000, 10'000};

static constexpr auto r\_interval\_counts = {5, 10, 20, 50, 100};

static std::vector<std::shared\_ptr<IBaseIntegrate>> methods =

{std::make\_shared<DefaultEulerExplicitMethod>(),

std::make\_shared<DefaultEulerImplicitMethod>(),

std::make\_shared<RKFMethod>()};

for(auto const r\_count : r\_interval\_counts) {

for(auto const t\_count : t\_interval\_counts) {

for(auto& method : methods) {

auto t\_points = split\_interval(0, params->T, t\_count);

auto r\_points = split\_interval(params->Rl, params->Rr, r\_count);

DefaultMainMatrixCalculator calc(params, r\_points);

Eigen::SparseVector<Number\_t> start\_v(r\_points.size() - 1);

for(auto i = 0; i < r\_points.size() - 1; ++i) {

start\_v.insert(i) = params->phi(r\_points.at(i + 1));

}

auto result = integrate(calc, r\_points, t\_points, start\_v, \*method);

print\_row(

expected\_func,

r\_points,

t\_points,

start\_v,

std::get<0>(result),

\*method,

std::get<1>(result),

std::get<2>(result)

);

std::cerr << "Finished for r: " << r\_count << " t: " << t\_count << "\n";

std::cerr.flush();

std::cout.flush();

}

std::cout << std::string(10, '-') << std::endl;

}

std::cout << std::string(50, '-') << std::endl;

}

}

void basic\_example()

{

std::shared\_ptr<InputParameters> params = std::make\_shared<InputParameters>();

params->Rl = 1;

params->Rr = 10;

params->T = 1;

params->v1 = [](double t) { return 2 + t; };

params->hi2 = 3;

params->phi = [](double r) { return 2 \* r; };

params->v2 = [](double t) { return 62 + 3 \* t; };

params->k = [](double r, double t) { return 1.0; };

params->q = [](double r, double t) { return 3.0; };

params->f = [](double r, double t) { return 3 \* t + 6 \* r + 1 - 2 / r; };

auto expected\_func = [](double r, double t) { return t + 2 \* r; };

do\_all(params, expected\_func);

}

void hard\_example()

{

std::shared\_ptr<InputParameters> params = std::make\_shared<InputParameters>();

params->Rl = 1;

params->Rr = 10;

params->T = 1;

params->v1 = [](double t) { return 5 + t \* t; };

params->hi2 = 5;

params->phi = [](double r) { return 5 \* r \* r; };

params->v2 = [](double t) { return 5\*(500 + t \* t) + (20 + t\*t)\*(100 + t \* t); };

params->k = [](double r, double t) { return 2 \* r + t \* t; };

params->q = [](double r, double t) { return 3 \* r \* r + t; };

params->f = [](double r, double t) {

return 2 \* t - 40 - 20 \* t \* t + 15 \* r \* r \* r \* r + 3 \* r \* r \* t \* t + 5 \* r \* r \* t

+ t \* t \* t;

};

auto expected\_func = [](double r, double t) { return 5 \* r \* r + t \* t; };

do\_all(params, expected\_func);

}

int main()

{

// basic\_example();

hard\_example();

return 0;

}

cmake\_minimum\_required(VERSION 3.16.3)

project(lab2 LANGUAGES CXX)

set(CMAKE\_CXX\_STANDARD 20)

add\_subdirectory(external/src/eigen)

add\_library(${PROJECT\_NAME} STATIC

external/src/contract/src/contract.cpp

src/default\_impl/euler\_explicit\_method.cc

src/default\_impl/euler\_implicit\_method.cc

src/default\_impl/main\_matrix\_calculator.cc

src/default\_impl/rkf\_method.cc

src/interval\_splitter.cc

)

target\_include\_directories(${PROJECT\_NAME}

PUBLIC

include/public

external/src/contract/include

)

target\_link\_libraries(${PROJECT\_NAME}

PUBLIC

Eigen3::Eigen

)

add\_executable(${PROJECT\_NAME}-main src/main.cc)

target\_link\_libraries(${PROJECT\_NAME}-main ${PROJECT\_NAME})

include(FetchContent)

FetchContent\_Declare(

googletest

GIT\_REPOSITORY https://github.com/google/googletest.git

GIT\_TAG v1.14.0

)

FetchContent\_MakeAvailable(googletest)

add\_executable(${PROJECT\_NAME}-test

tests/test.cc

tests/explicit-eugen-test.cc

tests/implicit-eugen-test.cc

)

target\_link\_libraries(${PROJECT\_NAME}-test

${PROJECT\_NAME}

gtest

)