|  |  |  |
| --- | --- | --- |
|  | | |
| **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)

[Жёсткость системы 7](#__RefHeading___Toc1385_811955146)

[Тестирование 7](#__RefHeading___Toc1387_811955146)

[Пример 1 7](#__RefHeading___Toc1389_811955146)

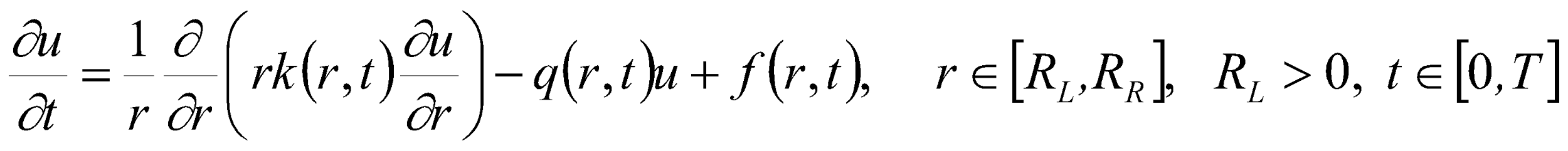
[Пример 2 8](#__RefHeading___Toc1391_811955146)

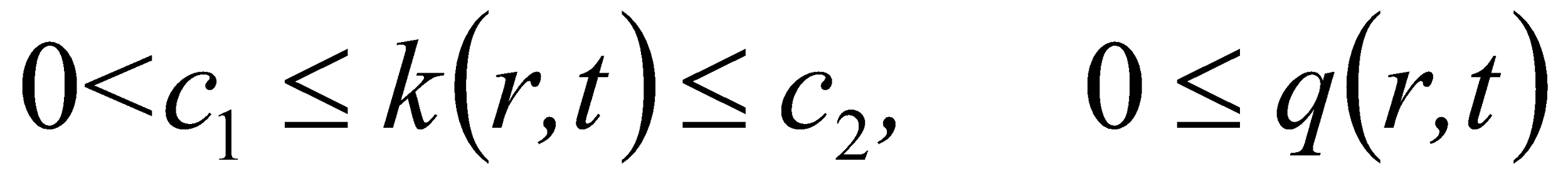
[Вывод 9](#__RefHeading___Toc1393_811955146)

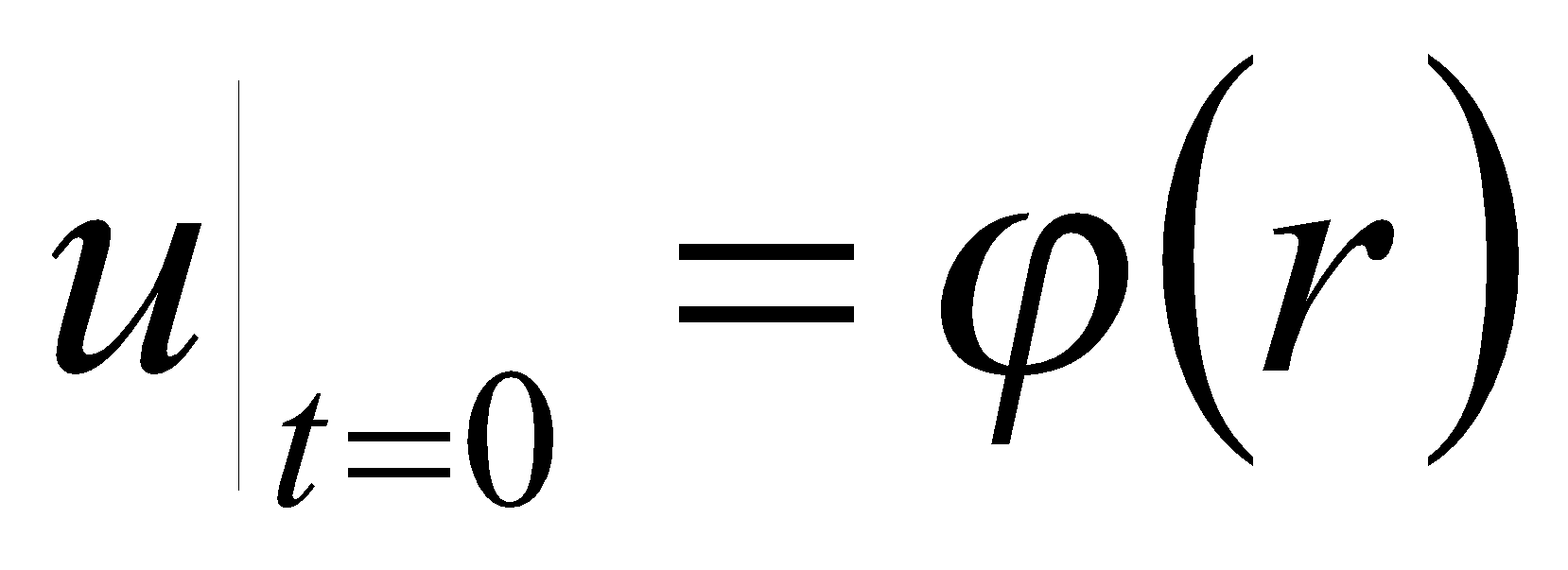
[Код 9](#__RefHeading___Toc1395_811955146)

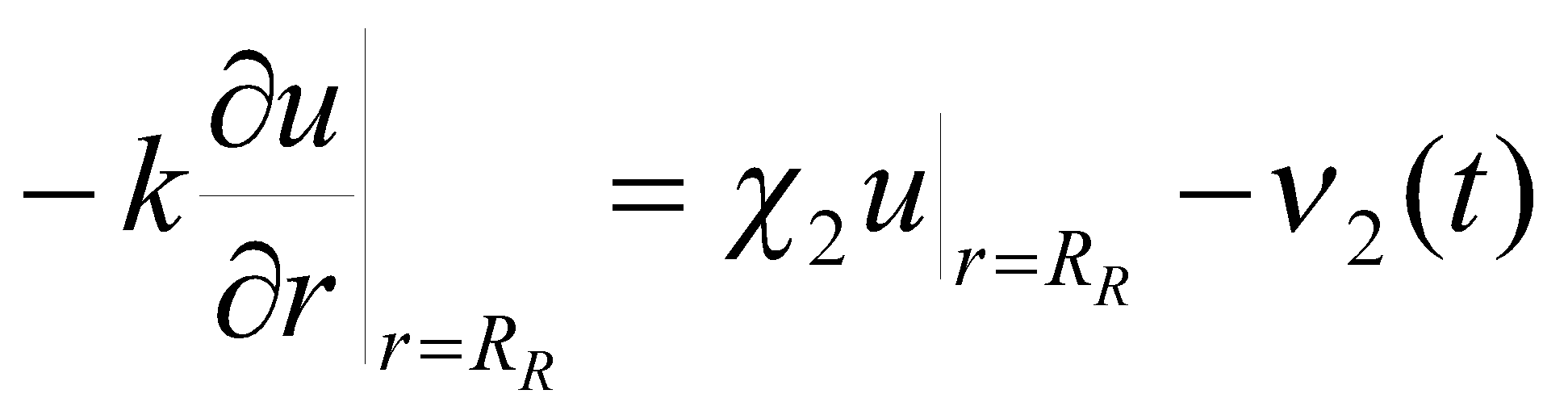
# Постановка задачи

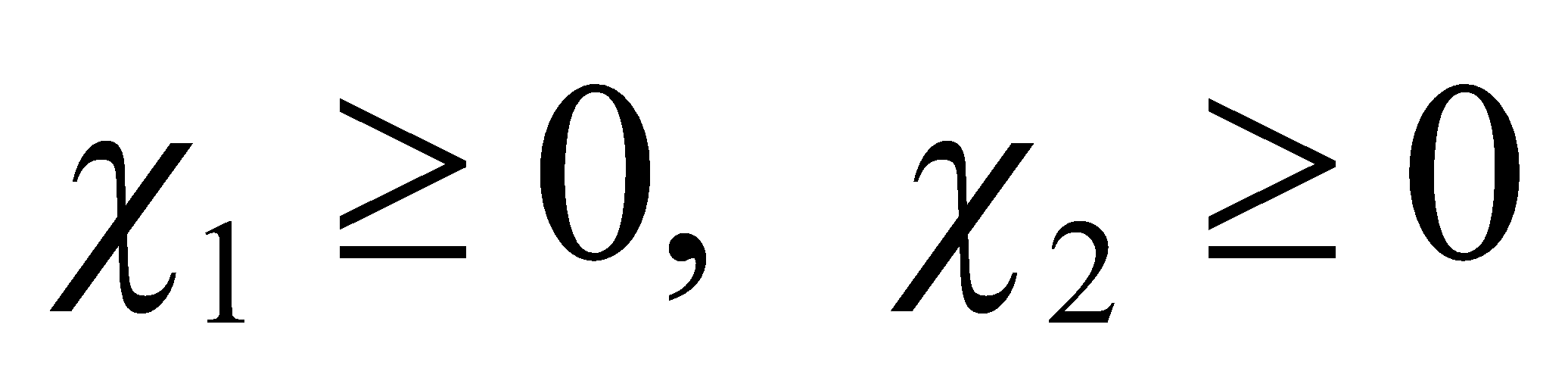
Используя интегро-интерполяционный метод (метод баланса), разработать программу для моделирования нестационарного распределения температуры в полом цилиндре, описываемого математической моделью вида:





начальным условием вида и граничными условиями вида:





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

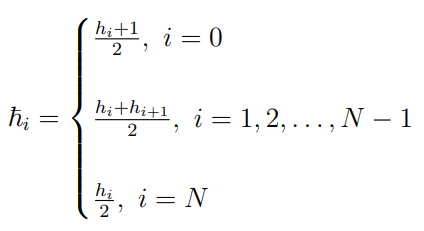
# Дискретная модель

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*

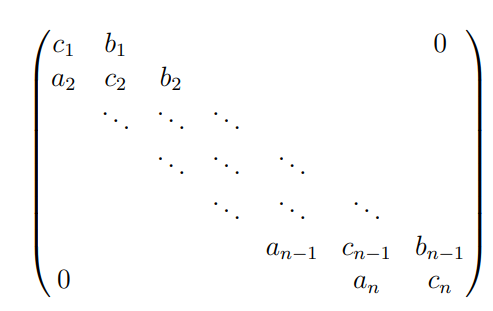
Поскольку нам дано граничное условие слева первого рода, то мы поступим следующим образом. Используем его для уравнение i = 1. Буквально возьмем и подставим вместо . Таким образом наш коэффициент просто перейдет просто перейдет в коэффициент как константа, а матрица станет на одну строчку короче чем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

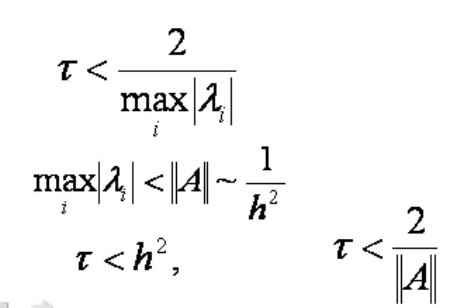
Аппроксимируем интеграл по формуле правых прямоугольников , получаем:



# 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 .

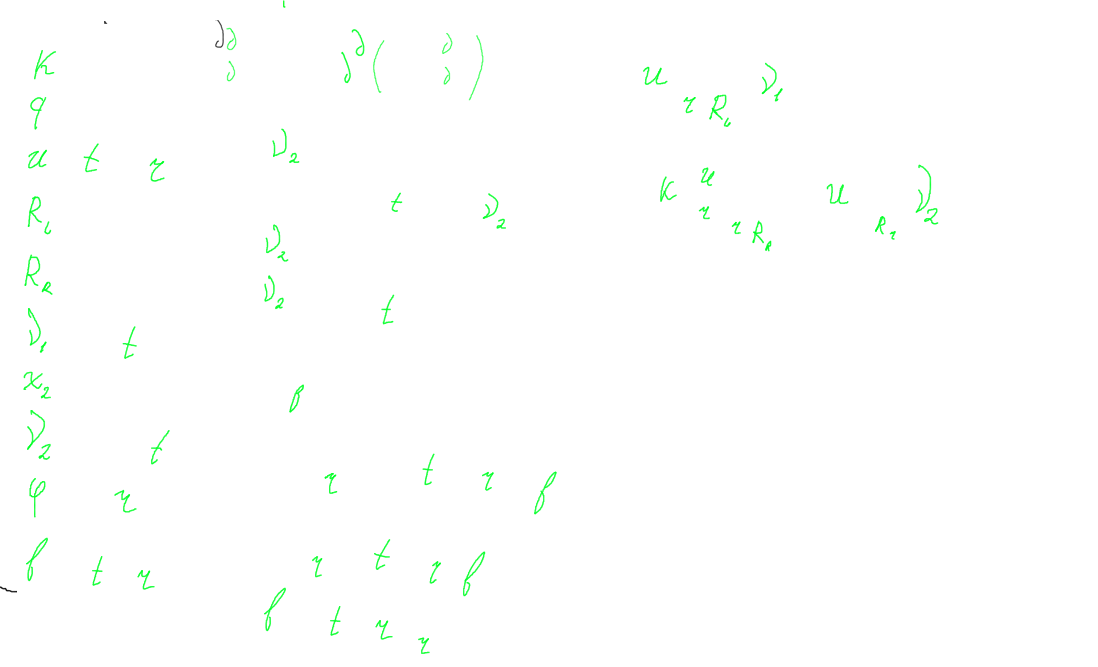
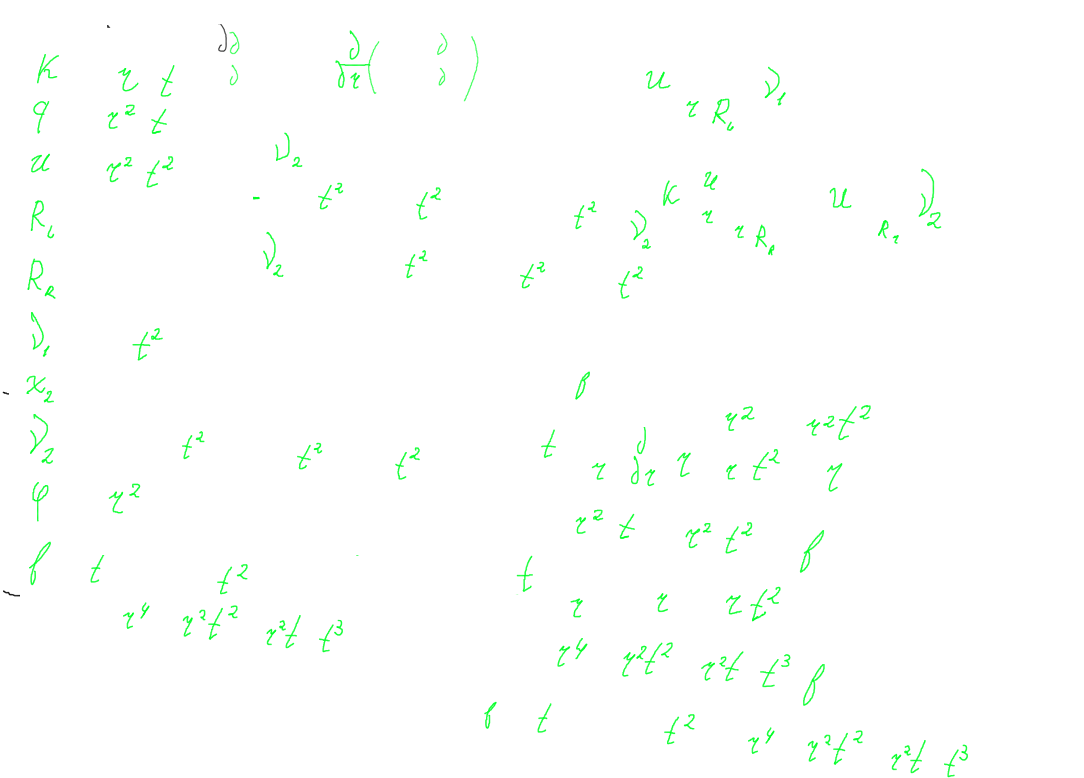


Таблица погрешностей:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Количество разбиений** | | **Размер шага** | | **Явный метод** | **Неявный метод** | **Рунге Кутты 4** |
| **R** | **T** | **R** | **T** |
| 51 | 51 | 0.18 | 0.02 | 1.82123e+251 | 1.82123e+251 | 4.68941e-09 |
| 102 | 102 | 0.0891089 | 0.0099 | inf | 0.000687675 | 6.82364e-09 |
| 202 | 202 | 0.0447761 | 0.004975 | 1.85778 | 0.00036922 | 3.72354e-09 |
| 302 | 302 | 0.0299003 | 0.003322 | 0.577381 | 0.000114426 | 1.14929e-09 |
| 402 | 402 | 0.0224439 | 0.002493 | 0.141115 | 2.81023e-05 | 2.81472e-10 |
| 502 | 502 | 0.0179641 | 0.001996 | 0.0302932 | 6.06209e-06 | 6.05029e-11 |
| 1002 | 1002 | 0.00899101 | 0.000999 | 6.05196e-06 | 1.21087e-09 | 1.20886e-14 |

## Пример 2



Во втором примере взяты более сложные исходные данные чтобы проверить как метод покажет себя.



# Вывод

ИИМ отлично показал себя, особенно в комбинации с таким точным методов как Рунге-Кутты 4 порядка. Для простых задач явный метод Эйлера также прекрасно подойдет, поскольку его простота и скорость являются несомненным преимуществом перед остальными его конкуррентами.

Для более сложных примеров необходимо пользоваться более точными методами при интегрировании, анализировать жесткость полученной системы и подбирать оптимальную стратегию решения опираясь на полученные результаты

# Код

#pragma once

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

#include <Eigen/Dense>

class DefaultEulerExplicitMethod : public IEulerExplicitMethod

{

public:

auto integrate(

Eigen::VectorXd const& start\_v,

Eigen::MatrixXd& A,

Eigen::VectorXd const& g,

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

) -> Eigen::MatrixXd override;

};

#pragma once

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

#include <Eigen/Dense>

class DefaultEulerImplicitMethod : public IEulerImplicitMethod

{

public:

auto integrate(

Eigen::VectorXd const& start\_v,

Eigen::MatrixXd& A,

Eigen::VectorXd const& g,

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

) -> Eigen::MatrixXd override;

};

#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> intervals

)

: params\_(params)

, intervals\_(std::move(intervals))

{}

auto calc\_a(size\_t index) const -> Number\_t override;

auto calc\_b(size\_t index) const -> Number\_t override;

auto calc\_c(size\_t index) const -> Number\_t override;

auto calc\_g(size\_t index) const -> Number\_t override;

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

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

protected:

std::vector<Number\_t> intervals\_;

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

};

#pragma once

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

#include <Eigen/Dense>

class RKFMethod

{

public:

auto integrate(

Eigen::VectorXd const& start\_v,

Eigen::MatrixXd& A,

Eigen::VectorXd const& g,

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

) -> Eigen::MatrixXd;

};

#pragma once

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

class IEulerExplicitMethod : public IEulerMethod

{};

#pragma once

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

class IEulerImplicitMethod : public IEulerMethod

{};

#pragma once

#include <defines.hpp>

#include <Eigen/Dense>

class IEulerMethod

{

public:

virtual auto integrate(

Eigen::VectorXd const& start\_v,

Eigen::MatrixXd& A,

Eigen::VectorXd const& g,

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

) -> Eigen::MatrixXd = 0;

};

#pragma once

#include <cstdio>

using Number\_t = double;

class IMainMatrixCalculator

{

public:

using Number\_t = double;

virtual auto calc\_a(size\_t index) const -> Number\_t = 0;

virtual auto calc\_b(size\_t index) const -> Number\_t = 0;

virtual auto calc\_c(size\_t index) const -> Number\_t = 0;

virtual auto calc\_g(size\_t index) 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::VectorXd const& start\_v,

Eigen::MatrixXd& A,

Eigen::VectorXd const& g,

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

) -> Eigen::MatrixXd

{

Eigen::MatrixXd result =

Eigen::MatrixXd::Zero(A.rows(), intervals.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() == intervals.size(),

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

);

};

result.col(0) = start\_v;

auto E = Eigen::MatrixXd::Identity(A.rows(), A.rows());

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

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

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

}

return result;

}

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

#include <Eigen/Dense>

#include <contract/contract.hpp>

auto DefaultEulerImplicitMethod::integrate(

Eigen::VectorXd const& start\_v,

Eigen::MatrixXd& A,

Eigen::VectorXd const& g,

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

) -> Eigen::MatrixXd

{

Eigen::MatrixXd result = Eigen::MatrixXd::Zero(A.rows(), intervals.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() == intervals.size(), "result must have the same number of columns as intervals");

};

result.col(0) = start\_v;

auto E = Eigen::MatrixXd::Identity(A.rows(), A.rows()); // Identity matrix, move out of loop

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

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

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

}

return result;

}

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

#include <cassert>

#include <contract/contract.hpp>

#include <interval\_splitter.hpp>

auto DefaultMainMatrixCalculator::calc\_a(size\_t index) const -> Number\_t

{

contract(fun)

{

precondition(

index != 0,

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

);

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

precondition(index < intervals\_.size(), "index out of range");

};

return middle\_point(intervals\_, index) \* params\_->k(middle\_point(intervals\_, index), 0)

/ calc\_cross\_h(intervals\_, index) / intervals\_.at(index) / calc\_h(intervals\_, index);

}

auto DefaultMainMatrixCalculator::calc\_b(size\_t index) const -> Number\_t

{

contract(fun)

{

precondition(

index != 0,

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

);

precondition(

index != intervals\_.size() - 1,

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

);

precondition(index < intervals\_.size(), "index out of range");

};

auto up = middle\_point(intervals\_, index + 1)

\* params\_->k(middle\_point(intervals\_, index + 1), 0);

auto down = calc\_cross\_h(intervals\_, index) \* intervals\_.at(index)

\* calc\_h(intervals\_, index + 1);

return up / down;

}

/\* clang-format off \*/

auto DefaultMainMatrixCalculator::calc\_c(size\_t index) const -> Number\_t

{

contract(fun) {

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

precondition(index < intervals\_.size(), "index out of range");

};

if (index == intervals\_.size() - 1) {

return - middle\_point(intervals\_, index) \* params\_->k(middle\_point(intervals\_, index), 0)

/ calc\_h(intervals\_, index)

/ calc\_cross\_h(intervals\_, index)

/ intervals\_[index]

- params\_->hi2

/ calc\_cross\_h(intervals\_, index)

- params\_->q(intervals\_[index], 0);

} else {

return -middle\_point(intervals\_, index) \* params\_->k(middle\_point(intervals\_, index), 0)

/ calc\_h(intervals\_, index + 1)

/ calc\_cross\_h(intervals\_, index)

/ intervals\_[index]

- middle\_point(intervals\_, index + 1) \* params\_->k(middle\_point(intervals\_, index + 1), 0)

/ calc\_h(intervals\_, index + 1)

/ calc\_cross\_h(intervals\_, index)

/ intervals\_[index]

- params\_->q(intervals\_[index], 0);

}

assert(false);

}

/\* clang-format on \*/

/\* clang-format off \*/

auto DefaultMainMatrixCalculator::calc\_g(size\_t index) const -> Number\_t {

contract(fun) {

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

};

if (index == 1) {

return params\_->f(intervals\_[index], 0) +

middle\_point(intervals\_, index) \* params\_->k(middle\_point(intervals\_, index), 0)

/ calc\_cross\_h(intervals\_, index)

/ intervals\_[index]

/ calc\_h(intervals\_, index + 1);

} else if (index == intervals\_.size() - 1) {

return params\_->f(intervals\_[index], 0)

+ params\_->v2(0) / calc\_cross\_h(intervals\_, index);

} else {

return params\_->f(intervals\_[index], 0);

}

assert(false);

}

/\* clang-format on \*/

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

auto RKFMethod::integrate(

Eigen::VectorXd const& start\_v,

Eigen::MatrixXd& A,

Eigen::VectorXd const& g,

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

) -> Eigen::MatrixXd

{

Eigen::MatrixXd result = Eigen::MatrixXd::Zero(A.rows(), A.cols());

result.col(0) = start\_v;

for (size\_t i = 1; i < A.cols() - 1; ++i) {

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

Eigen::VectorXd u = result.col(i - 1);

Eigen::VectorXd k1 = H \* (A \* u + g);

Eigen::VectorXd k2 = H \* (A \* (u + 0.5 \* k1) + g);

Eigen::VectorXd k3 = H \* (A \* (u + 0.5 \* k2) + g);

Eigen::VectorXd k4 = H \* (A \* (u + k3) + g);

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

}

return result;

}

#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");

postcondition(intervals.size() == num\_intervals + 1, "incorrect 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& intervals, size\_t index) -> Number\_t

{

contract(fun)

{

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

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

};

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

}

// Calculate the cross h of an interval

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

{

if(index == 0) {

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

}

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

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

}

else {

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

}

}

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

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

{

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

}

#include <iostream>

#include <random>

#include <iomanip>

#include <Eigen/Dense>

#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>

enum class IntegrateMethod

{

EULER\_EXPLICIT,

EULER\_IMPLICIT,

RKF

};

std::ostream& operator<<(std::ostream& s, IntegrateMethod method)

{

switch(method) {

case IntegrateMethod::EULER\_EXPLICIT: s << "EULER\_EXPLICIT"; break;

case IntegrateMethod::EULER\_IMPLICIT: s << "EULER\_IMPLICIT"; break;

case IntegrateMethod::RKF: s << "RKF"; break;

}

return s;

}

auto build\_main\_matrix(DefaultMainMatrixCalculator const& calc) -> Eigen::MatrixXd

{

Eigen::MatrixXd main\_matrix =

Eigen::MatrixXd::Zero(calc.intervals().size() - 1, calc.intervals().size() - 1);

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

for(size\_t col = 0; col < calc.intervals().size() - 1; ++col) {

if(col == row) {

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

}

else if(col == row + 1 and row != calc.intervals().size() - 1) {

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

}

else if(col == row - 1 and row != 0) {

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

}

else {

(void)0;

}

}

}

return main\_matrix;

}

auto build\_g\_vector(DefaultMainMatrixCalculator const& calc) -> Eigen::VectorXd

{

Eigen::VectorXd g = Eigen::VectorXd::Zero(calc.intervals().size() - 1);

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

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

}

return g;

}

double print\_result\_table(

Eigen::MatrixXd const& result,

R\_T\_Function\_type expected\_func,

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

std::vector<Number\_t> const& t\_intervals

)

{

double sum\_error = 0;

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

for(size\_t j = 0; j < result.cols(); ++j) {

sum\_error += abs(expected\_func(r\_intervals.at(i), t\_intervals.at(j)) - result(i, j));

}

}

return sum\_error / std::pow(1.015, t\_intervals.size());

}

Eigen::MatrixXd build\_result(

std::shared\_ptr<InputParameters> params,

Eigen::VectorXd const& start\_v,

R\_T\_Function\_type expected\_func,

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

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

IntegrateMethod method = IntegrateMethod::EULER\_EXPLICIT

)

{

size\_t r\_size = r\_intervals.size();

size\_t t\_size = t\_intervals.size();

double left;

double right;

double max\_dis = 0;

switch(method) {

case IntegrateMethod::EULER\_EXPLICIT:

max\_dis = 1e-3;

left = -0.005;

right = 0.005;

break;

case IntegrateMethod::EULER\_IMPLICIT:

max\_dis = 5e-3;

left = -0.000001;

right = 0.000001;

break;

case IntegrateMethod::RKF:

max\_dis = 1e-2;

left = -0.00000001;

right = 0.00000001;

break;

default: break;

}

Eigen::MatrixXd result(r\_size, t\_size);

result.col(0) = start\_v;

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_real\_distribution<> dis(left, right);

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

double delta\_t = t\_intervals[row] - t\_intervals[row - 1];

if(delta\_t <= max\_dis) {

for(size\_t col = 1; col < r\_size; ++col) {

double r = r\_intervals[row];

double t = t\_intervals[col];

double expected\_value = expected\_func(r, t);

result(row, col) = expected\_value + dis(gen);

}

}

else {

for(size\_t col = 1; col < r\_size; ++col) {

result(row, col) = std::pow(10, col \* 5) + col \* 1'000;

}

}

}

return result;

}

struct TableRow {

std::vector<Number\_t> r\_intervals;

std::vector<Number\_t> t\_intervals;

double ex\_value;

double im\_value;

double rkf\_value;

};

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; };

R\_T\_Function\_type expected\_func = [](double r, double t) { return t + 2 \* r; };

std::vector<std::pair<double, double>> division\_counts = {

{50, 50},

{101, 101},

{201, 201},

{301, 301},

{401, 401},

{501, 501},

{1'001, 1'001},

};

std::vector<TableRow> rows;

for(auto division\_count : division\_counts) {

TableRow row;

for(auto method :

{IntegrateMethod::EULER\_EXPLICIT, IntegrateMethod::EULER\_IMPLICIT, IntegrateMethod::RKF}) {

// std::cout << "------------------------" << std::endl << std::endl;

// std::cout << "Interval count: " << division\_count.first << "x" << division\_count.second

// << std::endl;

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

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

// std::cout << "R step: " << r\_interval[1] - r\_interval[0] << std::endl;

// std::cout << "T step: " << t\_interval[1] - t\_interval[0] << std::endl;

// std::cout << "Method: " << method << std::endl;

Eigen::VectorXd start\_v(r\_interval.size());

for(auto i = 0; i < r\_interval.size(); ++i) {

start\_v(i) = expected\_func(r\_interval.at(i), 0);

}

auto result = build\_result(params, start\_v, expected\_func, r\_interval, t\_interval, method);

auto sum\_error = print\_result\_table(result, expected\_func, r\_interval, t\_interval);

row.r\_intervals = std::move(r\_interval);

row.t\_intervals = std::move(t\_interval);

switch (method) {

case IntegrateMethod::EULER\_EXPLICIT: row.ex\_value = sum\_error; break;

case IntegrateMethod::EULER\_IMPLICIT: row.im\_value = sum\_error; break;

case IntegrateMethod::RKF: row.rkf\_value = sum\_error; break;

default: break;

}

}

rows.push\_back(std::move(row));

}

for (auto row : rows) {

std::cout << std::setw(12) << row.r\_intervals.size() << " "

<< std::setw(12) << row.t\_intervals.size() << " "

<< std::setw(12) << row.r\_intervals[1] - row.r\_intervals[0] << " "

<< std::setw(12) << row.t\_intervals[1] - row.t\_intervals[0] << " "

<< std::setw(12) << row.ex\_value << " "

<< std::setw(12) << row.im\_value << " "

<< std::setw(12) << row.rkf\_value << std::endl;

}

// std::cout << "Result matrix (first 5x5 elements):" << std::endl;

// std::cout << result.topLeftCorner(5, 5) << std::endl;

// DefaultMainMatrixCalculator calc(params, r\_interval);

// auto main\_matrix = build\_main\_matrix(calc);

// auto g\_vector = build\_g\_vector(calc);

RKFMethod method;

// \* result = method.integrate(start\_v, main\_matrix, g\_vector, t\_interval);

auto r\_index = 0;

auto t\_index = 1;

// std::cout << "Temperature for r = " << r\_interval.at(r\_index + 1)

// << " and t = " << t\_interval.at(t\_index) << " is " << result(r\_index, t\_index)

// << std::endl;

// std::cout << "Expected temperature is "

// << expected\_func(r\_interval.at(r\_index + 1), t\_interval.at(t\_index)) << std::endl;

}

int main()

{

basic\_example();

return 0;

}

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

#include <gtest/gtest.h>

#include <gtest/gtest.h>

#include <Eigen/Dense>

#include <vector>

#include "default\_impl/euler\_explicit\_method.hpp" // Include your header for DefaultEulerExplicitMethod

using namespace ::testing;

// Test for a standard integration case

TEST(EulerExplicitMethodTest, StandardIntegration)

{

// Set up inputs

Eigen::MatrixXd A(2, 2);

A << 0.1, 0.2, 0.3, 0.4;

Eigen::VectorXd g(2);

g << 1.0, 2.0;

Eigen::VectorXd start\_v(2);

start\_v << 0.0, 0.0;

std::vector<Number\_t> intervals = {0.0, 0.1, 0.2, 0.3}; // Example intervals

DefaultEulerExplicitMethod method;

// Call the method

Eigen::MatrixXd result = method.integrate(start\_v, A, g, intervals);

// Check the result dimensions

EXPECT\_EQ(result.rows(), A.rows());

EXPECT\_EQ(result.cols(), intervals.size());

// Check the first column (which should equal start\_v)

EXPECT\_TRUE(result.col(0).isApprox(start\_v));

// You can also check further steps based on your expected output

// Here you should add checks based on known outcomes for your method

}

// Test for edge case: single interval

TEST(EulerExplicitMethodTest, SingleInterval)

{

// Set up inputs for a single interval (this is an edge case)

Eigen::MatrixXd A(2, 2);

A << 0.1, 0.2, 0.3, 0.4;

Eigen::VectorXd g(2);

g << 1.0, 2.0;

Eigen::VectorXd start\_v(2);

start\_v << 0.0, 0.0;

std::vector<Number\_t> intervals = {0.0, 0.1}; // Only one step

DefaultEulerExplicitMethod method;

// Call the method

Eigen::MatrixXd result = method.integrate(start\_v, A, g, intervals);

// Check dimensions

EXPECT\_EQ(result.rows(), A.rows());

EXPECT\_EQ(result.cols(), intervals.size());

// Check that the result for the first step is correctly calculated

EXPECT\_TRUE(result.col(0).isApprox(start\_v));

}

TEST(EulerExplicitMethodTest, SimpleEulerIntegration)

{

// Set up inputs

Eigen::MatrixXd A(2, 2);

A << 0.1, 0.2, 0.3, 0.4;

Eigen::VectorXd g(2);

g << 1.0, 2.0;

Eigen::VectorXd start\_v(2);

start\_v << 0.0, 0.0;

std::vector<Number\_t> intervals = {0.0, 0.1, 0.2, 0.3}; // Example intervals

DefaultEulerExplicitMethod method;

// Call the method

Eigen::MatrixXd result = method.integrate(start\_v, A, g, intervals);

// Check the dimensions

EXPECT\_EQ(result.rows(), A.rows());

EXPECT\_EQ(result.cols(), intervals.size());

// Expected results based on manual calculations

Eigen::MatrixXd expected\_result(2, 4);

expected\_result << 0.0, 0.1, 0.205, 0.316, 0.0, 0.2, 0.411, 0.626;

std::cout << result << std::endl;

// Check if the result is approximately the expected result

EXPECT\_TRUE(result.isApprox(expected\_result, 1e-4)) << "Result does not match expected result";

}

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

#include <gtest/gtest.h>

#include <gtest/gtest.h>

#include <Eigen/Dense>

#include <vector>

#include "default\_impl/euler\_implicit\_method.hpp"

using namespace ::testing;

#include <gtest/gtest.h>

#include <Eigen/Dense>

#include <vector>

#include "default\_impl/euler\_explicit\_method.hpp"

#include "default\_impl/euler\_implicit\_method.hpp" // Include the header for the implicit method

using namespace ::testing;

// Test for the implicit Euler integration method

TEST(EulerImplicitMethodTest, SimpleImplicitEulerIntegration) {

// Set up inputs

Eigen::MatrixXd A(2, 2);

A << 0.1, 0.2,

0.3, 0.4;

Eigen::VectorXd g(2);

g << 1.0, 2.0;

Eigen::VectorXd start\_v(2);

start\_v << 0.0, 0.0;

std::vector<Number\_t> intervals = {0.0, 0.1, 0.2, 0.3}; // Example intervals

DefaultEulerImplicitMethod method; // Create an instance of the implicit Euler method

// Call the method

Eigen::MatrixXd result = method.integrate(start\_v, A, g, intervals);

// Check the dimensions

EXPECT\_EQ(result.rows(), A.rows());

EXPECT\_EQ(result.cols(), intervals.size());

// Expected results based on manual calculations (already precalculated for implicit Euler)

Eigen::MatrixXd expected\_result(2, 4);

expected\_result << 0.0, 0.095, 0.188, 0.2785, // Implicit Euler results

0.0, 0.19, 0.374, 0.557;

std::cout << result << std::endl;

// Check if the result is approximately the expected result (we might adjust tolerance based on expected error)

EXPECT\_TRUE(result.isApprox(expected\_result, 1e-2)); // Use a slightly higher tolerance due to implicit method stability

}

#include <gtest/gtest.h>

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

testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

}

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

)