Лабораторная работа №1

По дисциплине

«Разработка программного обеспечения для моделирования физических процессов»

Выполнил студент

группы № 5130904/10101

Руководитель Воскобойников С.П

Оглавление

[Постановка задачи 3](#_Toc184591977)

[Аппроксимация 4](#_Toc184591978)

[Разностная схема 5](#_Toc184591979)

[Метод прогонки 6](#_Toc184591980)

[Оценка погрешности 7](#_Toc184591981)

[Невязка разностной схемы 7](#_Toc184591982)

[Анализ порядка аппроксимации 7](#_Toc184591983)

[Оценка работы программы на различных примерах 8](#_Toc184591984)

[Пример 1 8](#_Toc184591985)

[Пример 2 9](#_Toc184591986)

[Пример 3 10](#_Toc184591987)

[Вывод 12](#_Toc184591988)

[Код программы 13](#_Toc184591989)

[main.rs 13](#_Toc184591990)

[coeff\_calculator.rs 20](#_Toc184591991)

[solver.rs 24](#_Toc184591992)

[stepping.rs 30](#_Toc184591993)

# Постановка задачи

Вариант № CP1

Постановка задачи. Вариант CP. Используя интегро-интерполяционный метод (метод баланса), разработать программу для моделирования стационарного распределения температуры в полом цилиндре, описываемого математической моделью вида



с граничными условиями

 

# Аппроксимация



, ,

Аппроксимация уравнения слева с граничными условиями первого рода:

Аппроксимация уравнения в середине:

Изображение выглядит как текст, рукописный текст, Шрифт, белый

Автоматически созданное описание

Аппроксимация уравнения справа с граничными условиями первого рода:

Получаем систему:



# Разностная схема

Для i = 0

b0 = 0 c0 = 1 g0 = i = 0

Для i = 1,2,…,N-1

Для i = N

aN = 0 cN = 1 gN = i = N

# Метод прогонки

Изображение выглядит как текст, чек, Шрифт, снимок экрана

Автоматически созданное описание

Изображение выглядит как текст, чек, снимок экрана, Шрифт

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт, диаграмма

Автоматически созданное описание

# Оценка погрешности

## Невязка разностной схемы

Невязка для i = 1,2,…,N-1:

## Анализ порядка аппроксимации

1. Левое граничное условие – условие первого порядка. Оно аппроксимируется точно, следовательно, не выносит никакой погрешности.
2. Результат преобразований уравнения невязки внутри интервала:

*Порядок аппроксимации* ***p=2***

1. Правое граничное условие – условие первого порядка. Оно аппроксимируется точно, следовательно, не выносит никакой погрешности.

# Оценка работы программы на различных примерах

## Пример 1

 

k = 2 RL = 1

q = 3 RR = 11

U = 2r

Данный пример предполагает k и q константные для простоты. Введем данные в программу и посмотрим на результат.

|  |  |
| --- | --- |
| шаги | Максимальная неточность |
| 2 | 0e0 |
| 4 | 1.7763568394002505e-15 |
| 8 | 3.552713678800501e-15 |
| 16 | 7.105427357601002e-15 |
| 32 | 7.105427357601002e-15 |
| 64 | 7.105427357601002e-15 |
| 128 | 2.1316282072803006e-14 |

При уменьшении шага ошибка сначала быстро снижается, а затем стабилизируется на уровне порядка 10-15. Это свидетельствует о том, что метод сходимости достиг предела точности, который связан с ограничениями вычислительных ресурсов и машинной точности.

Ошибки в вычислениях на шаге 128 оказались немного больше, что можно объяснить ограничениями машинной точности и процессом округления чисел в вычислениях.

Таким образом, несмотря на влияние различных факторов, таких как аппроксимация производных и машинная точность, численный метод в данной задаче дал достаточно точный результат. Ошибка стабилизировалась на уровне порядка 10-15.

## Пример 2

 

k = 2r RL = 1

q = 5 RR = 10

U = 10r

|  |  |
| --- | --- |
| шаги | Максимальная неточность |
| 2 | 0e0 |
| 4 | 7.105427357601002e-15 |
| 8 | 0e0 |
| 16 | 0e0 |
| 32 | 5.684341886080802e-14 |
| 64 | 6.394884621840902e-14 |
| 128 | 3.126388037344441e-13 |

Анализ:

На первых шагах (2, 4, 8, 16) ошибки равны нулю или очень малы, что свидетельствует о хорошем соответствии численного решения аналитическому результату.

На более мелких шагах (32, 64, 128) ошибка начинает увеличиваться, что может быть связано с числовыми погрешностями, возникающими из-за округления и ограничений машинной точности.

* Машинная точность: при шагах разбиения 32 и выше ошибка становится более заметной, что может быть связано с погрешностями, вызванными ограничениями машинной точности, особенно на последнем шаге (128), где ошибка достигла порядка 10-13. Это подтверждает, что слишком малые шаги не всегда приводят к улучшению точности из-за ограничений вычислительных ресурсов.
* Ошибка численного решения сильно зависит от выбранного шага. При больших шагах ошибка минимальна, но с уменьшением шага ошибка начинает расти, что является следствием увеличения погрешности, вызванной машинной точностью и округлением.

## Пример 3

 

k = 2r2 RL = 1

q = 5r2 RR = 10

U = 10r2

|  |  |
| --- | --- |
| шаги | Максимальная неточность |
| 2 | 5.055381118409684e0 |
| 4 | 3.366417907584548e0 |
| 8 | 1.5941547976068264e0 |
| 16 | 5.077037091673624e-1 |
| 32 | 1.3170595958556675e-1 |
| 64 | 3.327196880501049e-2 |
| 128 | 8.340581405210656e-3 |

На первых шагах (2, 4) ошибка довольно велика и составляет порядка 100, что говорит о низкой точности решения при относительно крупных шагах разбиения.

При дальнейшем уменьшении шага (8, 16, 32, 64, 128) ошибка значительно снижается. На шаге 128 максимальная ошибка 8,34 \* 10-3, что указывает на улучшение точности решения.

* Ошибка в решении уменьшается с уменьшением шага разбиения, что ожидаемо, поскольку меньший шаг повышает точность численного метода. Однако, как видно из таблицы, снижение ошибки происходит с замедлением, что может свидетельствовать о пределе точности, с которым метод способен работать.
* Погрешности, возникающие при аппроксимации производных через конечные разности, с уменьшающимся шагом начинают снижаться, однако на начальных шагах (например, на шаге 2) погрешности остаются достаточно высокими. Это объясняется недостаточной точностью аппроксимации для более крупных шагов.
* При малых шагах ошибка стабилизируется, что свидетельствует о приближении решения к аналитическому результату. Однако на малых шагах (например, шаг 128) ошибка продолжает снижаться, но её величина остаётся ощутимой и на уровне порядка 10-3, что связано с ограничениями машинной точности и округлением.

Численное решение задачи показывает, что для более крупных шагов ошибка значительна, и её снижение с уменьшением шага разбиения происходит плавно. Ошибка стабилизируется на уровне порядка 10-3 при шаге 128.

# Вывод

**Вывод:**

Применённый алгоритм эффективно решает дифференциальные уравнения второго порядка с различными типами граничных условий, продемонстрировав хорошие результаты в решении задач с переменными коэффициентами и константными значениями функций. В частности, решение было получено для уравнений с различными параметрами для различных значений шагов, что позволило провести детальный анализ точности решения.

Для простых случаев (Пример 1) с константными коэффициентами, таких как 2k(r) = 2 и q(r)=3, ошибка численного решения быстро снижалась с уменьшением шага, стабилизируясь на уровне порядка 10-15, что подтверждает высокую точность алгоритма в таких случаях. Однако для более сложных случаев (Пример 2 и 3), где функции k(r) и q(r) зависимы от r, точность решения также улучшалась с уменьшением шага, но на самых мелких шагах (32, 64, 128) начинали проявляться погрешности, связанные с ограничениями машинной точности и процессом округления.

Численные ошибки для шагов 2, 4, 8 и 16 оставались минимальными. Однако при дальнейшем уменьшении шага (например, на шаге 128 в Примере 3) ошибка стабилизировалась, и её величина незначительно возросла из-за влияния числовых погрешностей и ограничений вычислительных ресурсов. Этот эффект подчёркивает важность оптимального выбора шага для сбалансированного достижения точности и эффективности вычислений.

Таким образом, алгоритм продемонстрировал высокую эффективность и точность при решении дифференциальных уравнений второго порядка с различными граничными условиями и параметрами.

# Код программы

## main.rs

mod math;

use math::{coeff\_calculator::\*, solver};

use math::stepping::{IntervalSplitter, Stepping};

use nalgebra::DVector;

fn generate\_points(left: f64, right: f64, step\_count: usize) -> Vec<f64> {

let step\_size = (right - left) / step\_count as f64;

let mut points: Vec<f64> = Vec::with\_capacity(step\_count + 1);

for i in 0..=step\_count {

points.push(left + i as f64 \* step\_size);

}

points

}

fn exercise\_accuracy\_base\_example() {

const LEFT: f64 = 1.0;

const RIGHT: f64 = 11.0;

// k(r) = 1, q(r) = 0, f(r) = 0 for a simple cylindrical heat conduction case

let kfunc = LambdaFunction::from(|\_r: f64| 2.0);

let qfunc = LambdaFunction::from(|\_r: f64| 3.0);

let ffunc = LambdaFunction::from(|r: f64| -4.0 / r + 6.0 \* r);

// Boundary conditions: T(0) = 0 (center), T(1) = 1 (outer radius)

let y1 = 2.0;

let y2 = 22.0;

// The 'n' value for cylindrical symmetry (n=1 for 2D axisymmetric cylindrical case)

let n = 1;

let original\_function = |r: f64| 2.0\*r;

let mut step\_count\_vec = Vec::<i32>::new();

let mut avg\_inaccuracy\_vec = Vec::<f64>::new();

let mut max\_inaccuracy\_vec = Vec::<f64>::new();

let mut inaccuracy\_in\_first\_half\_vec = Vec::<f64>::new();

let mut inaccuracy\_in\_second\_half\_vec = Vec::<f64>::new();

for step\_count in [2, 4, 8, 16, 32, 64, 128].iter() {

let points = generate\_points(LEFT, RIGHT, \*step\_count);

let splitter = IntervalSplitter::new(points.clone());

let coeff\_calculator\_v =

math::coeff\_calculator::first\_first\_calculator::FirstFirstCalculator::new(

splitter,

&kfunc,

&qfunc,

&ffunc,

y1,

y2,

n,

);

let (A, g) = math::coeff\_calculator::matrix\_building::build\_tridiagonal\_matrix(&coeff\_calculator\_v);

let calculated\_v = solver::solve(&A, &g);

let expected\_v: DVector<f64> = DVector::from\_vec(points.iter().map(|x| original\_function(\*x)).collect());

let accuracy = &calculated\_v - &expected\_v;

let avg\_inaccuracy = accuracy.fold(0.0, |acc, x| acc + x.abs()) / (accuracy.len() as f64 + 1.0);

step\_count\_vec.push(\*step\_count as i32);

avg\_inaccuracy\_vec.push(avg\_inaccuracy);

max\_inaccuracy\_vec.push(accuracy.iter().fold(0.0, |max, x| x.abs().max(max)));

let first\_half\_inaccuracy: f64 = accuracy

.iter()

.take(accuracy.len() / 2)

.fold(0.0, |acc, x| acc + x.abs());

let second\_half\_inaccuracy: f64 = accuracy

.iter()

.skip(accuracy.len() / 2)

.fold(0.0, |acc, x| acc + x.abs());

inaccuracy\_in\_first\_half\_vec.push(first\_half\_inaccuracy);

inaccuracy\_in\_second\_half\_vec.push(second\_half\_inaccuracy);

}

let mut rel\_inaccuracy: Vec<f64> = vec![];

for i in 1..avg\_inaccuracy\_vec.len() {

rel\_inaccuracy.push(avg\_inaccuracy\_vec[i-1] / avg\_inaccuracy\_vec[i]);

}

let digits\_after\_dot = 16;

println!("{}", "-".repeat(136));

println!(

"| {:>5} | {:width$.width$} | {:width$.width$} | {:width$.width$} | {:width$.width$} | {:width$.width$} |",

"steps",

"avg inaccuracy",

"max inaccuracy",

"first half inaccuracy",

"second half inaccuracy",

"rel inaccuracy",

width = digits\_after\_dot + 6

);

println!("{}", "-".repeat(136));

for i in 0..step\_count\_vec.len() {

println!(

"| {:>5} | {:width$e} | {:width$e} | {:width$e} | {:width$e} | {:width$.width$} |",

step\_count\_vec[i],

avg\_inaccuracy\_vec[i],

max\_inaccuracy\_vec[i],

inaccuracy\_in\_first\_half\_vec[i],

inaccuracy\_in\_second\_half\_vec[i],

if i > 0 { rel\_inaccuracy[i - 1] } else { 0.0 },

width = digits\_after\_dot + 6

);

}

println!("{}", "-".repeat(136));

}

fn exercise\_liner\_function\_base\_example() {

const LEFT: f64 = 1.0;

const RIGHT: f64 = 10.0;

// k(r) = 1, q(r) = 0, f(r) = 0 for a simple cylindrical heat conduction case

let kfunc = LambdaFunction::from(|r: f64| 2.0\*r);

let qfunc = LambdaFunction::from(|\_r: f64| 5.0);

let ffunc = LambdaFunction::from(|r: f64| 50.0\*r - 40.0);

// Boundary conditions: T(0) = 0 (center), T(1) = 1 (outer radius)

let y1 = 10.0;

let y2 = 100.0;

// The 'n' value for cylindrical symmetry (n=1 for 2D axisymmetric cylindrical case)

let n = 1;

let original\_function = |r: f64| 10.0\*r;

let mut step\_count\_vec = Vec::<i32>::new();

let mut avg\_inaccuracy\_vec = Vec::<f64>::new();

let mut max\_inaccuracy\_vec = Vec::<f64>::new();

let mut inaccuracy\_in\_first\_half\_vec = Vec::<f64>::new();

let mut inaccuracy\_in\_second\_half\_vec = Vec::<f64>::new();

for step\_count in [2, 4, 8, 16, 32, 64, 128].iter() {

let points = generate\_points(LEFT, RIGHT, \*step\_count);

let splitter = IntervalSplitter::new(points.clone());

let coeff\_calculator\_v =

math::coeff\_calculator::first\_first\_calculator::FirstFirstCalculator::new(

splitter,

&kfunc,

&qfunc,

&ffunc,

y1,

y2,

n,

);

let (A, g) = math::coeff\_calculator::matrix\_building::build\_tridiagonal\_matrix(&coeff\_calculator\_v);

let calculated\_v = solver::solve(&A, &g);

let expected\_v: DVector<f64> = DVector::from\_vec(points.iter().map(|x| original\_function(\*x)).collect());

let accuracy = &calculated\_v - &expected\_v;

let avg\_inaccuracy = accuracy.fold(0.0, |acc, x| acc + x.abs()) / (accuracy.len() as f64 + 1.0);

step\_count\_vec.push(\*step\_count as i32);

avg\_inaccuracy\_vec.push(avg\_inaccuracy);

max\_inaccuracy\_vec.push(accuracy.iter().fold(0.0, |max, x| x.abs().max(max)));

let first\_half\_inaccuracy: f64 = accuracy

.iter()

.take(accuracy.len() / 2)

.fold(0.0, |acc, x| acc + x.abs());

let second\_half\_inaccuracy: f64 = accuracy

.iter()

.skip(accuracy.len() / 2)

.fold(0.0, |acc, x| acc + x.abs());

inaccuracy\_in\_first\_half\_vec.push(first\_half\_inaccuracy);

inaccuracy\_in\_second\_half\_vec.push(second\_half\_inaccuracy);

}

let mut rel\_inaccuracy: Vec<f64> = vec![];

for i in 1..avg\_inaccuracy\_vec.len() {

rel\_inaccuracy.push(avg\_inaccuracy\_vec[i-1] / avg\_inaccuracy\_vec[i]);

}

let digits\_after\_dot = 16;

println!("{}", "-".repeat(136));

println!(

"| {:>5} | {:width$.width$} | {:width$.width$} | {:width$.width$} | {:width$.width$} | {:width$.width$} |",

"steps",

"avg inaccuracy",

"max inaccuracy",

"first half inaccuracy",

"second half inaccuracy",

"rel inaccuracy",

width = digits\_after\_dot + 6

);

println!("{}", "-".repeat(136));

for i in 0..step\_count\_vec.len() {

println!(

"| {:>5} | {:width$e} | {:width$e} | {:width$e} | {:width$e} | {:width$.width$} |",

step\_count\_vec[i],

avg\_inaccuracy\_vec[i],

max\_inaccuracy\_vec[i],

inaccuracy\_in\_first\_half\_vec[i],

inaccuracy\_in\_second\_half\_vec[i],

if i > 0 { rel\_inaccuracy[i - 1] } else { 0.0 },

width = digits\_after\_dot + 6

);

}

println!("{}", "-".repeat(136));

}

fn exercise\_nonliner\_function\_base\_example() {

const LEFT: f64 = 1.0;

const RIGHT: f64 = 10.0;

// k(r) = 1, q(r) = 0, f(r) = 0 for a simple cylindrical heat conduction case

let kfunc = LambdaFunction::from(|r: f64| 2.0\* r.powf(2.0));

let qfunc = LambdaFunction::from(|r: f64| 5.0 \* r.powf(2.0));

let ffunc = LambdaFunction::from(|r: f64| 50.0\* r.powf(4.0) - 160.0\* r.powf(2.0));

// Boundary conditions: T(0) = 0 (center), T(1) = 1 (outer radius)

let y1 = 10.0;

let y2 = 1000.0;

// The 'n' value for cylindrical symmetry (n=1 for 2D axisymmetric cylindrical case)

let n = 1;

let original\_function = |r: f64| 10.0 \* r.powf(2.0);

let mut step\_count\_vec = Vec::<i32>::new();

let mut avg\_inaccuracy\_vec = Vec::<f64>::new();

let mut max\_inaccuracy\_vec = Vec::<f64>::new();

let mut inaccuracy\_in\_first\_half\_vec = Vec::<f64>::new();

let mut inaccuracy\_in\_second\_half\_vec = Vec::<f64>::new();

for step\_count in [2, 4, 8, 16, 32, 64, 128].iter() {

let points = generate\_points(LEFT, RIGHT, \*step\_count);

let splitter = IntervalSplitter::new(points.clone());

let coeff\_calculator\_v =

math::coeff\_calculator::first\_first\_calculator::FirstFirstCalculator::new(

splitter,

&kfunc,

&qfunc,

&ffunc,

y1,

y2,

n,

);

let (A, g) = math::coeff\_calculator::matrix\_building::build\_tridiagonal\_matrix(&coeff\_calculator\_v);

let calculated\_v = solver::solve(&A, &g);

let expected\_v: DVector<f64> = DVector::from\_vec(points.iter().map(|x| original\_function(\*x)).collect());

let accuracy = &calculated\_v - &expected\_v;

let avg\_inaccuracy = accuracy.fold(0.0, |acc, x| acc + x.abs()) / (accuracy.len() as f64 + 1.0);

step\_count\_vec.push(\*step\_count as i32);

avg\_inaccuracy\_vec.push(avg\_inaccuracy);

max\_inaccuracy\_vec.push(accuracy.iter().fold(0.0, |max, x| x.abs().max(max)));

let first\_half\_inaccuracy: f64 = accuracy

.iter()

.take(accuracy.len() / 2)

.fold(0.0, |acc, x| acc + x.abs());

let second\_half\_inaccuracy: f64 = accuracy

.iter()

.skip(accuracy.len() / 2)

.fold(0.0, |acc, x| acc + x.abs());

inaccuracy\_in\_first\_half\_vec.push(first\_half\_inaccuracy);

inaccuracy\_in\_second\_half\_vec.push(second\_half\_inaccuracy);

}

let mut rel\_inaccuracy: Vec<f64> = vec![];

for i in 1..avg\_inaccuracy\_vec.len() {

rel\_inaccuracy.push(avg\_inaccuracy\_vec[i-1] / avg\_inaccuracy\_vec[i]);

}

let digits\_after\_dot = 16;

println!("{}", "-".repeat(136));

println!(

"| {:>5} | {:width$.width$} | {:width$.width$} | {:width$.width$} | {:width$.width$} | {:width$.width$} |",

"steps",

"avg inaccuracy",

"max inaccuracy",

"first half inaccuracy",

"second half inaccuracy",

"rel inaccuracy",

width = digits\_after\_dot + 6

);

println!("{}", "-".repeat(136));

for i in 0..step\_count\_vec.len() {

println!(

"| {:>5} | {:width$e} | {:width$e} | {:width$e} | {:width$e} | {:width$.width$} |",

step\_count\_vec[i],

avg\_inaccuracy\_vec[i],

max\_inaccuracy\_vec[i],

inaccuracy\_in\_first\_half\_vec[i],

inaccuracy\_in\_second\_half\_vec[i],

if i > 0 { rel\_inaccuracy[i - 1] } else { 0.0 },

width = digits\_after\_dot + 6

);

}

println!("{}", "-".repeat(136));

}

fn main() {

// Set radial points (ranging from 0 to 1, simulating the radius of the cylinder)

let points = (1..=10).map(|i| (i as f64) / 10.0).collect::<Vec<f64>>();

let splitter = IntervalSplitter::new(points);

// k(r) = 1, q(r) = 0, f(r) = 0 for a simple cylindrical heat conduction case

let kfunc = LambdaFunction::from(|\_r| 1.0);

let qfunc = LambdaFunction::from(|\_r| 0.0);

let ffunc = LambdaFunction::from(|\_r| 0.0);

// Boundary conditions: T(0) = 0 (center), T(1) = 1 (outer radius)

let y1 = 0.0;

let y2 = 1.0;

// The 'n' value for cylindrical symmetry (n=1 for 2D axisymmetric cylindrical case)

let n = 1;

let coeff\_calculator\_v =

math::coeff\_calculator::first\_first\_calculator::FirstFirstCalculator::new(

splitter,

&kfunc,

&qfunc,

&ffunc,

y1,

y2,

n,

);

// Build the matrix and right-hand side vector

let (A, g) = math::coeff\_calculator::matrix\_building::build\_tridiagonal\_matrix(&coeff\_calculator\_v);

println!("Matrix A:\n{}", A);

println!("Vector g:\n{}", g);

// Solve the system to get the temperature distribution

let v = solver::solve(&A, &g);

println!("Solution vector v (temperature distribution):\n{}", v);

// exercise\_accuracy\_base\_example();

// exercise\_liner\_function\_base\_example();

exercise\_nonliner\_function\_base\_example();

}

## coeff\_calculator.rs

pub trait CoeffCalculator<Number> {

fn calc\_a(&self, i: usize) -> Number;

fn calc\_b(&self, i: usize) -> Number;

fn calc\_c(&self, i: usize) -> Number;

fn calc\_g(&self, i: usize) -> Number;

fn size(&self) -> usize;

}

pub struct LambdaFunction<Number, Func: Fn(Number) -> Number> {

ffunc: Func,

phatnom: std::marker::PhantomData<Number>,

}

impl<Number, Func: Fn(Number) -> Number> Function<Number> for LambdaFunction<Number, Func> {

fn calc(&self, x: Number) -> Number {

(self.ffunc)(x)

}

}

impl<Number, Func: Fn(Number) -> Number> std::convert::From<Func> for LambdaFunction<Number, Func> {

fn from(ffunc: Func) -> Self {

LambdaFunction {

ffunc,

phatnom: std::marker::PhantomData,

}

}

}

pub trait Function<Number> {

fn calc(&self, x: Number) -> Number;

}

/// This module will contain calculator that works

/// when we have \*\*1st condition\*\* at the left and \*\*1st condition\*\* at the right

pub mod first\_first\_calculator {

use std::marker::PhantomData;

use super::Function;

use crate::math::coeff\_calculator::CoeffCalculator;

use crate::math::stepping::{NumberTrait, Stepping};

pub struct FirstFirstCalculator<'a,

SteppingObject: Stepping<Number>,

KFunctionType: Function<Number>,

QFunctionType: Function<Number>,

FunctionType: Function<Number>,

Number: NumberTrait,

> {

stepping: SteppingObject,

kfunc: &'a KFunctionType,

qfunc: &'a QFunctionType,

ffunc: &'a FunctionType,

y1: Number,

y2: Number,

n: u16,

}

impl<

'a,

SteppingObject: Stepping<Number>,

KFunctionType: Function<Number>,

QFunctionType: Function<Number>,

FunctionType: Function<Number>,

Number: NumberTrait,

> FirstFirstCalculator<'a, SteppingObject, KFunctionType, QFunctionType, FunctionType, Number>

{

pub fn new(

stepping: SteppingObject,

kfunc: &'a KFunctionType,

qfunc: &'a QFunctionType,

ffunc: &'a FunctionType,

y1: Number,

y2: Number,

n: u16,

) -> FirstFirstCalculator<'a, SteppingObject, KFunctionType, QFunctionType, FunctionType, Number>

{

FirstFirstCalculator {

stepping,

kfunc,

qfunc,

ffunc,

y1,

y2,

n,

}

}

}

impl<

'a,

SteppingObject: Stepping<Number>,

KFunctionType: Function<Number>,

QFunctionType: Function<Number>,

FunctionType: Function<Number>,

Number: NumberTrait,

> CoeffCalculator<Number>

for FirstFirstCalculator<'a, SteppingObject, KFunctionType, QFunctionType, FunctionType, Number>

{

fn calc\_a(&self, i: usize) -> Number {

if i == 0 {

panic!("i == 0")

} else if i < self.stepping.points().len() - 1 {

Number::from(-1)

\* self.stepping.middle\_point(i - 1).pow(self.n)

\* self.kfunc.calc(self.stepping.middle\_point(i - 1))

/ self.stepping.step(i)

} else if i == self.stepping.points().len() - 1 {

Number::from(0)

} else {

panic!("i > self.stepping.steps\_count()")

}

}

fn calc\_b(&self, i: usize) -> Number {

if i == 0 {

Number::from(0)

} else if i < self.stepping.points().len() - 1 {

Number::from(-1)

\* self.stepping.middle\_point(i).pow(self.n)

\* self.kfunc.calc(self.stepping.middle\_point(i))

/ self.stepping.step(i + 1)

} else {

panic!("i >= self.stepping.points().len()")

}

}

fn calc\_c(&self, i: usize) -> Number {

if i == 0 {

Number::from(1)

} else if i < self.stepping.points().len() - 1 {

self.stepping.middle\_point(i - 1).pow(self.n)

\* self.kfunc.calc(self.stepping.middle\_point(i - 1))

/ self.stepping.step(i)

+ self.stepping.middle\_point(i).pow(self.n)

\* self.kfunc.calc(self.stepping.middle\_point(i))

/ self.stepping.step(i + 1)

+ self.stepping.cross\_step(i)

\* self.stepping.point(i).pow(self.n)

\* self.qfunc.calc(self.stepping.point(i))

} else if i == self.stepping.points().len() - 1 {

Number::from(1)

} else {

panic!("i > self.stepping.points().len()")

}

}

fn calc\_g(&self, i: usize) -> Number {

if i == 0 {

self.y1

} else if i < self.stepping.points().len() - 1 {

self.stepping.cross\_step(i)

\* self.stepping.point(i).pow(self.n)

\* self.ffunc.calc(self.stepping.point(i))

} else if i == self.stepping.points().len() - 1 {

self.y2

} else {

panic!("i >= self.stepping.points().len()")

}

}

fn size(&self) -> usize {

self.stepping.points().len()

}

}

}

pub mod matrix\_building {

use crate::CoeffCalculator;

use nalgebra::{DMatrix, DVector};

/// Builds the tridiagonal matrix and the right-hand side vector

/// using the provided coefficient calculator.

pub fn build\_tridiagonal\_matrix<Number, Calculator>(

calculator: &Calculator,

) -> (DMatrix<Number>, DVector<Number>)

where

Number: nalgebra::RealField, // Ensures we can work with nalgebra numbers

Calculator: CoeffCalculator<Number>,

{

let n = calculator.size();

// Create an NxN matrix initialized to zeros

let mut matrix = DMatrix::zeros(n, n);

// Create a vector for the right-hand side of the equation

let mut rhs\_vector = DVector::zeros(n);

// Fill the matrix with values from the coefficient calculator

for i in 0..n {

if i > 0 {

matrix[(i, i - 1)] = calculator.calc\_a(i); // Sub-diagonal (below the main diagonal)

}

matrix[(i, i)] = calculator.calc\_c(i); // Main diagonal

if i < n - 1 {

matrix[(i, i + 1)] = calculator.calc\_b(i); // Super-diagonal (above the main diagonal)

}

// Fill the right-hand side vector (the g vector)

rhs\_vector[i] = calculator.calc\_g(i);

}

(matrix, rhs\_vector)

}

}

## solver.rs

use core::panic;

use nalgebra::DMatrix;

use nalgebra::DVector;

/// This type allow us to specify the exact type of the number we are using in the solver

#[allow(unused)]

pub type Number = f64;

/// This function checks if the matrix is tri-diagonally dominant.

///

/// Dominant tridiagonal matrices are the ones that have only

/// non-zero elements on the main diagonal and upper and lower diagonals

#[allow(unused)]

fn check\_if\_three\_diagonals(a: &DMatrix<Number>) -> () {

if a.nrows() != a.ncols() {

panic!("The matrix is not square");

}

for row in 0..a.nrows() {

for col in 0..a.ncols() {

if !(row.abs\_diff(col) <= 1) && (a[(row, col)] != 0.0) {

panic!("Matrix is not diagonally dominant");

}

if row > 0 && row < a.nrows() - 1 && row == col && (a[(row-1, col)] + a[(row+1, col)] > a[(row, col)]) {

panic!("Matrix is not diagonally dominant");

}

if row == 0 && row < a.nrows() - 1 && row == col && (a[(row+1, col)] > a[(row, col)]) {

panic!("Matrix is not diagonally dominant");

}

}

}

}

/// This function calculates v-coefficients

#[allow(unused)]

fn calculate\_v\_coefficients(a: &DMatrix<Number>) -> DVector<Number> {

let mut v\_arr = DVector::<Number>::from\_vec(vec![0.0; a.nrows()]);

v\_arr[0] = a[(0, 1)] / (-a[(0, 0)]);

for i in 1..a.nrows() - 1 {

v\_arr[i] = a[(i, i + 1)] / (-a[(i, i)] - a[(i, i - 1)] \* v\_arr[i - 1]);

}

v\_arr[a.nrows() - 1] = 0.0;

return v\_arr;

}

/// This function calculates u-coefficients.

/// It depends on v-coefficients

#[allow(unused)]

fn calculate\_u\_coefficients(

a: &DMatrix<Number>,

b: &DVector<Number>,

v\_arr: &DVector<Number>,

) -> DVector<Number> {

let mut u\_arr = DVector::<Number>::from\_vec(vec![0.0; a.nrows()]);

u\_arr[0] = -b[0] / (-a[(0, 0)]);

for i in 1..a.nrows() - 1 {

u\_arr[i] =

(a[(i, i - 1)] \* u\_arr[i - 1] - b[i]) / (-a[(i, i)] - a[(i, i - 1)] \* v\_arr[i - 1]);

}

u\_arr[a.nrows() - 1] = (a[(a.nrows() - 1, a.nrows() - 2)] \* u\_arr[a.nrows() - 2]

- b[a.nrows() - 1])

/ (-a[(a.nrows() - 1, a.nrows() - 1)]

- a[(a.nrows() - 1, a.nrows() - 2)] \* v\_arr[a.nrows() - 2]);

return u\_arr;

}

/// This function checks if two vectors are almost equal

///

/// We need it because nalgebra::DVector does not have almost\_equal function.

/// Generally 0.99999999999 and 1.00000000001 are not equal completely, so we need to use almost\_equal

#[allow(unused)]

pub fn vectors\_almost\_equal(

vec1: &DVector<Number>,

vec2: &DVector<Number>,

epsilon: Number,

) -> bool {

if vec1.len() != vec2.len() {

return false;

}

for i in 0..vec1.len() {

if (vec1[i] - vec2[i]).abs() > epsilon {

return false;

}

}

true

}

/// This function solves the system of linear equations

/// Ax = b

/// It uses tridiagonal matrix method

/// Make sure the matrix is tri-diagonally dominant

#[allow(unused)]

pub fn solve(a: &DMatrix<Number>, b: &DVector<Number>) -> DVector<Number> {

check\_if\_three\_diagonals(a);

let mut v\_arr = calculate\_v\_coefficients(a);

let mut u\_arr = calculate\_u\_coefficients(a, b, &v\_arr);

let mut x\_arr = DVector::<Number>::zeros(a.ncols());

x\_arr[a.nrows() - 1] = u\_arr[a.nrows() - 1];

for i in (0..a.nrows() - 1).rev() {

x\_arr[i] = u\_arr[i] + v\_arr[i] \* x\_arr[i + 1];

}

return x\_arr;

}

/// Here are just some tests

mod tests {

use super::\*;

#[allow(unused\_imports)]

use nalgebra::{DMatrix, RowDVector};

/// This test checks if the matrix is tri-diagonally dominant

#[test]

#[allow(non\_snake\_case)]

fn check\_if\_three\_diagonals\_\_correct\_matrix() {

let example\_matrix = DMatrix::<Number>::from\_rows(&[

RowDVector::from\_vec(vec![5., 1., 0., 0., 0.]),

RowDVector::from\_vec(vec![1., 5., 1., 0., 0.]),

RowDVector::from\_vec(vec![0., 1., 5., 1., 0.]),

RowDVector::from\_vec(vec![0., 0., 1., 5., 1.]),

RowDVector::from\_vec(vec![0., 0., 0., 1., 5.]),

]);

check\_if\_three\_diagonals(&example\_matrix);

}

/// This test checks if the matrix is not tri-diagonally dominant and should panic because matrix is not even square

#[test]

#[should\_panic]

#[allow(non\_snake\_case)]

fn check\_if\_three\_diagonals\_\_incorrect\_matrix\_1() {

let example\_matrix = DMatrix::<Number>::from\_rows(&[

RowDVector::from\_vec(vec![1., 2., 3., 0., 0.]),

RowDVector::from\_vec(vec![0., 1., 2., 0., 0.]),

RowDVector::from\_vec(vec![0., 0., 1., 2., 0.]),

RowDVector::from\_vec(vec![0., 0., 0., 1., 2.]),

]);

check\_if\_three\_diagonals(&example\_matrix);

}

/// This test checks if the matrix is not tri-diagonally dominant and should panic

#[test]

#[should\_panic]

#[allow(non\_snake\_case)]

fn check\_if\_three\_diagonals\_\_incorrect\_matrix\_2() {

let example\_matrix = DMatrix::<Number>::from\_rows(&[

RowDVector::from\_vec(vec![1., 2., 0., 0., 0.]),

RowDVector::from\_vec(vec![3., 1., 2., 0., 0.]),

RowDVector::from\_vec(vec![0., 0., 1., 2., 0.]),

RowDVector::from\_vec(vec![0., 1., 3., 1., 2.]),

RowDVector::from\_vec(vec![0., 0., 0., 1., 2.]),

]);

check\_if\_three\_diagonals(&example\_matrix);

}

/// This test checks if the matrix is not tri-diagonally dominant and should panic

#[test]

#[should\_panic]

#[allow(non\_snake\_case)]

fn check\_if\_three\_diagonals\_\_incorrect\_matrix\_3() {

let example\_matrix = DMatrix::<Number>::from\_rows(&[

RowDVector::from\_vec(vec![1., 2., 0., 0., 0.]),

RowDVector::from\_vec(vec![3., 1., 2., 1., 0.]),

RowDVector::from\_vec(vec![0., 1., 2., 0., 0.]),

RowDVector::from\_vec(vec![0., 0., 1., 2., 1.]),

RowDVector::from\_vec(vec![0., 0., 0., 1., 2.]),

]);

check\_if\_three\_diagonals(&example\_matrix);

}

#[test]

#[allow(non\_snake\_case)]

fn check\_if\_three\_diagonals\_\_correct\_matrix\_2() {

let example\_matrix = DMatrix::<Number>::from\_rows(&[

RowDVector::from\_vec(vec![4., 1., 0., 0., 0.]),

RowDVector::from\_vec(vec![2., 4., 1., 0., 0.]),

RowDVector::from\_vec(vec![0., 2., 4., 1., 0.]),

RowDVector::from\_vec(vec![0., 0., 2., 4., 1.]),

RowDVector::from\_vec(vec![0., 0., 0., 2., 4.]),

]);

check\_if\_three\_diagonals(&example\_matrix);

}

#[test]

#[should\_panic]

#[allow(non\_snake\_case)]

fn check\_if\_three\_diagonals\_\_incorrect\_matrix\_4() {

let example\_matrix = DMatrix::<Number>::from\_rows(&[

RowDVector::from\_vec(vec![6., 2., 0., 0., -1.]),

RowDVector::from\_vec(vec![3., 1., 2., 0., 0.]),

RowDVector::from\_vec(vec![0., 1., 2., 0., 0.]),

RowDVector::from\_vec(vec![0., 0., 1., 2., 0.]),

RowDVector::from\_vec(vec![0., 0., 1., 1., 2.]),

]);

check\_if\_three\_diagonals(&example\_matrix);

}

/// Here are some tests directly for the solver

mod solver {

use super::\*;

use rand::Rng;

/// This function calculates b to prepare data for text

#[allow(unused)]

fn calculate\_b(a: &DMatrix<Number>, x\_arr: &DVector<Number>) -> DVector<Number> {

return a \* x\_arr;

}

/// Here is we print the equation that about to be solved

#[allow(unused)]

fn print\_equation(a: &DMatrix<Number>, x: &DVector<Number>, b: &DVector<Number>) {

print!("A:{}", a);

println!("\*");

print!("x:{}", x);

println!("=");

print!("b:{}", b);

}

/// This test is used to avoid repeating of code and call the solver with different input parameters

#[allow(unused)]

fn base\_solve\_test(a: &DMatrix<Number>, example\_x: &DVector<Number>) {

let example\_b = calculate\_b(&a, &example\_x);

print\_equation(&a, &example\_x, &example\_b);

let solve\_x = solve(&a, &example\_b);

println!("Calculated X: {}", solve\_x.to\_string());

println!("Expected X: {}", example\_x.to\_string());

println!("----------------------------");

assert!(vectors\_almost\_equal(&solve\_x, example\_x, 0.001));

}

#[test]

#[allow(non\_snake\_case)]

fn solve\_\_correct\_matrix\_1() {

let a = DMatrix::<Number>::from\_rows(&[

RowDVector::from\_vec(vec![5., 1., 0., 0., 0.]),

RowDVector::from\_vec(vec![1., 5., 1., 0., 0.]),

RowDVector::from\_vec(vec![0., 1., 5., 1., 0.]),

RowDVector::from\_vec(vec![0., 0., 1., 5., 1.]),

RowDVector::from\_vec(vec![0., 0., 0., 1., 5.]),

]);

let example\_x = DVector::<Number>::from\_vec(vec![1., 3., 2., 5., 4.]);

base\_solve\_test(&a, &example\_x);

}

#[test]

#[allow(non\_snake\_case)]

fn solve\_\_correct\_matrix\_2() {

let a = DMatrix::<Number>::from\_rows(&[

RowDVector::from\_vec(vec![4., 2., 0., 0.]),

RowDVector::from\_vec(vec![1., 3., 1., 0.]),

RowDVector::from\_vec(vec![0., 1., 4., 2.]),

RowDVector::from\_vec(vec![0., 0., 1., 4.]),

]);

let example\_x = DVector::<Number>::from\_vec(vec![1., 2., 3., 4.]);

base\_solve\_test(&a, &example\_x);

}

#[test]

#[allow(non\_snake\_case)]

fn solve\_\_correct\_matrix\_3() {

let a = DMatrix::<Number>::from\_rows(&[

RowDVector::from\_vec(vec![3., 1., 0., 0., 0.]),

RowDVector::from\_vec(vec![1., 3., 1., 0., 0.]),

RowDVector::from\_vec(vec![0., 1., 3., 1., 0.]),

RowDVector::from\_vec(vec![0., 0., 1., 4., 1.]),

RowDVector::from\_vec(vec![0., 0., 0., 1., 2.]),

]);

let example\_x = DVector::<Number>::from\_vec(vec![5., 4., 3., 2., 1.]);

base\_solve\_test(&a, &example\_x);

}

#[test]

#[allow(non\_snake\_case)]

fn solve\_\_correct\_matrix\_4() {

let a = DMatrix::<Number>::from\_rows(&[

RowDVector::from\_vec(vec![10., 2., 0.]),

RowDVector::from\_vec(vec![3., 8., 1.]),

RowDVector::from\_vec(vec![0., 4., 7.]),

]);

let example\_x = DVector::<Number>::from\_vec(vec![2., 1., 3.]);

base\_solve\_test(&a, &example\_x);

}

#[test]

#[allow(non\_snake\_case)]

fn solve\_\_correct\_matrix\_5() {

let a = DMatrix::<Number>::from\_rows(&[

RowDVector::from\_vec(vec![6., 1., 0.]),

RowDVector::from\_vec(vec![1., 6., 2.]),

RowDVector::from\_vec(vec![0., 2., 5.]),

]);

let example\_x = DVector::<Number>::from\_vec(vec![1., 1., 1.]);

base\_solve\_test(&a, &example\_x);

}

/// Here we run solve tests with random matrices

#[test]

fn solve\_random\_matrix\_multiple\_tests() {

println!("----------------------------");

println!("Running random matrix tests...");

let mut rng = rand::thread\_rng();

// Define the number of test runs

let test\_runs = 10; // You can adjust the number of iterations for larger tests

for \_ in 0..test\_runs {

let n = rng.gen\_range(2..10); // Matrix size, adjust for larger matrices

// Generate a random tridiagonal matrix

let mut matrix = DMatrix::<Number>::zeros(n, n);

for i in 0..n {

// Diagonal element

matrix[(i, i)] = rng.gen\_range(2.1..10.0);

// Subdiagonal element (if applicable)

if i > 0 {

matrix[(i, i - 1)] = rng.gen\_range(0.1..1.0);

}

// Superdiagonal element (if applicable)

if i < n - 1 {

matrix[(i, i + 1)] = rng.gen\_range(0.1..1.0);

}

}

// Generate a random x vector

let example\_x = DVector::<Number>::from\_fn(n, |\_, \_| rng.gen\_range(1.0..10.0));

// Call the base\_solve\_test function to perform the test

base\_solve\_test(&matrix, &example\_x);

}

}

}

}

## stepping.rs

pub trait NumberTrait:

std::ops::Div<Self, Output = Self>

+ std::ops::Sub<Output = Self>

+ std::ops::Add<Output = Self>

+ std::ops::Mul<Output = Self>

+ num\_traits::Pow<u16, Output = Self>

+ Sized

+ Copy

+ std::convert::From<i32>

+ std::convert::From<u16>

{

}

impl NumberTrait for i32 {}

impl NumberTrait for f64 {}

pub trait Stepping<Number>

where

Number: NumberTrait,

{

/// \*\*NOTE:\*\* `i > 0` and `i <= self.steps\_count()`

fn step(&self, i: usize) -> Number;

/// \*\*NOTE:\*\* `i >= 0` and `i <= self.steps\_count()`

fn cross\_step(&self, i: usize) -> Number {

assert!(i < self.points().len());

if i == 0 {

self.step(1) / Number::from(2)

} else if i == self.points().len() - 1 {

self.step(i) / Number::from(2)

} else {

(self.step(i + 1) - self.step(i)) / Number::from(2)

}

}

fn points(&self) -> &Vec<Number>;

fn point(&self, i: usize) -> Number;

/// Returns the middle point between `i` and `i+1`

fn middle\_point(&self, i: usize) -> Number;

}

pub struct IntervalSplitter<Number> {

points: Vec<Number>,

/// points.size() - 1

steps: Vec<Number>,

/// points.size()

cross\_steps: Vec<Number>,

}

impl<Number> IntervalSplitter<Number>

where

Number: NumberTrait,

{

pub fn new(points: Vec<Number>) -> IntervalSplitter<Number> {

let mut steps: Vec<Number> = vec![];

for i in 1..points.len() {

steps.push(points[i] - points[i - 1]);

}

let mut cross\_steps: Vec<Number> = vec![];

for i in 0..points.len() {

if i == 0 {

cross\_steps.push(steps[i] / Number::from(2));

} else if i == points.len() - 1 {

cross\_steps.push(steps[i - 1] / Number::from(2));

} else {

cross\_steps.push((steps[i - 1] + steps[i]) / Number::from(2));

}

}

IntervalSplitter {

points,

steps,

cross\_steps,

}

}

}

impl<Number> Stepping<Number> for IntervalSplitter<Number>

where

Number: NumberTrait,

{

fn step(&self, i: usize) -> Number {

self.steps[i - 1]

}

fn cross\_step(&self, i: usize) -> Number {

self.cross\_steps[i]

}

fn points(&self) -> &Vec<Number> {

&self.points

}

fn point(&self, i: usize) -> Number {

self.points[i]

}

fn middle\_point(&self, i: usize) -> Number {

assert!(i < self.points.len() - 1);

(self.points[i] + self.points[i + 1]) / Number::from(2)

}

}