Министерство образования и науки РФ
Санкт-Петербургский Политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности
Высшая школа программной инженерии

Работа №1

по дисциплине

«Разработка программного обеспечения для моделирования физических процессов»

Вариант СР3

Выполнил
студент гр. 5130904/10101
Абраамян А. М.

Преподаватель
Воскобойников С.П.

Санкт-Петербург
2024

Оглавление

Постановка задачи....3
Аппроксимация....3
Разностная схема....5
Метод прогонки....5
Вычисление погрешности....7
Оценка работы алгоритма на различных примерах....9
  Пример 1....9
  Пример 2....10
  Пример 3....10
Код....11

Постановка задачи

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

$$-\left[\frac{1}{r}\frac{d}{dr}\left(rk(r)\frac{du}{dr}\right)-q(r)u\right]=f(r),\quad r\in[R_L,R_R],\ R_L>0,$$
$$0<c_1\le k(r)\le c_2,\quad 0\le q(r)$$

с граничными условиями, определяемыми вариантом задания

3. $u_{r=R_L}=\nu_1$, $-k\frac{du}{dr}\Big|_{r=R_R}=\chi_2 u|_{r=R_R}-\nu_2$, $\chi_2\ge 0$,

Аппроксимация

Запишем исходное уравнение:

Интегро-интерполяционный метод (метод баланса)

Аппроксимация уравнения

$$-\left[\frac{1}{r^n}\frac{d}{dr}\left(r^n k(r)\frac{du(r)}{dr}\right)-q(r)u(r)\right]=f(r),\qquad 0<C_1\le k(r)\le C_2,\qquad r\in[0,R]$$

Проинтегриурем обе части от $r_{(i-1/2)}$ до $r_{(i+1/2)}$

$$-\int_{r_{i-1/2}}^{r_{i+1/2}}\left[\frac{d}{dr}\left(r^n k(r)\frac{du(r)}{dr}\right)-r^n q(r)u(r)\right]dr=\int_{r_{i-1/2}}^{r_{i+1/2}}r^n f(r)dr,\qquad i=1,2,\ldots,N-1$$

Дальше раскроем скобки и применим формулу левой разностной производной.

$$-\left[r^n k(r)\frac{du(r)}{dr}\Big|_{r=r_{i+1/2}}-r^n k(r)\frac{du(r)}{dr}\Big|_{r=r_{i-1/2}}-\int_{r_{i-1/2}}^{r_{i+1/2}}r^n q(r)u(r)dr\right]=\int_{r_{i-1/2}}^{r_{i+1/2}}r^n f(r)dr,$$

$$\frac{du(r)}{dr}\Big|_{r=r_{i-1/2}}\approx\frac{u_i-u_{i-1}}{2\frac{h_i}{2}}=\frac{u_i-u_{i-1}}{h_i}$$

Выразив $\nu_{(i+1)},\nu_{(i)},\nu_{(i-1)}$, - получим необходимые формулы для коэффициентов a, b и c при i = 1..N-1. Значения коэффициентов для i = N и для i = 0 будут получены с помощью граничных условий далее.

$$-\left[r^n_{i+1/2}k_{i+1/2}\frac{v_{i+1}-v_i}{h_{i+1}}-r^n_{i-1/2}k_{i-1/2}\frac{v_i-v_{i-1}}{h_i}-\hbar_i r_i^n q_i v_i\right]=\hbar_i r_i^n f_i,\quad i=1,2,\ldots,N-1$$

Воспользуемся знанием что у нас правое граничное условие третьего рода.

$$Краевое\ \ условие\ третьего\ рода:\ -k(r)\frac{du(r)}{dr}\Big|_{r=R}=\chi u(r)\Big|_{r=R}-\gamma,\qquad \chi>0$$

Выполнив подстановку в предыдущее уравнение получаем следующее:

$$-\left[-r_i^n(\chi v_i|-\gamma)-r^n_{i-1/2}k_{i-1/2}\frac{v_i-v_{i-1}}{h_i}-\hbar_i r_i^n q_i v_i\right]=\hbar_i r_i^n f_i,\quad i=N$$

Разностная схема

$$a_i=-r^n_{i-1/2}\frac{k_{i-1/2}}{h_i},\quad c_i=r^n_{i-1/2}\frac{k_{i-1/2}}{h_i}+r^n_{i+1/2}\frac{k_{i+1/2}}{h_{i+1}}+\hbar_i r_i^n q_i,\qquad b_i=-r^n_{i+1/2}\frac{k_{i+1/2}}{h_{i+1}},\quad g_i=\hbar_i r_i^n f_i,\quad i=1,2,\ldots,N-1$$

$$a_i=-r^n_{i-1/2}\frac{k_{i-1/2}}{h_i},\quad c_i=r^n_{i-1/2}\frac{k_{i-1/2}}{h_i}+\hbar_i r_i^n q_i+r_i^n\chi,\qquad g_i=\hbar r_i^n f_i+r_i^n\gamma,\quad i=N$$

,

$$Av=g,\quad A-(N+1)\times(N+1),\ v,g\in R^{(N+1)}$$

$$A=\begin{bmatrix} c_0 & b_0 & & & & & \\ a_1 & c_1 & b_1 & & & & \\ & \cdot & \cdot & \cdot & & & \\ & & \cdot & \cdot & \cdot & & \\ & & & \cdot & \cdot & \cdot & \\ & & & & a_{N-1} & c_{N-1} & b_{N-1} \\ & & & & & a_N & c_N \end{bmatrix},\qquad v=\begin{bmatrix} v_0 \\ v_1 \\ \cdot \\ \cdot \\ \cdot \\ v_{N-1} \\ v_N \end{bmatrix},\qquad g=\begin{bmatrix} g_0 \\ g_1 \\ \cdot \\ \cdot \\ \cdot \\ g_{N-1} \\ g_N \end{bmatrix}$$

$$A=A^T$$

Из приведенной выше схемы для моего варианта задания подойдёт всё кроме **i=0.** Для i=0 подойдут следующие коэффициенты – b = 0, c = 1.

Метод прогонки

Прямой ход:

Определение прогоночных коэффициентов:

$$v_{[i]} = \frac{a_{[i][i+1]}}{-a_{[i][i]} - a_{[i][i-1]} \cdot v_{[i-1]}} \text{ и } u_{[i]} = \frac{a_{[i][i-1]} \cdot u_{[i-1]} - b_{[i]}}{-a_{[i][i]} - a_{[i][i-1]} \cdot v_{[i-1]}}$$

Уточнение для первой строки матрицы:

$$v_{[0]} = \frac{a_{[0][1]}}{-a_{[0][0]}} \text{ и } u_{[0]} = \frac{-b_{[0]}}{-a_{[0][0]}}$$

Уточнение для последней строки матрицы:

$$v_{[n-1]} = 0 \text{ и } u_{[n-1]} = \frac{a_{[n-1][n-2]} \cdot u_{[n-2]} - b_{[n-1]}}{-a_{[n-1][n-1]} - a_{[n-1][n-2]} \cdot v_{[n-2]}}$$

Обратный ход:

Вычисление корней начинается с конца.

Конечный x_n корень вычисляется по формуле $x_{[n-1]} = u_{[n-1]}$.

Остальные корни считаются с помощью рекуррентного соотношения:

$$x_{[i-1]} = v_{[i-1]} \cdot x_{[i]} + u_{[i-1]}$$

Вывод результатов в виде вектора решений: $X = \begin{bmatrix} x_{[0]} \\ x_{[1]} \\ \dots \\ x_{[n-1]} \end{bmatrix}$

Проверить работоспособность метода помогут тесты. Ниже приведен один из многочисленных тестов доказывающих правильность работы программы и алгоритма в целом

```
successes:

---- math::solver::tests::solver::solve__correct_matrix_3 stdout ----
A:
⎡ 3 1 0 0 0 ⎤
⎢ 1 3 1 0 0 ⎥
⎢ 0 1 3 1 0 ⎥
⎢ 0 0 1 4 1 ⎥
⎣ 0 0 0 1 2 ⎦

*
x:
⎡ 5 ⎤
⎢ 4 ⎥
⎢ 3 ⎥
⎢ 2 ⎥
⎣ 1 ⎦

=
b:
⎡ 19 ⎤
⎢ 20 ⎥
⎢ 15 ⎥
⎢ 12 ⎥
⎣  4 ⎦

Calculated X:
⎡                 5 ⎤
⎢ 4.000000000000001 ⎥
⎢                 3 ⎥
⎢                 2 ⎥
⎣                 1 ⎦
```

```
Expected X:
⎡ 5 ⎤
⎢ 4 ⎥
⎢ 3 ⎥
⎢ 2 ⎥
⎣ 1 ⎦
```

Вычисление погрешности

Вклад от погрешности решения системы алгебраических уравнений

$$\|z\| \le \|A^{-1}\|\|r\| = \|A\|\|A^{-1}\|\frac{\|r\|}{\|A\|} \qquad \|A\| \ge \frac{\|g\|}{\|v\|}$$

$$\|z\| \le \|A\|\|A^{-1}\|\frac{\|r\|}{\|g\|}\|v\|$$

$$cond(A) = \|A\|\|A^{-1}\| \qquad \frac{\|r\|}{\|g\|} \sim \varepsilon_M$$

$$\|z\| \le cond(A)\frac{\|r\|}{\|g\|}\|v\| \sim cond(A)\varepsilon_M\|v\|$$

$$\varepsilon_M \sim \begin{cases} 10^{-7}, & 4\ bytes \\ 10^{-16}, & 8\ bytes \end{cases}$$

Анализируя погрешность важно понимать что она складывается из двух частей. Первая – погрешность вычислительная. Чем хуже обусловлена у нас матрица – тем быстрее она будет расти. На современных вычислительный машинах при хорошей обусловленности погрешность будет примерно 1е-16.

Разложение невязки для уравнения в цилиндрической и сферической системе координат

$$-\left[\frac{1}{r^n}\frac{d}{dr}\left(r^n k\frac{du}{dr}\right)-qu(r)\right]=f,\qquad -\left[\frac{d}{dr}\left(r^n k\frac{du}{dr}\right)-r^n qu(r)\right]=r^n f,$$

$$\tilde{k}=r^n k,\qquad \tilde{q}=r^n q,\qquad \tilde{f}=r^n f,\qquad -\left[\frac{d}{dr}\left(\tilde{k}\frac{du}{dr}\right)-\tilde{q}u(r)\right]=\tilde{f},$$

$$\xi_i=h\left[\tilde{f}+\frac{d}{dr}\left(\tilde{k}\frac{du}{dr}\right)-\tilde{q}u\right]_{r=r_i}+h^3\left[\frac{1}{12}\tilde{k}\frac{d^4u}{dr^4}+\frac{1}{6}\frac{d\tilde{k}}{dr}\frac{d^3u}{dr^3}+\frac{1}{8}\frac{d^2\tilde{k}}{dr^2}\frac{d^2u}{dr^2}+\frac{1}{24}\frac{d^3\tilde{k}}{dr^3}\frac{du}{dr}\right]_{r=r_i}+O(h^4)$$

$$\xi_i=h\left[r^n f+\frac{d}{dr}\left(r^n k\frac{du}{dr}\right)-r^n qu\right]_{r=r_i}+h^3\left[\frac{1}{12}r^n k\frac{d^4u}{dr^4}+\frac{1}{6}\frac{d\left(r^n k\right)}{dr}\frac{d^3u}{dr^3}+\frac{1}{8}\frac{d^2\left(r^n k\right)}{dr^2}\frac{d^2u}{dr^2}+\frac{1}{24}\frac{d^3\left(r^n k\right)}{dr^3}\frac{du}{dr}\right]_{r=r_i}+O(h^4)$$

Разложение невязки для вий 3-его рода в цилиндрической и сферической системе координат

$$-k(r)\frac{du(r)}{dr}\bigg|_{r=R}=\chi u(r)\big|_{r=R}-\gamma,\qquad \chi>0\qquad r^n\big|_{r=R},\qquad -r^n k(r)\frac{du(r)}{dr}\bigg|_{r=R}=r^n\chi u(r)\big|_{r=R}-r^n\big|_{r=R}\gamma,$$

$$\tilde{k}=r^n k(r),\qquad \tilde{\chi}=r^n\chi,\qquad \tilde{\gamma}=r^n\gamma,\qquad -\tilde{k}(r)\frac{du(r)}{dr}\bigg|_{r=R}=\tilde{\chi}u(r)\big|_{r=R}-\tilde{\gamma},$$

$$\xi_i=h^0\left[-\left(\tilde{\chi}u-\tilde{\gamma}\right)-\tilde{k}\frac{du}{dx}\right]_{r=r_i}+h\left[\tilde{f}-\frac{d}{dr}\left(\tilde{k}\frac{du}{dr}\right)-\tilde{q}u(r)\right]_{r=r_i}-$$

$$-h^2\left[\frac{1}{6}\tilde{k}\frac{d^3u}{dr^3}+\frac{1}{4}\frac{d\tilde{k}}{dr}\frac{d^2u}{dr^2}+\frac{1}{8}\frac{d^2\tilde{k}}{dr^2}\frac{du}{dr}\right]_{r=r_i}+O(h^3),\quad i=N$$

$$\xi_i=h^0\left[-r^n\left(\chi u-\gamma\right)-r^n k\frac{du}{dr}\right]_{r=r_i}+h\left[r^n f-\frac{d}{dr}\left(r^n k\frac{du}{dr}\right)-\tilde{q}u(r)\right]_{r=r_i}-$$

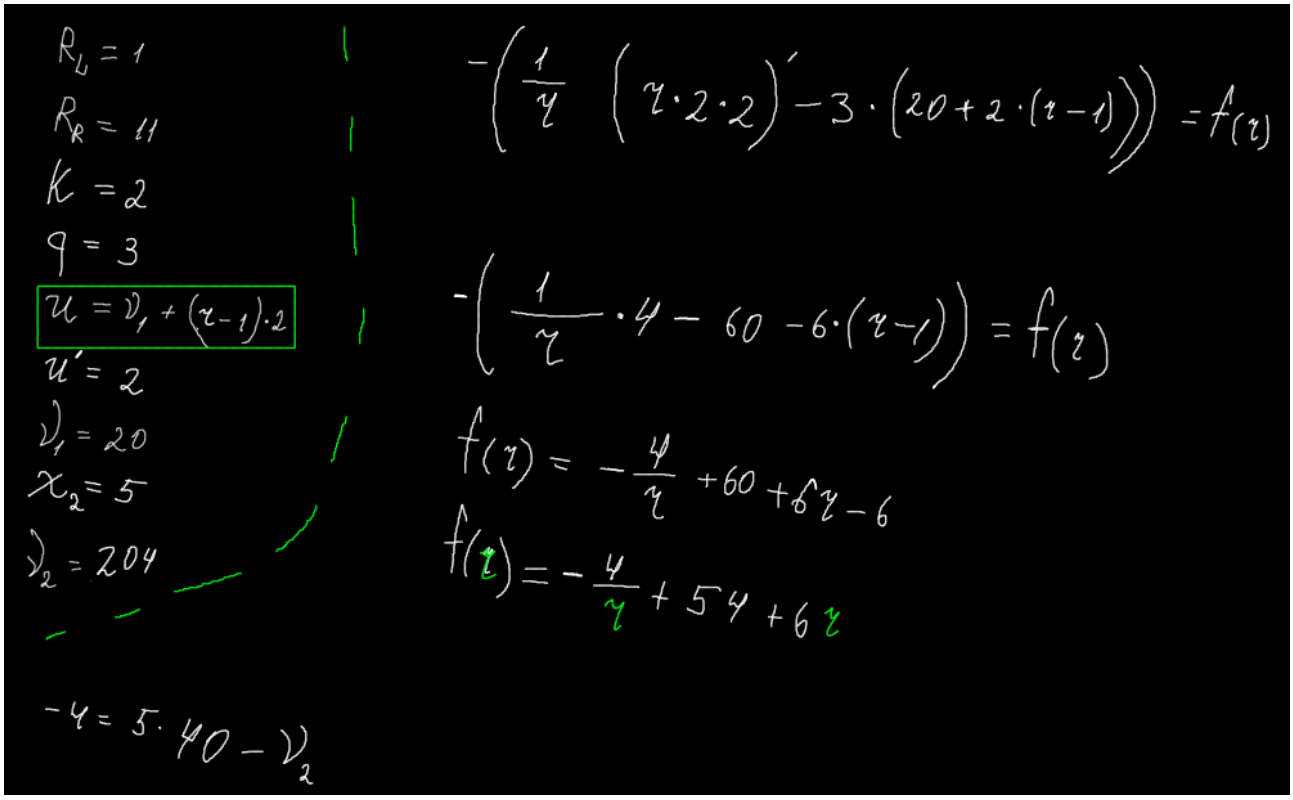
$$-h^2\left[\frac{1}{6}r^n k\frac{d^3u}{dr^3}+\frac{1}{4}\frac{d\left(r^n k\right)}{dr}\frac{d^2u}{dr^2}+\frac{1}{8}\frac{d^2\left(r^n k\right)}{dr^2}\frac{du}{dr}\right]_{r=r_i}+O(h^3),\quad i=N$$

Вторая часть – погрешность аппроксимации. Из формул понятно, что при увеличении величины шага – погрешность также будет расти. Из приведенных выше рисунков – первый описывает разложение невязки для основного уравнения. Второй рисунок показывает вектор незязки для граничного условия третьего рода. Надо учитывать, что оно тоже будет влиять на полученную погрешность.

Оценка работы алгоритма на различных примерах

Для оценки работы составим различные функции и сравним теоретические и экспериментальные данные.

Пример 1



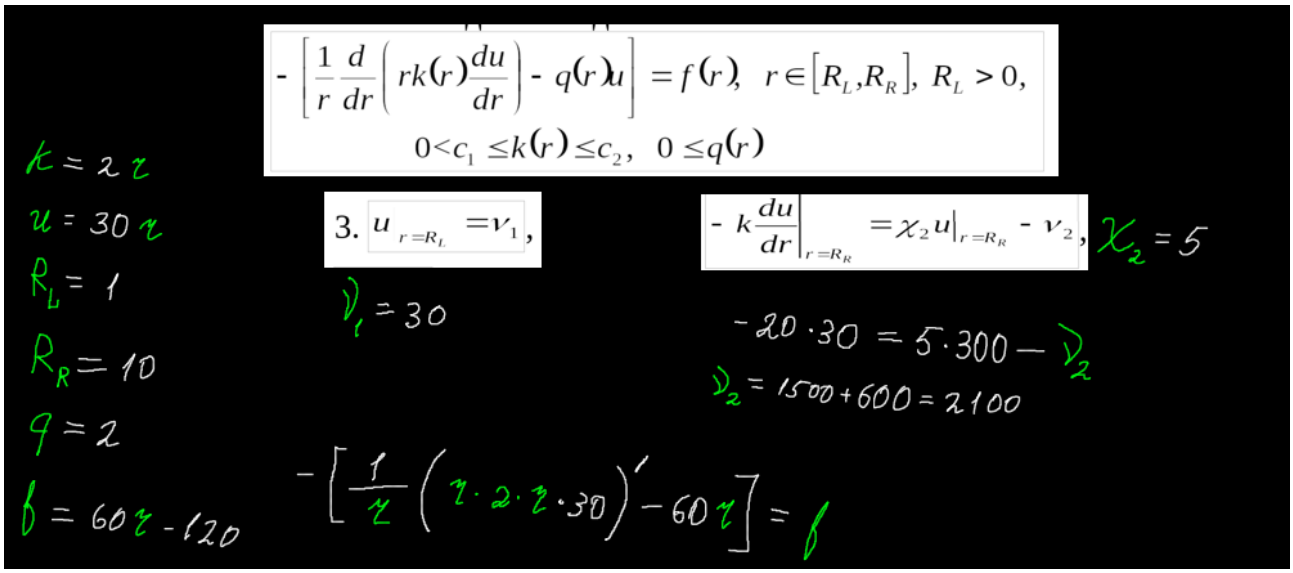
Данный пример предполагает k и q константные для простоты. Введем данные в программу и посмотрим на результат.

steps	avg inaccuracy	max inaccuracy	first half inaccuracy	second half inaccuracy	rel inaccuracy
2	8.881784197001252e-16	3.552713678800501e-15	0e0	3.552713678800501e-15	0.00000000000000000000
4	1.7763568394002505e-15	7.105427357601002e-15	0e0	1.0658141036401503e-14	0.50000000000000000000
8	1.7763568394002505e-15	7.105427357601002e-15	7.105427357601002e-15	1.0658141036401503e-14	1.00000000000000000000
16	3.7500866609560844e-15	1.4210854715202004e-14	3.552713678800501e-15	6.394884621840902e-14	0.47368421052631578947
32	5.015595781836002e-15	1.4210854715202004e-14	3.552713678800501e-14	1.3500311979441904e-13	0.74768518518518518942
64	1.0119851085068094e-14	2.4868995751603507e-14	4.227729277772596e-13	2.4513724383723456e-13	0.49561952440550688381 29
128	2.3775853081203353e-14	6.394884621840902e-14	2.1600499167107046e-12	9.308109838457312e-13	0.42563566701497734801 05

Получаем таблицу показывающую нулевые погрешности. Первый столбец это сумма погрешностей каждого x_i. Второй столбец – максимальная погрешность между i элементами. Третий столбец – сумма погрешностей первой половины найденного вектора. Четвертый столбец – сумма погрешностей второй половины найденного вектора. Пятый столбец – отношение текущей средней погрешности к предыдущей.

Погрешность оказалась близка к нулю, что ожидаемо, поскольку k q были константы, а функция u была линейной. Вектор невязки для основного уравнения станет равен нулю. Остается только вопрос – почему с уменьшением шага – погрешность тоже растет. Ответ прост – то что мы видим – результат именно **вычислительной** погрешности а не аппроксимации. От того что мы дробим шаг – мы только копим погрешность.

Пример 2

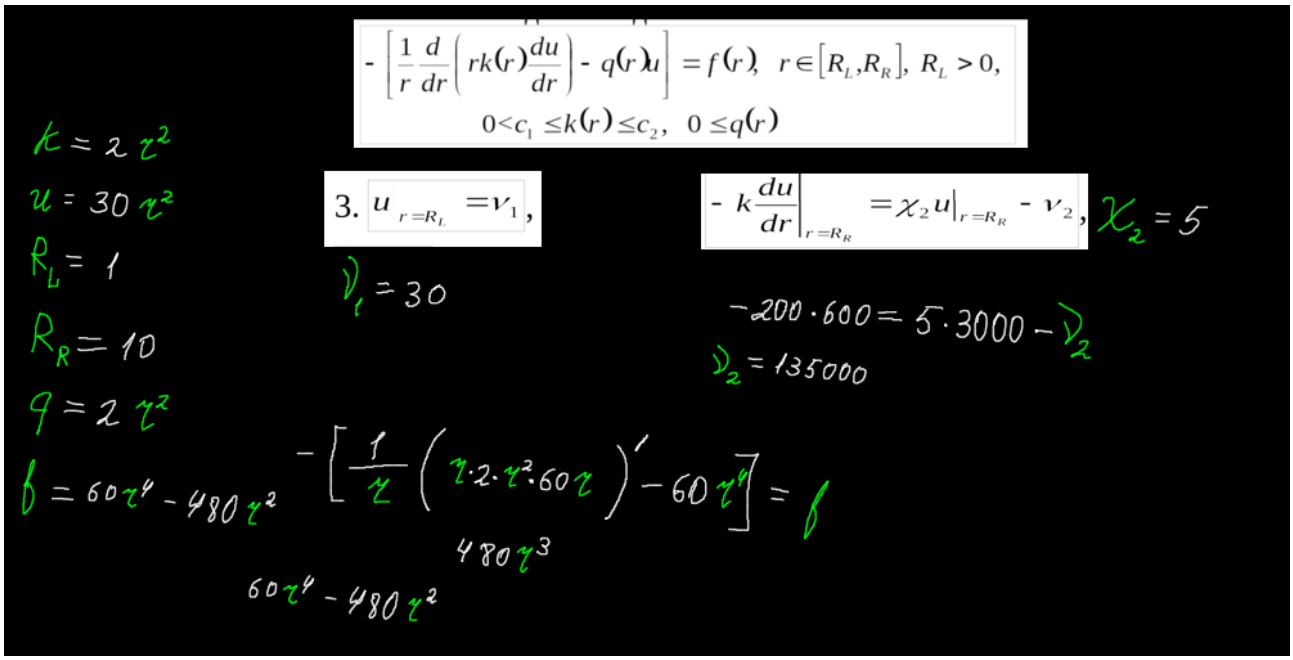


Второй пример уже немного интереснее – в качестве входных данных были взяты все константы, кроме k. k стала линейной функцией. Результаты работы программы следующие:

steps	avg inaccuracy	max inaccuracy	first half inaccuracy	second half inaccuracy	rel inaccuracy
2	8.946981232219002e-1	2.6907936326591084e0	0e0	3.578792492887601e0	0.00000000000000000000
4	2.4920111100447664e-1	7.398007565803368e-1	1.0794381075996284e-1	1.387262855266897e0	3.590265386922082857523
8	6.58778896826128e-2	1.9001759213455216e-1	8.239182455070448e-2	5.763870722754234e-1	3.782773130789107707272
16	1.7007307296837207e-2	4.783836957864196e-2	4.852122639580614e-2	2.5761030494726356e-1	3.873504990108803891502
32	4.3282344926191575e-3	1.1980760058122542e-2	2.6089069629875894e-2	1.2107090311917545e-1	3.929386757080604475334
64	1.0923082092675945e-3	2.9965178208612997e-3	1.3496779701021921e-2	5.859556211063932e-2	3.962466322139325036744
128	2.744066418568554e-4	7.49212519053799e-4	6.860544783307887e-3	2.8812318658083313e-2	3.980618697405285733026

Погрешность уже стала чуть более весомой по сравнению с предыдущим примером. Несмотря на то что вектор невязки основного уравнения снова стал равен нулю – погрешность всё же накопилась. Это связано с тем что погрешность нам еще и дает граничное условие третьего рода справа. Если посмотреть на вектор невязки – то первое слагаемое даже после производной u не стало равно нулю. Можно заметить что с уменьшением размера шага – погрешность также уменьшается, что тоже наталкивает нас на мысль что погрешность дала нам права граница третьего рода.

Пример 3



Теперь приведём пример где все функции хотя бы второй степени.

steps	avg inaccuracy	max inaccuracy	first half inaccuracy	second half inaccuracy	rel inaccuracy
2	2.2950009726705417e1	6.459531828590752e1	0e0	9.180003890682167e1	0.0000000000000000000000
4	1.0059380939642324e1	2.931683399137637e1	2.1205503628243605e1	3.915078200961034e1	2.2814534874868193092823
8	3.5479614247957416e0	1.0178439386364971e1	1.819979402737644e1	1.7279820220580973e1	2.8352565699672025445466
16	1.0212193897626643e0	2.8735740504903333e0	1.1001874677807848e1	7.380074337919723e0	3.4742401685306592007407
32	2.693643120249313e-1	7.438318469280603e-1	5.861247479846739e0	3.297139129000925e0	3.7912200843745140765861
64	6.865296864495173e-2	1.876453708432564e-1	2.989635738202743e0	1.5414601923640703e0	3.9235639381886873167105
128	1.729372769737437e-2	4.701843727116284e-2	1.504506461550612e0	7.436781391080558e-1	3.9698189913892889535418

Теперь к итоговой погрешности добавляется еще и вектор невязки основного уравнения. Производная второй степени больше не зануляет никакие слагаемые и мы видим что даже с разбиением на 128 отрезков – погрешность примерно равна 2e-2.

Выводы

Примененный алгоритм позволяет успешно решать дифференциальные уравненя второго порядка с различными типами граничных условий. Важно отметить что максимальный потенциал этого алгоритма можно раскрыть применив разряженные матрицы для хранения трехдиагональной матрицы. Написанный ниже код выявил невероятную эффективность при применении разряженной матрицы для этого алгоритма. Даже матрица размером 1e6 на 1e6 не стала препятствием чтобы справиться с задачей за считанные милисекунды.

Код

Код программы находится на гитхабе по ссылке https://github.com/Hryapusek/rust-tridiagonal-matrix-vector.git в ветке CP3.

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: i32) -> Vec<f64> {
let step_size = (right - left) / step_count as f64;
let mut points: Vec<f64> = Vec::<f64>::new();

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 + 54.0 + 6.0 * r);

// Boundary conditions: T(0) = 0 (center), T(1) = 1 (outer radius)
let y1 = 20.0;
let hi2 = 5.0;
let y2 = 204.0;

// The 'n' value for cylindrical symmetry (n=1 for 2D axisymmetric cylindrical case)
let n = 1;
let original_function = |r: f64| y1 + (r-1.0) * 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_third_calculator::FirstThirdCalculator::new(
splitter,
&kfunc,
&qfunc,
&ffunc,
y1,
hi2,
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);

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_accuracy_kr_high_accuracy() {
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| 2.0);
let ffunc = LambdaFunction::from(|r: f64| 60.0 * r - 120.0);

// Boundary conditions: T(0) = 0 (center), T(1) = 1 (outer radius)
let y1 = 30.0;
let hi2 = 5.0;
let y2 = 2100.0;

// The 'n' value for cylindrical symmetry (n=1 for 2D axisymmetric cylindrical case)
let n = 1;
let original_function = |r: f64| 30.0 * 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_third_calculator::FirstThirdCalculator::new(
splitter,
&kfunc,
&qfunc,
&ffunc,
y1,
hi2,
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);

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_accuracy_kr_no_accuracy() {
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 * r);
let qfunc = LambdaFunction::from(|r: f64| 2.0 * r * r);
let ffunc = LambdaFunction::from(|r: f64| 60.0 * r.powf(4.0) - 480.0 * r * r);

// Boundary conditions: T(0) = 0 (center), T(1) = 1 (outer radius)
let y1 = 30.0;
let hi2 = 5.0;
let y2 = 135000.0;

// The 'n' value for cylindrical symmetry (n=1 for 2D axisymmetric cylindrical case)
let n = 1;
let original_function = |r: f64| 30.0 * r * 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_third_calculator::FirstThirdCalculator::new(
splitter,
&kfunc,
&qfunc,
&ffunc,
y1,
hi2,
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);

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() {
// exercise_accuracy_base_example();
// exercise_accuracy_kr_high_accuracy();
exercise_accuracy_kr_no_accuracy();
}
```

Math.rs

```
pub mod solver;
pub mod stepping;
pub mod coeff_calculator;
pub mod sparse_matrixes;
```

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)
}
}
```

sparse_matrixes.rs

```
use crate::math::solver::vectors_almost_equal;
use std::ops::Index;
use std::ops::IndexMut;
use std::ops::Mul;

trait TridiagonalMatrixSize {
fn size(&self) -> usize;
}
trait TridiagonalNumberType<NumberType>:
std::convert::From<i32>
+ Clone
+ std::cmp::PartialEq
+ std::cmp::PartialOrd
+ std::ops::Add<Output = NumberType>
+ std::ops::Sub<Output = NumberType>
+ std::ops::Mul<Output = NumberType>
+ std::ops::Div<Output = NumberType>
+ std::ops::AddAssign
+ std::ops::SubAssign
+ std::ops::MulAssign
+ std::ops::DivAssign
+ std::ops::Neg
+ std::ops::Add<NumberType, Output = NumberType>
+ std::ops::Sub<NumberType, Output = NumberType>
+ std::ops::Mul<NumberType, Output = NumberType>
+ std::ops::Div<NumberType, Output = NumberType>
+ std::ops::AddAssign<NumberType>
+ std::ops::SubAssign<NumberType>
+ std::ops::MulAssign<NumberType>
+ std::ops::DivAssign<NumberType>
+ std::ops::Neg<Output = NumberType>
{
}

impl TridiagonalNumberType<i32> for i32 {}
impl TridiagonalNumberType<f64> for f64 {}

pub struct TridiagonalSparseMatrix<NumberType: TridiagonalNumberType<NumberType>> {
pub subdiagonal: Vec<NumberType>,
```

```
pub maindiagonal: Vec<NumberType>,
pub updiagonal: Vec<NumberType>,
pub zero: NumberType,
}

impl<'a, NumberType> Mul<&Vec<NumberType>> for &'a TridiagonalSparseMatrix<NumberType>
where
NumberType: TridiagonalNumberType<NumberType>,
{
type Output = Vec<NumberType>;

fn mul(self, vec: &Vec<NumberType>) -> Self::Output {
let n = self.size();
let mut result = vec![self.zero.clone(); n];

for i in 0..n {
result[i] = self.maindiagonal[i].clone() * vec[i].clone();
if i > 0 {
result[i] =
result[i].clone() + self.subdiagonal[i - 1].clone() * vec[i - 1].clone();
}
if i < n - 1 {
result[i] = result[i].clone() + self.updiagonal[i].clone() * vec[i + 1].clone();
}
}

result
}
}

impl<NumberType: TridiagonalNumberType<NumberType>> TridiagonalSparseMatrix<NumberType> {
fn new(n: usize) -> TridiagonalSparseMatrix<NumberType> {
TridiagonalSparseMatrix {
subdiagonal: vec![NumberType::from(0); n - 1],
maindiagonal: vec![NumberType::from(0); n],
updiagonal: vec![NumberType::from(0); n - 1],
zero: NumberType::from(0),
}
}

#[allow(unused)]
fn check_if_three_diagonals(&self) {
let n = self.size();
if n == 0 {
panic!("The matrix has zero size");
}

// Iterate over each row
for i in 0..n {
let main_value = self.maindiagonal[i].clone();

// Check if the main diagonal element is zero (it shouldn't be)
if main_value == self.zero {
panic!("Matrix is not diagonally dominant: zero on the main diagonal");
}

// Check the condition that the sum of the subdiagonal and updiagonal elements
// is less than or equal to the main diagonal element
let mut sum_of_neighbors = self.zero.clone();

if i > 0 {
let sub_value = self.subdiagonal[i - 1].clone();
sum_of_neighbors = sum_of_neighbors + sub_value;
}

if i < n - 1 {
let up_value = self.updiagonal[i].clone();
sum_of_neighbors = sum_of_neighbors + up_value;
}

if sum_of_neighbors > main_value {
panic!(
"Matrix is not diagonally dominant: the sum of subdiagonal and updiagonal elements at row {} is greater than the main
diagonal element",
i
);
}
}
```

```
}

/// This function calculates the v-coefficients for the sparse matrix
fn calculate_v_coefficients(&self) -> Vec<NumberType> {
let n = self.size();
let mut v_arr = vec![self.zero.clone(); n];
if n > 1 {
v_arr[0] = self.updiagonal[0].clone() / (-self.maindiagonal[0].clone());
for i in 1..n - 1 {
let denom = -self.maindiagonal[i].clone()
- self.subdiagonal[i - 1].clone() * v_arr[i - 1].clone();
v_arr[i] = self.updiagonal[i].clone() / denom;
}
v_arr[n - 1] = self.zero.clone();
}
v_arr
}

/// This function calculates the u-coefficients for the sparse matrix
fn calculate_u_coefficients(
&self,
b: &Vec<NumberType>,
v_arr: &Vec<NumberType>,
) -> Vec<NumberType> {
let n = self.size();
let mut u_arr = vec![self.zero.clone(); n];
if n > 0 {
u_arr[0] = -b[0].clone() / (-self.maindiagonal[0].clone());
for i in 1..n - 1 {
let denom = -self.maindiagonal[i].clone()
- self.subdiagonal[i - 1].clone() * v_arr[i - 1].clone();
u_arr[i] =
(self.subdiagonal[i - 1].clone() * u_arr[i - 1].clone() - b[i].clone()) / denom;
}
let denom_last = -self.maindiagonal[n - 1].clone()
- self.subdiagonal[n - 2].clone() * v_arr[n - 2].clone();
u_arr[n - 1] = (self.subdiagonal[n - 2].clone() * u_arr[n - 2].clone()
- b[n - 1].clone())
/ denom_last;
}
u_arr
}

/// This function solves the system of linear equations Ax = b
/// using the tridiagonal matrix algorithm (Thomas algorithm).
pub fn solve(&self, b: &Vec<NumberType>) -> Vec<NumberType> {
// Check if the matrix is diagonally dominant
self.check_if_three_diagonals();

// Calculate the v and u coefficients
let v_arr = self.calculate_v_coefficients();
let u_arr = self.calculate_u_coefficients(b, &v_arr);

// Back substitution to solve for x
let n = self.size();
let mut x_arr = vec![self.zero.clone(); n];
if n > 0 {
x_arr[n - 1] = u_arr[n - 1].clone();
for i in (0..n - 1).rev() {
x_arr[i] = u_arr[i].clone() + v_arr[i].clone() * x_arr[i + 1].clone();
}
}

x_arr
}
}

impl<'a, NumberType: TridiagonalNumberType<NumberType>> Index<(usize, usize)>
for TridiagonalSparseMatrix<NumberType>
{
type Output = NumberType;
fn index(&self, idx: (usize, usize)) -> &NumberType {
let (x, y) = idx;
match (x, y) {
(x, y) if x == y => self.maindiagonal.get(x).unwrap(),
(x, y) if x == y - 1 => self.subdiagonal.get(x).unwrap(),
(x, y) if x == y + 1 => self.updiagonal.get(x).unwrap(),
_ => &self.zero,
```

```
}
}
}

impl<'a, NumberType: TridiagonalNumberType<NumberType>> IndexMut<(usize, usize)>
for TridiagonalSparseMatrix<NumberType>
{
fn index_mut(&mut self, idx: (usize, usize)) -> &mut NumberType {
let (x, y) = idx;
match (x, y) {
(x, y) if x == y => self.maindiagonal.get_mut(x).unwrap(),
(x, y) if x == y - 1 => self.subdiagonal.get_mut(x).unwrap(),
(x, y) if x == y + 1 => self.updiagonal.get_mut(x).unwrap(),
_ => panic!("Index out of bounds"),
}
}
}

impl<'a, NumberType: TridiagonalNumberType<NumberType>> TridiagonalMatrixSize
for TridiagonalSparseMatrix<NumberType>
{
fn size(&self) -> usize {
self.maindiagonal.len()
}
}

#[cfg(test)]
mod tests {
use crate::math::solver::solve;

use super::*;
use nalgebra::{DMatrix, DVector};
use rand::Rng;
use std::ops::Mul;

// A helper function to create a simple tridiagonal matrix for testing
fn create_test_matrix(
main_diagonal: Vec<f64>,
sub_diagonal: Vec<f64>,
up_diagonal: Vec<f64>,
) -> TridiagonalSparseMatrix<f64> {
TridiagonalSparseMatrix {
maindiagonal: main_diagonal,
subdiagonal: sub_diagonal,
updiagonal: up_diagonal,
zero: 0.0,
}
}

#[test]
fn test_valid_tridiagonal_matrix() {
// A valid tridiagonal matrix where the main diagonal elements are larger
// than the sum of their neighbors
let main_diagonal = vec![4.0, 5.0, 6.0];
let sub_diagonal = vec![1.0, 1.0];
let up_diagonal = vec![1.0, 1.0];

let matrix = create_test_matrix(main_diagonal, sub_diagonal, up_diagonal);
matrix.check_if_three_diagonals(); // Should not panic
}

#[test]
#[should_panic(expected = "Matrix is not diagonally dominant")]
fn test_invalid_tridiagonal_matrix() {
// An invalid tridiagonal matrix where the main diagonal elements are smaller
// than the sum of their neighbors
let main_diagonal = vec![2.0, 2.0, 2.0];
let sub_diagonal = vec![1.5, 1.5];
let up_diagonal = vec![1.5, 1.5];

let matrix = create_test_matrix(main_diagonal, sub_diagonal, up_diagonal);
matrix.check_if_three_diagonals(); // Should panic
}

#[test]
fn test_single_element_matrix() {
// A matrix with a single element, which should be considered diagonally dominant
let main_diagonal = vec![1.0];
```

```rust
let sub_diagonal = vec![];
let up_diagonal = vec![];

let matrix = create_test_matrix(main_diagonal, sub_diagonal, up_diagonal);
matrix.check_if_three_diagonals(); // Should not panic
}

#[test]
#[should_panic]
fn test_zero_size_matrix() {
// A matrix with no elements, which should panic
let main_diagonal = vec![];
let sub_diagonal = vec![];
let up_diagonal = vec![];

let matrix = create_test_matrix(main_diagonal, sub_diagonal, up_diagonal);
matrix.check_if_three_diagonals(); // Should panic
}

#[test]
fn test_basic_multiplication() {
// A simple 3x3 tridiagonal matrix
let main_diagonal = vec![4.0, 5.0, 6.0];
let sub_diagonal = vec![1.0, 1.0];
let up_diagonal = vec![1.0, 1.0];
let matrix = create_test_matrix(main_diagonal, sub_diagonal, up_diagonal);

// Vector to multiply with
let vector: Vec<f64> = vec![1.0, 2.0, 3.0];

// Expected result is calculated as:
// [4*1 + 1*2, 1*1 + 5*2 + 1*3, 1*2 + 6*3] = [6, 12, 20]
let expected_result = vec![6.0, 14.0, 20.0];

// Perform the multiplication
let result = matrix.mul(&vector);

// Check if the result matches the expected values
assert_eq!(result, expected_result);
}

#[test]
fn test_single_element_multiplication() {
// A 1x1 tridiagonal matrix
let main_diagonal = vec![2.0];
let sub_diagonal = vec![];
let up_diagonal = vec![];
let matrix = create_test_matrix(main_diagonal, sub_diagonal, up_diagonal);

// Vector to multiply with
let vector = vec![3.0];

// Expected result: [2 * 3] = [6]
let expected_result = vec![6.0];

// Perform the multiplication
let result = matrix.mul(&vector);

// Check if the result matches the expected values
assert_eq!(result, expected_result);
}

#[test]
fn test_zero_size_multiplication() {
// A matrix with zero size
let main_diagonal = vec![];
let sub_diagonal = vec![];
let up_diagonal = vec![];
let matrix = create_test_matrix(main_diagonal, sub_diagonal, up_diagonal);

// Empty vector to multiply with
let vector: Vec<f64> = vec![];

// Expected result: empty vector
let expected_result: Vec<f64> = vec![];

// Perform the multiplication
let result = matrix.mul(&vector);
```

```rust
// Check if the result matches the expected values
assert_eq!(result, expected_result);
}

#[test]
fn test_larger_matrix_multiplication() {
// A 4x4 tridiagonal matrix
let main_diagonal = vec![4.0, 5.0, 6.0, 7.0];
let sub_diagonal = vec![1.0, 1.0, 1.0];
let up_diagonal = vec![1.0, 1.0, 1.0];
let matrix = create_test_matrix(main_diagonal, sub_diagonal, up_diagonal);

// Vector to multiply with
let vector = vec![1.0, 2.0, 3.0, 4.0];

// Expected result is calculated as:
// [4*1 + 1*2, 1*1 + 5*2 + 1*3, 1*2 + 6*3 + 1*4, 1*3 + 7*4]
let expected_result = vec![6.0, 14.0, 24.0, 31.0];

// Perform the multiplication
let result = matrix.mul(&vector);

// Check if the result matches the expected values
assert_eq!(result, expected_result);
}

/// Function to calculate b = A * x for the sparse matrix
#[allow(unused)]
fn calculate_b_sparse(a: &TridiagonalSparseMatrix<f64>, x_arr: &Vec<f64>) -> Vec<f64> {
a * x_arr
}

/// Print the equation for the sparse matrix
#[allow(unused)]
fn print_equation_sparse(a: &TridiagonalSparseMatrix<f64>, x: &Vec<f64>, b: &Vec<f64>) {
println!("Tridiagonal Matrix A:");
println!("Main Diagonal: {:?}", a.maindiagonal);
println!("Sub Diagonal: {:?}", a.subdiagonal);
println!("Up Diagonal: {:?}", a.updiagonal);
println!("x: {:?}", x);
println!("b: {:?}", b);
}

/// Base test function to solve the sparse matrix system
#[allow(unused)]
fn base_solve_test_sparse(a: &TridiagonalSparseMatrix<f64>, example_x: &Vec<f64>) {
let example_b = calculate_b_sparse(a, example_x);
print_equation_sparse(a, example_x, &example_b);
let solve_x = a.solve(&example_b);
println!("Calculated X: {:?}", solve_x);
println!("Expected X: {:?}", example_x);
println!("----------------------------");
assert!(vectors_almost_equal(
&DVector::from_vec(solve_x),
&DVector::from_vec(example_x.clone()),
0.001
));
}

#[test]
#[allow(non_snake_case)]
fn solve__correct_matrix_1_sparse() {
// A sparse tridiagonal matrix with size 5x5
let main_diagonal = vec![5.0, 5.0, 5.0, 5.0, 5.0];
let sub_diagonal = vec![1.0, 1.0, 1.0, 1.0];
let up_diagonal = vec![1.0, 1.0, 1.0, 1.0];

let matrix = TridiagonalSparseMatrix {
maindiagonal: main_diagonal,
subdiagonal: sub_diagonal,
updiagonal: up_diagonal,
zero: 0.0,
};

// Example solution vector x
let example_x = vec![1.0, 3.0, 2.0, 5.0, 4.0];
```

```
// Test the solver
base_solve_test_sparse(&matrix, &example_x);
}

/// Here we run solve tests with random tridiagonal sparse matrices
#[test]
fn solve_random_matrix_multiple_tests_sparse() {
println!("----------------------------");
println!("Running random sparse 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 random main, sub, and superdiagonals for the tridiagonal matrix
let mut main_diagonal = vec![0.0; n];
let mut sub_diagonal = vec![0.0; n - 1];
let mut up_diagonal = vec![0.0; n - 1];

for i in 0..n {
// Generate the main diagonal element
main_diagonal[i] = rng.gen_range(2.1..10.0);

// Generate the subdiagonal element if applicable
if i > 0 {
sub_diagonal[i - 1] = rng.gen_range(0.1..1.0);
}

// Generate the updiagonal element if applicable
if i < n - 1 {
up_diagonal[i] = rng.gen_range(0.1..1.0);
}
}

// Create the sparse matrix
let matrix = TridiagonalSparseMatrix {
maindiagonal: main_diagonal,
subdiagonal: sub_diagonal,
updiagonal: up_diagonal,
zero: 0.0,
};

// Generate a random example x vector
let example_x: Vec<f64> = (0..n).map(|_| rng.gen_range(1.0..10.0)).collect();

// Perform the test using the base_solve_test_sparse function
base_solve_test_sparse(&matrix, &example_x);
}
}

use std::time::Instant;

/// Test with a large matrix to compare the performance between dense and sparse representations
#[test]
fn solve_large_matrix_performance_test() {
println!("----------------------------");
println!("Running large matrix performance test...");

let mut rng = rand::thread_rng();

// Define the size of the matrix (large size for performance testing)
let n = 100000; // Adjust the size for even larger tests if desired

// Generate random main, sub, and superdiagonals for the tridiagonal matrix
let mut main_diagonal = vec![0.0; n];
let mut sub_diagonal = vec![0.0; n - 1];
let mut up_diagonal = vec![0.0; n - 1];

for i in 0..n {
// Generate the main diagonal element
main_diagonal[i] = rng.gen_range(2.1..10.0);

// Generate the subdiagonal element if applicable
if i > 0 {
sub_diagonal[i - 1] = rng.gen_range(0.1..1.0);
```

```
}

// Generate the updiagonal element if applicable
if i < n - 1 {
up_diagonal[i] = rng.gen_range(0.1..1.0);
}
}

// Create the sparse matrix
let sparse_matrix = TridiagonalSparseMatrix {
maindiagonal: main_diagonal.clone(),
subdiagonal: sub_diagonal.clone(),
updiagonal: up_diagonal.clone(),
zero: 0.0,
};

// Generate a random example x vector
let example_x: Vec<f64> = (0..n).map(|_| rng.gen_range(1.0..10.0)).collect();

// Calculate b = A * x using the sparse matrix
let example_b_sparse = calculate_b_sparse(&sparse_matrix, &example_x);

// Measure the time taken to solve using the sparse matrix approach
let start_sparse = Instant::now();
let solve_x_sparse = sparse_matrix.solve(&example_b_sparse);
let duration_sparse = start_sparse.elapsed();

println!("Solution using sparse matrix(first five elements): {:?}",
solve_x_sparse.iter().take(5usize).collect::<Vec<_>>());

println!(
"Sparse matrix solution time: {:.2?} for size {}",
duration_sparse, n
);
}
}
```

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)]
```

```rust
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);
}
}
}
```

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 **3rd condition** at the right
pub mod first_third_calculator {
use std::marker::PhantomData;

use super::Function;
use crate::math::coeff_calculator::CoeffCalculator;
use crate::math::stepping::{NumberTrait, Stepping};

pub struct FirstThirdCalculator<'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,
hi2: Number,
y2: Number,
n: u16,
}

impl<
'a,
SteppingObject: Stepping<Number>,
KFunctionType: Function<Number>,
QFunctionType: Function<Number>,
FunctionType: Function<Number>,
Number: NumberTrait,
> FirstThirdCalculator<'a, SteppingObject, KFunctionType, QFunctionType, FunctionType, Number>
{
pub fn new(
stepping: SteppingObject,
kfunc: &'a KFunctionType,
qfunc: &'a QFunctionType,
ffunc: &'a FunctionType,
y1: Number,
hi2: Number,
y2: Number,
n: u16,
) -> FirstThirdCalculator<'a, SteppingObject, KFunctionType, QFunctionType, FunctionType, Number>
{
FirstThirdCalculator {
stepping,
kfunc,
qfunc,
ffunc,
y1,
hi2,
y2,
n,
}
}
}
```

```
impl<
'a,
SteppingObject: Stepping<Number>,
KFunctionType: Function<Number>,
QFunctionType: Function<Number>,
FunctionType: Function<Number>,
Number: NumberTrait,
> CoeffCalculator<Number>
for FirstThirdCalculator<'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() {
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 {
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 {
// Contribution from the left neighbor (i-1)
let a_term = self.stepping.middle_point(i - 1).pow(self.n)
* self.kfunc.calc(self.stepping.middle_point(i - 1))
/ self.stepping.step(i);

// Source term at the boundary (if any)
let q_term = self.stepping.cross_step(i)
* self.stepping.point(i).pow(self.n)
* self.qfunc.calc(self.stepping.point(i));

// Robin boundary condition
let boundary_term = self.stepping.point(i).pow(self.n) * self.hi2;

// Return the sum of all terms
a_term + q_term + boundary_term
} 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.stepping.cross_step(i)
* self.stepping.point(i).pow(self.n)
* self.ffunc.calc(self.stepping.point(i))
+ self.stepping.point(i).pow(self.n) * 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)
}
}
```