МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра вычислительной математики

ЛУЦЫК Даниил Васильевич

РЕАЛИЗАЦИЯ НА МНОГОЯДЕРНОМ ПРОЦЕССОРЕ ОБОБЩЕННОГО МЕТОДА МИНИМАЛЬНЫХ НЕВЯЗОК ЧИСЛЕННОГО РЕШЕНИЯ ДВУМЕРНЫХ ПАРАБОЛИЧЕСКИХ УРАВНЕНИЙ ГЕМОДИНАМИКИ

Дипломная работа

Научный руководитель:
доктор физ.-мат. наук,
профессор Н.А. Лиходед

Допущена к защите

“___” _____ 2025 г

Зав. кафедрой вычислительной математики

кандидат физико-математических наук, доцент В.И. Репников

Минск, 2025

ОГЛАВЛЕНИЕ

РЕФЕРАТ

Дипломная работа, 45 с., 16 рис., 3 приложения, 5 источников.

МЕТОД ОБОБЩЕННЫХ МИНИМАЛЬНЫХ НЕВЯЗОК, ГЕМОДИНАМИКА, OPENMP, ОРТОГОНАЛИЗАЦИЯ АРНОЛЬДИ, ВРАЩЕНИЯ ГИВЕНСА, БЛОЧНО-ТРЕХДИАГОНАЛЬНАЯ МАТРИЦА, ПАРАЛЕЛЬННОЕ ВЫПОЛНЕНИЕ.

Объект исследования – метод обобщенных минимальных невязок, OpenMP.

Предметом исследования является метод решения системы уравнений с блочно-трехдиагональной матрицей для двумерных параболических уравнений гемодинамики.

Цель исследования – разработка и реализация последовательного и параллельного алгоритма, анализ эффективности метода обобщенных минимальных невязок.

Методами исследования являются разработка алгоритма обобщенного метода минимальных невязок, параллельные вычисления, вычислительные эксперименты.

В результате работы был успешно разработан алгоритм обобщенных минимальных невязок для двумерных параболических уравнений гемодинамики, адаптирован для параллельного выполнения, проведены вычислительные эксперименты.

Достоверность материалов и результатов дипломной работы: использованные материалы и результаты дипломной работы являются достоверными. Работа выполнена самостоятельно.

Область возможного практического применения: Моделирование кровотока в сосудах, биомедицинские симуляции, численное решение параболических уравнений в инженерных и физических приложениях.

РЭФЕРАТ

Дыпломная работа, 45 с., 16 рыс., 3 дадаткі, 5 крыніц.

МЕТАД АБАГУЛЕНЫХ МІНІМАЛЬНЫХ НЕСУПАДЗЕННЯЎ, ГЕМАДЫНАМІКА, OPENMP, АРТАГАНАЛІЗАЦЫЯ АРНОЛЬДЗІ, ВАРОТЫ ГІВЕНСА, БЛОЧНА-ТРЫДЫЯГАНАЛЬНАЯ МАТРЫЦА, ПАРАЛЕЛЬНАЕ ВЫКАНАННЕ.

Аб’ект даследавання – метад абагуленых мінімальных несупадзенняў, OpenMP.

Прадмет даследавання – метад вырашэння сістэмы раўнанняў з блочна-трыдыяганальнай матрыцай для двухмерных парабалічных раўнанняў гемадынамікі.

Мэта даследавання – распрацоўка і рэалізацыя паслядоўнага і паралельнага алгарытмаў, аналіз эфектыўнасці метаду абагуленых мінімальных несупадзенняў.

Метады даследавання – распрацоўка алгарытма, паралельныя вылічэнні, вылічальныя эксперыменты.

Вынікі – паспяхова распрацаваны і адаптаваны да паралельнага выканання алгарытм GMRES для двухмерных парабалічных раўнанняў гемадынамікі, праведзены вылічальныя эксперыменты.

Даставернасць – усе матэрыялы і вынікі дыпломнай работы з’яўляюцца дакладнымі. Работа выканана самастойна.

Сфера практычнага прымянення – мадэляванне крывацёку, біямедыцынскія сімуляцыі, лікавае вырашэнне парабалічных раўнанняў у інжынерных і фізічных задачах.

ANNOTATION

Thesis, 45p., 16 ill., 3 appendices, 5 sources.

GENERALIZED MINIMAL RESIDUAL METHOD, HEMODYNAMICS, OPENMP, ARNOLDI ORTHOGONALIZATION, GIVENS ROTATIONS, BLOCK TRIDIAGONAL MATRIX, PARALLEL COMPUTING.

Object of the research – Generalized Minimal Residual Method (GMRES), OpenMP.

Subject of the research – The method for solving systems of equations with a block tridiagonal matrix arising from two-dimensional parabolic hemodynamic equations.

Research aim – Development and implementation of sequential and parallel GMRES algorithms and analysis of their efficiency.

Research methods – Algorithm development, parallel computing using OpenMP, computational experiments.

Research results – A GMRES algorithm was successfully developed and adapted for parallel execution. Computational experiments were carried out to evaluate performance.

Reliability – All materials and results presented in the thesis are reliable. The work was completed independently.

Practical application – Blood flow modeling, biomedical simulations, numerical solution of parabolic equations in engineering and physics.

ВВЕДЕНИЕ

Моделирование гемодинамических процессов имеет важное прикладное значение, поскольку заболевания сердечно-сосудистой системы остаются одной из ведущих причин смертности. Одной из основных трудностей при проведении таких исследований является высокая вычислительная нагрузка.

При численном решении двумерных параболических уравнений, описывающих кровоток, возникают системы линейных алгебраических уравнений с блочно-трёхдиагональной структурой. Для их решения эффективно используется обобщённый метод минимальных невязок (GMRes).

С увеличением объёмов вычислений и развитием многоядерных процессоров возникает необходимость реализации методов с применением параллельных вычислений, например, с использованием технологии OpenMP.

Цель данной работы – разработка и реализация последовательного и параллельного алгоритма GMRes, а также исследование его эффективности на примере задач, возникающих в численном решении задач гемодинамики.

ГЛАВА 1
АЛГОРИТМ ОБОБЩЕННОГО МЕТОДА МИНИМАЛЬНЫХ НЕВЯЗОК ДЛЯ ВЫЧИСЛЕНИЯ ДАВЛЕНИЯ

1.1 Постановка задачи

Обобщенный метод минимальных невязок (GMRes – Generalized Mimimal Residual method) [1, 2]. считается в настоящее время одними из наиболее эффективных (часто в сочетании с так называемым предобусловливанием) итерационных методов решения СЛАУ с произвольными вещественными матрицами. GMRes при вычислениях без округлений приводит к точному решению за n (порядок матрицы системы) шагов. Но этот метод принято считать не прямым, а итерационным методом: обычно требуемая точность достигается при некотором m, которое намного меньше порядка матрицы.

Адаптируем алгоритм GMRes для блочно-трёхдиагональных систем линейных алгебраических уравнений, у которых на главной диагонали – трёхдиагональные матрицы-блоки, на поддиагонали и на наддиагонали – диагональные матрицы-блоки. Системы уравнений такого вида часто возникают в приложениях. В частности, такие СЛАУ возникают на каждом временном слое при численном решении двумерных параболических уравнений гемодинамики для получения давления крови в кровеносных сосудах.

Алгоритм программно реализован с использованием технологии параллельного программирования OpenMP. Разработанные алгоритмы и программы могут быть использованы как задел для построения алгоритмов и программ для OpenMP-реализаций двумерных и трёхмерных параболических уравнений гемодинамики.

1.2 Система двумерных параболических уравнений гемодинамики

Движение крови по сосуду (то есть гемодинамику) можно описать системой уравнений в декартовой системе координат *x, z*:

$$\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}+w\frac{\partial u}{\partial z}=-\frac{1}{\rho}\frac{\partial p}{\partial x}+\frac{1}{\rho}\left(\frac{\partial \tau_{xx}}{\partial x}+\frac{\partial \tau_{xz}}{\partial z}\right),$$

$$\frac{\partial w}{\partial t}+u\frac{\partial w}{\partial x}+w\frac{\partial w}{\partial z}=-\frac{1}{\rho}\frac{\partial p}{\partial x}+\frac{1}{\rho}\left(\frac{\partial \tau_{zx}}{\partial x}+\frac{\partial \tau_{zz}}{\partial z}\right),$$

$$\left(\frac{\partial^2 p}{\partial x^2}+\frac{\partial^2 p}{\partial z^2}\right)=2\rho\left(\frac{\partial u}{\partial x}\frac{\partial w}{\partial z}-\frac{\partial u}{\partial z}\frac{\partial w}{\partial x}\right).$$

Здесь u – поперечная компонента скорости, w – продольная компонента скорости, p – давление, ρ − плотность крови, τ_{xx}, τ_{xz}, τ_{zx}, τ_{zz} – некоторые функции.

Найти для требуемого t компоненты скорости u и w – отдельная задача. Найти давление p сводится к решению рассматриваемой далее системы линейных алгебраических уравнений специального вида (1.1).

1.3 Блочно-трёхдиагональная система линейных алгебраических уравнений специального вида

Пусть N, M – некоторые натуральные числа, $A_2,\ldots,A_N$, $B_1,\ldots,B_{N-1}$, $C_1,\ldots,C_N$– вещественные матрицы порядка M, $Y_1,\ldots,Y_N$, $F_1,\ldots,F_N$ – M-мерные векторы. Рассмотрим блочно-трёхдиагональную систему линейных алгебраических уравнений вида

$$\begin{cases} C_1Y_1 + B_1Y_2 = F_1, \\ A_iY_{i-1} + C_iY_i + B_iY_{i+1} = F_i,\ i = 2,\ldots,N-1, \\ A_NY_{N-1} + C_NY_N = F_N \end{cases} \qquad (1.1)$$

Будем рассматривать важный для практического использования в гемодинамике случай трёхдиагональных матриц-блоков C_i:

$$C_i=\begin{pmatrix} c_1^{C_i} & b_1^{C_i} & 0 & 0 & \cdots \\ a_2^{C_i} & c_2^{C_i} & b_2^{C_i} & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & \cdots & a_{M-1}^{C_i} & c_{M-1}^{C_i} & b_{M-1}^{C_i} \\ \cdots & 0 & 0 & a_M^{C_i} & c_M^{C_i} \end{pmatrix},\ i=1,\ldots,N,$$

и диагональных матриц-блоков A_i, B_i:

$$A_i=\begin{pmatrix} c_1^{A_i} & 0 & 0 & 0 & \cdots \\ 0 & c_2^{A_i} & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & \cdots & 0 & c_{M-1}^{A_i} & 0 \\ \cdots & 0 & 0 & 0 & c_M^{A_i} \end{pmatrix},\ i=2,\ldots,N,$$

$$B_i = \begin{pmatrix} c_1^{B_i} & 0 & 0 & 0 & \cdots \\ 0 & c_2^{B_i} & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \cdots & \cdots & 0 & c_{M-1}^{B_i} & 0 \\ \cdots & 0 & 0 & 0 & c_M^{B_i} \end{pmatrix}, \; i = 1, \ldots, N-1.$$

Матрица системы имеет порядок NM и блочно-трёхдиагональную структуру. На главной диагонали – N трёхдиагональных матриц-блоков порядка M, на поддиагонали и на наддиагонали – N–1 диагональных матриц-блоков порядка M.

На рисунке изображён портрет матрицы (отмечены позиции ненулевых элементов) для случая N=4, M=4:

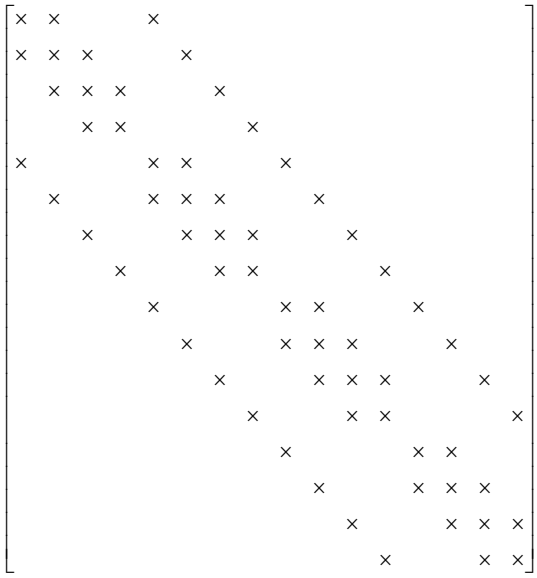

Рисунок 1.1 – Портрет матрицы

Блочные строки и столбцы имеют номера 1,…,N. Вектор неизвестных:

$$(Y_1, \ldots, Y_N) = (y_{1,1}, \ldots, y_{1,M}, \ldots, y_{N,1}, \ldots, y_{N,M}).$$

Вектор правой части:

$$(F_1, \ldots, F_N) = (f_{1,1}, \ldots, f_{1,M}, \ldots, f_{N,1}, \ldots, f_{N,M}).$$

1.4 Обобщенный метод минимальных невязок (GMRes)

Пусть $Ax=f$ – некоторая система линейных алгебраических уравнений, где A является матрицей порядка n, векторы x и f являются n-мерными

векторами. Рассмотрим алгоритм обобщенного метода минимальных невязок (GMRes – Generalized Mimimal Residual method).

Метод GMRES – это проекционный метод, применяемый для численного решения системы линейных уравнений с невырожденной матрицей *A*. При этом решение ищется в подпространстве Крылова:

$$K_m = span\{v_1, Av_1, A^2v_1, \dots, A^{(m-1)}v_1\},$$

где начальный вектор

$$v_1 = \frac{r_0}{\beta}, r_0 = b - Ax_0, \beta = \|r_0\|_2.$$

Для построения ортонормированного базиса K используется ортогонализация Арнольди, в результате которой формируется матрица V_{m+1}, состоящая из $m + 1$ ортонормированных векторов-столбцов:

$$V_{m+1} = [v_1, v_2, \dots, v_{m+1}].$$

Справедливо равенство $r = b - A(x_0 + V_m y) = V_{m+1}(\beta e_1 - \bar{H}_m y)$ где:

- $\bar{H}_m \in \mathbb{R}^{(m+1)\times m}$ – верхняя Хессенбергова матрица, полученная в процессе ортогонализации Арнольди;
- $e_1 \in \mathbb{R}^{m+1}$ – первый базисный вектор (единичный вектор с единицей на первом месте);
- βe_1 – вектор, пропорциональный начальному остатку r_0.

Поскольку матрица V_{m+1} ортонормирована, норма остатка сохраняется:

$$\| r \|_2 = \| V_{m+1}(\beta e_1 - \bar{H}_m y) \|_2 = \| \beta e_1 - \bar{H}_m y \|_2 = J(y).$$

Таким образом, задача минимизации нормы остатка сводится к задаче наименьших квадратов:

$$\min_y \| \beta e_1 - \bar{H}_m y \|_2.$$

Данная система является переопределённой, так как $\bar{H}_m$ имеет размерность $(m + 1) \times m$. Стандартный способ её решения – метод наименьших квадратов. Однако, с учётом структуры $\bar{H}_m$более эффективно применять вращения Гивенса, которые позволяют привести $\bar{H}_m$ к верхнетреугольному виду и упростить решение.

Алгоритмическое предписание GMRes(*m*):

- Выбрать x^0 (при первом старте x^0 – произвольный вектор),

вычислить $v_1 = \frac{r_0}{\|r_0\|_2}$ ($r_0 = f - Ax^0$), выбрать *m*, ε.

- Выполнять заданное число m шагов алгоритма Арнольди получения ортонормированного базиса с первым базисным вектором v_1. На шаге β получить β-е столбцы матриц V_m и $\overline{H}_m$.

- После каждого шага β алгоритма Арнольди посредством вращений приводить β-й столбец и β-й элемент правой части СЛАУ $\overline{H}_m y = \|r_0\|_2 e_1$ (e_1 – вектор-столбец размерности $m+1$ с единичной первой компонентой и нулевыми остальными компонентами) с почти треугольной матрицей $\overline{H}_m$ к столбцу и правой части системы $\overline{R}_m y = \overline{g}_m$, первые m уравнений которой образуют систему $R_m y = g_m$ с треугольной матрицей. Если при некотором $\beta = n_r$ ($\beta \le m$) требуемая точность ε достигнута, то перейти к решению системы $R_{n_r} y = g_{n_r}$. Проверка достижения точности осуществляется оценкой невязки $\left\| f - Ax^{n_r} \right\|_2 : \left| b_{n_r+1} \right| < \varepsilon$, где b_{n_r+1} – последний элемент вектора $\overline{g}_{n_r}$.

- Найти решение y_{n_r} системы $R_{n_r} y = g_{n_r}$. Вычислить $x^{n_r} = x^0 + V_{n_r} y_{n_r}$.

- Если заданная точность была достигнута, то выйти из итерационного процесса. Если заданная точность не была достигнута, то положить $x^0 = x^m$ и повторить шаги, начиная с первого шага.

При выполнении алгоритма GMRes возможны обрывы, если алгоритм Арнольди не сможет построить очередной вектор базиса. Это означает, что получено точное решение.

1.5.1 Ортогонализация Арнольди

Для построения базиса в подпространстве Крылова в методе GMRES сначала выбирается множество векторов:

$w_1 = v_1, w_2 = Aw_1, w_3 = A^2 w_1 = Aw_2, \dots, A^{m-1} w_1 = Aw_{m-1}$ ...

Тогда по определению подпространства Крылова $K_m(v_1, A) = span(w_1, w_2, \dots, w_m)$ переход от базиса $|w_1, w_2, \dots, w_m|$ к базису $|v_1, v_2, \dots, v_m|$ осуществляется с помощью процедуры ортогонализации $v_{k+1} = w_{k+1} - \sum_{i=1}^{k} \alpha_i v_i$. Полученные векторы нормируются. Предположим, что предыдущие k векторов построены, т.е.

$$\forall i, j (1 \le i, j \le k): (v_i, v_j) = \begin{cases} 0, i \ne j \\ 1, i = j \end{cases}$$

Тогда $v_{k+1} = Av_{k+1} - \sum_{i=1}^{k} \alpha_i v_i$.

Для выполнения условия ортогональности вектора v_{k+1} ко всем предыдущим векторам, умножим равенство скалярно на $v_j (j \le k)$ и приравняем результат к нулю.

$$(Av_k, v_j) - \sum_{i=1}^{k} \propto_i (v_i, v_j) = 0$$

Можно получить выражения для коэффициентов $\propto_j$:

$$\propto_j = (Av_k, v_j)$$

Алгоритм метода ортогонализации Арнольди:

Выбрать вектор v_1,$||v_1||=1$,

$j=1, 2, \ldots, m$

$w_j = Av_j$

$i=1, 2, \ldots, j$

$h_{ij} = (w_i, v_j)$

$w_j = w_j - h_{ij}v_i$

$h_{j+1,j} = ||w_j||_2$

$v_{j+1} = w_j / h_{j+1,j}$

1.5.2 Вращения для приведения СЛАУ с почти треугольной матрицей к системе с треугольной матрицей.

Требуется посредством вращений приводить, столбец за столбцом, СЛАУ $\overline{H}_m y = \|r_0\|_2 e_1$ с почти треугольной матрицей $\overline{H}_m$ к системе $\overline{R}_m y = \overline{g}_m$, первые m уравнений которой образуют систему $R_m y = g_m$ с треугольной матрицей. Предполагается, что на шаге $\beta=1,2,\ldots,m$, β-й столбец матрицы $\overline{H}_m$ известен, а до этого шага неизвестен.

Рассмотрим следующие матрицы вращения некоторого порядка n (в дальнейшем будет $n=m+1$):

$$T_{k,k+1} = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & c_k & s_k & & & \\ & & & -s_k & c_k & & & \\ & & & & & 1 & & \\ & & & & & & \ddots & \\ & & & & & & & 1 \end{bmatrix};$$

все необозначенные элементы – нулевые, а величины c_k и $\pm s_k$ $\left(c_{k,k+1}^2 + s_{k,k+1}^2 = 1\right)$ расположены на пересечении строк и столбцов с номерами k и $k+1$.

Если умножить матрицу $T_{k,k+1}$ на произвольную невырожденную матрицу A порядка n, то получим новую невырожденную матрицу B, у которой элементы k-й и $(k+1)$-й строк определяются по формулам

$$j = 1, 2, \ldots, n:$$

$$b_{kj} = a_{kj} c_k + a_{k+1\,j} s_k, \qquad (1.2)$$

$$b_{k+1\,j} = -a_{kj} s_k + a_{k+1\,j} c_k,$$

а остальные элементы матрицы B такие же, как и соответствующие элементы матрицы A. Известно, что если в матрице $T_{k,k+1}$ положить

$$c_k = \frac{a_{kk}}{\sqrt{a_{kk}^2 + a_{k+1\,k}^2}}, \quad s_k = \frac{a_{k+1\,k}}{\sqrt{a_{kk}^2 + a_{k+1\,k}^2}}. \qquad (1.3)$$

то после преобразования $B = T_{k,k+1} A$ получим $b_{k+1\,k} = 0$.

Опишем первых два шага метода вращений. Примем, что верхний индекс определяется первым индексом матрицы вращений, нулевой индекс приписан исходной системе $\overline{H}_m y = \|r_0\|_2 e_1$. Каждый шаг β, β=1,2,…,$m$, вращений сводится к обнулению только одного элемента (матрица $\overline{H}_m$ является почти треугольной), пересчёту элементов столбца выше обнуляемого элемента, пересчёту β-го и (β+1)-го элементов правой части системы $\overline{R}_m y = \overline{g}_m$. Нулевые элементы в первых β−1 столбцах при этом сохраняются (это гарантирует метод вращений).

На первом шаге (β=1) выполняется умножение матрицы T_{12} (используются c_1 и s_1) на первый столбец матрицы $\overline{H}_m$: элемент h_{11}^0 пересчитывается (и получается элемент r_{11} матрицы R_m), обнуляется элемент h_{21}^0. Пересчитываются также элементы правой части $\gamma_1^0 = \|r_0\|_2$ и $\gamma_2^0 = 0$. В первом столбце стоят нули во всех позициях, кроме первой. Преобразования:

$$r_{11} = h_{11}^1 = h_{11}^0 c_1 + h_{21}^0 s_1,$$

$$\gamma_1 = \gamma_1^1 = \gamma_1^0 c_1, \quad \gamma_2^1 = -\gamma_1^0 s_1,$$

$$c_1 = \frac{h_{11}^0}{\sqrt{\left(h_{11}^0\right)^2 + \left(h_{21}^0\right)^2}}, \quad s_1 = \frac{h_{21}^0}{\sqrt{\left(h_{11}^0\right)^2 + \left(h_{21}^0\right)^2}}.$$

На втором шаге (β=2) сначала выполняется умножение матрицы T_{12} (используются c_1 и s_1) на второй столбец матрицы $\overline{H}_m$: элемент h_{12}^0 пересчитывается (и получается элемент r_{12} матрицы R_m), пересчитывается элемент h_{22}^0 (и получается h_{22}^1). Затем выполняется умножение матрицы T_{23} (используются c_2 и s_2) на второй столбец получившейся матрицы: элемент h_{22}^1

пересчитывается (и получается элемент r_{22} матрицы R_m), обнуляется элемент h_{32}^0 Пересчитываются также элементы правой части γ_2^1 и $\gamma_3^0 = 0$. В первом столбце стоят нули во всех позициях, кроме первой, во втором столбце стоят нули во всех позициях, кроме первой и второй. Преобразования:

$$r_{12} = h_{12}^1 = h_{12}^0 c_1 + h_{22}^0 s_1,$$

$$h_{22}^1 = -h_{12}^0 s_1 + h_{22}^0 c_1,$$

$$r_{22} = h_{22}^2 = h_{22}^1 c_2 + h_{32}^1 s_2,$$

$$\gamma_2 = \gamma_2^2 = \gamma_2^1 c_2, \quad \gamma_3^2 = -\gamma_2^1 s_2,$$

$$c_2 = \frac{h_{22}^1}{\sqrt{\left(h_{22}^1\right)^2 + \left(h_{32}^1\right)^2}}, \quad s_2 = \frac{h_{32}^1}{\sqrt{\left(h_{22}^1\right)^2 + \left(h_{32}^1\right)^2}}.$$

ГЛАВА 2
РЕАЛИЗАЦИЯ МЕТОДА GMRES

2.1 Программная реализация последовательного алгоритма GMRES

Перед реализацией введем необходимые нам леммы.

Получим алгоритм умножения матрицы системы на вектор вида

$$X = (X_1,..., X_N) = \left(x_{1,1},..., x_{1,M}, \ ... \ , x_{N,1},..., x_{N,M}\right). \qquad (2.1)$$

Лемма 1. Пусть A – матрица системы вида (1.1), X – вектор вида (2.1). Тогда элементы вектора $Z=AX$,

$$Z = (Z_1,..., Z_N) = \left(z_{1,1},..., z_{1,M}, \ ... \ , z_{N,1},..., z_{N,M}\right), \qquad (2.2)$$

могут быть вычислены следующим образом:

Вход: двумерные массивы

$cC(i,k)$, $1 \le i \le N$, $1 \le k \le M$ // $cC(i,k) = c_k^{C_i}$

$aC(i,k)$, $1 \le i \le N$, $2 \le k \le M$ // $aC(i,k) = a_k^{C_i}$

$bC(i,k)$, $1 \le i \le N$, $1 \le k \le M-1$ // $bC(i,k) = b_k^{C_i}$

$cA(i,k)$, $1 \le i \le N$, $1 \le k \le M$ // $cA(i,k) = c_k^{A_i}$

$cB(i,k)$, $1 \le i \le N$, $1 \le k \le M$ // $cB(i,k) = c_k^{B_i}$

$x(i,k)$, $1 \le i \le N$, $1 \le k \le M$

Выход: двумерный массив

$z(i,k)$, $1 \le i \le N$, $1 \le k \le M$

do $i = 1, N$

if $i=1$ do

$z_{i,1} = c_1^{C_i} x_{i,1} + b_1^{C_i} x_{i,2} + c_1^{B_i} x_{i+1,1}$

do $k = 2, M-1$

$z_{i,k} = a_k^{C_i} x_{i,k-1} + c_k^{C_i} x_{i,k} + b_k^{C_i} x_{i,k+1} + c_k^{B_i} x_{i+1,k}$

enddo

$z_{i,M} = a_M^{C_i} x_{i,M-1} + c_M^{C_i} x_{i,M} + c_M^{B_i} x_{i+1,M}$

endif($i=1$)

if $i \ne 1$, $i \ne N$ do

$z_{i,1} = c_1^{A_i} x_{i-1,1} + c_1^{C_i} x_{i,1} + b_1^{C_i} x_{i,2} + c_1^{B_i} x_{i+1,1}$

do $k = 2, M-1$

$z_{i,k} = c_k^{A_i} x_{i-1,k} + a_k^{C_i} x_{i,k-1} + c_k^{C_i} x_{i,k} + b_k^{C_i} x_{i,k+1} + c_k^{B_i} x_{i+1,k}$

enddo

$z_{i,M} = c_M^{A_i} x_{i-1,M} + a_M^{C_i} x_{i,M-1} + c_M^{C_i} x_{i,M} + c_M^{B_i} x_{i+1,M}$

endif($i\neq 1$, $i\neq N$)

if $i=N$ do

$$z_{i,1} = c_1^{A_i} x_{i-1,1} + c_1^{C_i} x_{i,1} + b_1^{C_i} x_{i,2}$$

do $k = 2, M-1$

$$z_{i,k} = c_k^{A_i} x_{i-1,k} + a_k^{C_i} x_{i,k-1} + c_k^{C_i} x_{i,k} + b_k^{C_i} x_{i,k+1}$$

enddo

$$z_{i,M} = c_M^{A_i} x_{i-1,M} + a_M^{C_i} x_{i,M-1} + c_M^{C_i} x_{i,M}$$

endif$(i=N)$

enddo

Лемма 2. Скалярное произведение векторов вида (2), (3) может быть вычислено следующим образом:

Вход: двумерные массивы

$x(i,k)$, $1\le i\le N$, $1\le k\le M$

$z(i,k)$, $1\le i\le N$, $1\le k\le M$

Выход:

константа xz

do $i = 1, N$

$$s_i = x_{i,1} z_{i,1}$$

do $k = 2, M-1$

$$s_i = s_i + x_{i,k} z_{i,k}$$

enddo

enddo

$$xz = s_1$$

do $i = 1, N$

$$xz = xz + s_i$$

enddo

Лемма 3. Умножение вектора вида 2 на константу c может быть вычислено следующим образом:

Вход:

двумерный массив $x(i,k)$, $1\le i\le N$, $1\le k\le M$

константа c

Выход:

двумерный массив $z(i,k)$, $1\le i\le N$, $1\le k\le M$

do $i = 1, N$

do $k = 1, M$

$$z_{i,k} = x_{i,k} * c$$

enddo

enddo

Лемма 4. Пусть V – матрица размера $NM \times m$, y – m-мерный вектор. Тогда элементы вектора $t=Vy$,

$$(T_1,...,T_N) = \left(t_{1,1},...,t_{1,M}, \ ... \ ,t_{N,1},...,t_{N,M}\right),$$

могут быть вычислены следующим образом:

Вход:

трёхмерный массив $v(\beta,i,j)$, $1 \le \beta \le m$, $0 \le i \le N$, $1 \le k \le M$

Выход:

двумерный массив $t(i,j)$, $0 \le i \le N$, $1 \le k \le M$

```
do  i = 0, N
  do  k = 1, M
    t(i, j) =0
    do  β =1, m
      t(i,k) = t(i,k) + v(β,i,k) y(β)
    enddo
  enddo
enddo
```

2.2 Реализация метода

Псевдокод GMRes(m) с рестартами (алгоритм Арнольди и вращения выполняются одновременно):

Используемые массивы:

xm – двумерный массив, определяющий приближение к решению исходной системы $Ax=f$; $xm(i,j)$, $0 \le i \le N$, $1 \le j \le M-1$. При программировании будем считать $0 \le j \le M-1$; $j=0$ избыточно, но к существенным накладным расходам это не приводит. Избыточные элементы массивов фактически не используются.

fslae – двумерный массив, правая часть исходной СЛАУ; $fslae(i,j)$, $0 \le i \le N$, $1 \le j \le M-1$. При программировании будем считать $0 \le j \le M-1$.

v – трёхмерный массив, определяющий ортонормированный базис V_{m+1}; $v(\beta,i,j)$, $1 \le \beta \le m+1$, $0 \le i \le N$, $1 \le j \le M-1$. Матрица $V_{m+1}=(v_1,...,v_m,v_{m+1})$ имеет размер $((N+1)(M-1)) \times (m+1)$. При программировании будем считать $0 \le \beta \le m+1$, $0 \le j \le M-1$; $\beta=0$, $j=0$ избыточны.

*tempv*1 – временный двумерный массив; $tempv1(i,j)$, $0 \le i \le N$, $1 \le j \le M-1$. Используется в качестве вектора Ax^m (Ax^0 при первом старте), вектора $h_{\alpha\beta}v_\alpha$ (при фиксированных α и β), а также как вычисленный по алгоритму (mv*Vy*) результат умножения матрицы V_m на вектор y. При программировании будем считать $0 \le j \le M-1$.

*tempv*2 – временный двумерный массив; *tempv*2(*i*, *j*), 0≤*i*≤*N*, 1≤*j*≤*M*–1. Используется в качестве вектора невязки $r = r_0 = f - Ax^0$ (при первом старте) или $r = r_m = f - Ax^m$, а также в качестве вспомогательного вектора w. При программировании будем считать 0≤*j*≤*M*–1.

h – двумерный массив. При реализации алгоритма Арнольди $h(\alpha,\beta)$, 1≤β≤*m*, 1≤α≤*m*+1, – коэффициенты почти треугольной матрицы $\overline{H}_m$. При программировании будем считать 0≤β≤*m*, 0≤α≤*m*+1. Избыточные и нулевые элементы при вычислениях не используются.

r – двумерный массив, $r(\alpha,\beta)$, 1≤β≤*m*, 1≤α≤*m*, – коэффициенты матрицы R_m (при 1≤β≤*m*, 1≤α≤β, – ненулевые коэффициенты). При программировании будем считать 0≤β≤*m*, 0≤α≤*m*. Избыточные и нулевые элементы при вычислениях не используются.

b – одномерный массив, $b(\beta)$, 1≤β≤*m*+1, – правая часть системы $\overline{R}_m y = \overline{g}_m$. При программировании будем считать 0≤β≤*m*+1.

c, *s* – одномерные массивы, $c(\beta)$, $s(\beta)$, 1≤β≤*m*, – элементы матриц вращений. При программировании будем считать 0≤β≤*m*.

y – одномерный массив, *y*(α), 1≤α≤*m*. При программировании 0≤α≤*m*.

Вход: двумерный массив *xm*, задающий начальное приближение к решению СЛАУ *Ax*=*f*; двумерный массив *fslae*, задающий правую часть *f*; подпрограмма, реализующая алгоритм (mv*Ax*) умножения матрицы СЛАУ на вектор; подпрограмма, реализующая алгоритм (*sxz*) скалярного произведения векторов; подпрограмма, реализующая алгоритм (*xc*) умножение вектора на константу; целое число *maxstart* – максимальное число стартов; целое число *m* – максимальное число шагов алгоритма Арнольди на одном старте; число ε, характеризующее точность итерационного алгоритма.

Выход: двумерный массив *xm*(*i*, *j*), 0≤*i*≤*N*, 1≤*j*≤*M*–1, задающий приближение к решению исходной СЛАУ, *enres* – евклидова норма невязки $\left\| f - Ax^m \right\|_2$ на последнем старте.

xm= x^0 // если производится рестарт, то вектору *xm* автоматически присвоится вычисленное перед рестартом приближение к решению системы *Ax*=*f*

m=... // максимальное число шагов алгоритма Арнольди на одном старте

ε =... // требуемая евклидова норма невязки

maxstart=... // максимальное число стартов

start=0

start=*start*+1

while(*start*≤*maxstart*) do

$xm = x^0$ // если производится рестарт, то вектору xm автоматически присвоится вычисленное перед рестартом приближение к решению системы $Ax=f$

$tempv1 = \ldots$ // вектор Ax^m (Ax^0 при первом старте), полученный по алгоритму (mvAx) умножения матрицы A на вектор xm

$tempv2 = fslae - tempv1$ // $r = r_0 = f - Ax^0$ (при первом старте), $r = r_m = f - Ax^m$

if $start=1$ then $scal\_tempv2\_tempv2 = \ldots$ // число $(r,r) = (r_0, r_0)$, полученное по алгоритму (sxz) скалярного произведения векторов;

if $start=1$ then $enres = \sqrt{scal\_tempv2\_tempv2}$ // евклидова норма невязки

$b(1) = enres$ // $b = \|r\|_2 e_1$

$$enres1 = \frac{1}{enres}$$

$v_1 = \ldots$ // вектор $v_1 = \dfrac{r}{\|r\|_2}$, полученный по алгоритму леммы 3 умножения вектора $tempv2$ (равного r) на константу $enres1$;

фактически имеем массив $v(1, i, j)$, $0 \le i \le N$, $1 \le j \le M-1$, играющий роль вектор-столбца v_1

$\beta = 1, 2, \ldots, m$

$tempv2 = \ldots$ // вспомогательный вектор $w = Av_\beta$, полученный по алгоритму (mvAx) умножения матрицы A на вектор v_β

$\alpha = 1, 2, \ldots, \beta$ // модифицированный алгоритм Грамма-Шмидта

$scal\_v_\alpha\_tempv2 = \ldots$ // число (v_α, w), полученное по алгоритму леммы 2 скалярного произведения векторов

$h(\alpha,\beta) = scal\_v_\alpha\_tempv2$ // $h_{\alpha\beta} = (v_\alpha, w)$

$tempv1 = \ldots$ // вектор $h_{\alpha\beta} v_\alpha$, полученный по алгоритму леммы 3 умножения вектора v_α на константу $scal\_v_\alpha\_tempv2$ (равную $h_{\alpha\beta}$)

$tempv2 = tempv2 - tempv1$ // $w = w - h_{\alpha\beta} v_\alpha$

end(α)

$scal\_tempv2\_tempv2 = \ldots$ // число (w, w), полученное по алгоритму (sxz) скалярного произведения векторов

$h(\beta+1,\beta) = \sqrt{scal\_tempv2\_tempv2}$ // $h_{\beta+1,\beta} = \|w\|_2 = \sqrt{(w,w)}$

$$h1 = \frac{1}{h(\beta+1,\beta)}$$

$v_{\beta+1} = \ldots$ // очередной вектор $v_{\beta+1} = w / h_{\beta+1,\beta}$ ортонормированного базиса, полученный по алгоритму (*xc*) умножения вектора $tempv2$ (равного w) на константу $h1$;

фактически имеем массив

$v(\beta+1, i, j)$, $0 \le i \le N$, $1 \le j \le M-1$, играющий роль вектор-столбца $v_{\beta+1}$;

предостережение: $h_{\beta+1,\beta}$ может быть близким к 0

// Начало использования вращений для получения матрицы и правой части системы $\overline{R}_m y = \overline{g}_m$

$r(1,\beta) = h(1,\beta)$ // $r_{\alpha,\beta}$ используют и для получения промежуточных значений

$\alpha = 2, 3, \ldots, \beta$ // применение ранее полученных вращений, при $\beta=1$ цикл не выполняется

$temp = c(\alpha-1)r(\alpha-1,\beta) + s(\alpha-1)h(\alpha,\beta)$

$r(\alpha,\beta) = -s(\alpha-1)r(\alpha-1,\beta) + c(\alpha-1)h(\alpha,\beta)$

$r(\alpha-1,\beta) = temp$

end(α)

$\text{delta} = \sqrt{\left(r(\beta,\beta)\right)^2 + \left(h(\beta+1,\beta)\right)^2}$ // начало получение новых вращений

$$c(\beta) = \frac{r(\beta,\beta)}{\text{delta}}$$

$$s(\beta) = \frac{h(\beta+1,\beta)}{\text{delta}}$$ // конец получение новых вращений

$r(\beta,\beta) = c(\beta)r(\beta,\beta) + s(\beta)h(\beta+1,\beta)$ // вращение для диагональных элементов

$b(\beta+1) = -s(\beta)b(\beta)$ // применение вращений к правой части

$b(\beta) = c(\beta)b(\beta)$

// Конец использования вращений

if $enres = |b(\beta+1)| < \varepsilon$ $|b(\beta+1)| < \varepsilon$ then $nr = \beta$, goto Gauss

end(β)

$nr = m$

Gauss:

$$y(nr) = \frac{b(nr)}{r(nr, nr)}$$

$\alpha = nr-1, 1$ // шаг цикла равен -1

$y(\alpha) = b(\alpha)$

$\beta = \alpha+1, nr$

$y(\alpha) = y(\alpha) - r(\alpha,\beta)y(\beta)$

end (β)

$$y(\alpha) = \frac{y(\alpha)}{r(\alpha,\alpha)}$$

end (α)
end (Gauss)
$tempv1 = ...$ // вектор $V_{n_r} y_{n_r}$, полученный по алгоритму леммы 4
$xm = xm + tempv1$ // $x^{n_r} = x^0 + V_{n_r} y_{n_r}$
$enres = |b(nr+1)|$
goto $start=start+1$
end while($start \le maxstart$)
end
Реализация кода представлена в приложении А.

2.3 Вычислительные эксперименты последовательного алгоритма

Характеристики устройства, на котором выполнялись вычисления:

Модель	ПК на базе Ryzen
Процессор	AMD Ryzen 7 5800H
Оперативная память	16GB
Количество ядер	8

Таблица 2.1 – Характеристики вычислительного устройства

Рассмотрим несколько примеров задания элементов матриц-блоков:
Пример 1.
Все диагональные элементы диагональных матриц-блоков равны 4:
$cC(i,k) = 4,\ 1 \le i \le N,\ 1 \le k \le M;$
другие элементы:

$$aC(i,k) = \frac{2i+k}{2N+M},\ 1 \le i \le N,\ 2 \le k \le M;$$

$$bC(i,k) = \frac{2i+k}{2N+M},\ 1 \le i \le N,\ 1 \le k \le M-1;$$

$$cA(i,k) = \frac{2i+k}{2N+M},\ 1 \le i \le N,\ 1 \le j \le M;$$

$$cB(i,k) = \frac{2i+k}{2N+M},\ 1 \le i \le N,\ 1 \le j \le M.$$

Матрица при размере блоков равных $N, M = 4$ будет выглядеть следующим образом (рис 2.1)

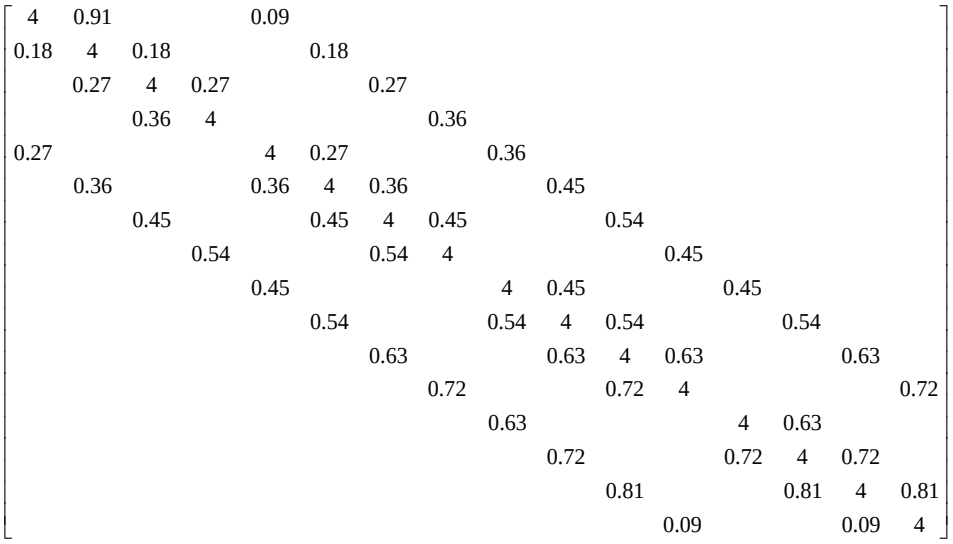

Рисунок 2.1 – Пример матрицы при $N, M = 4$

Пример 2.

Внутренние строки внутренних блочных строк,

$2 \le i \le N-1,\ 2 \le k \le M-1$:

$$cC(i,k) = -4,\ aC(i,k) = 1,\ bC(i,k) = 1,\ cA(i,k) = 1,\ cB(i,k) = 1.$$

Первая строка внутренних блочных строк,

$2 \le i \le N-1,\ k = 1$:

$$cC(i,k) = -5,\ bC(i,k) = 1,\ cA(i,k) = 1,\ cB(i,k) = 1.$$

Последняя строка внутренних блочных строк,

$2 \le i \le N-1,\ k = M$:

$$cC(i,k) = -5,\ aC(i,k) = 1,\ cA(i,k) = 1,\ cB(i,k) = 1.$$

Внутренние строки первой блочной строки,

$i = 1,\ 2 \le k \le M-1$:

$$cC(i,k) = -3,\ aC(i,k) = 1,\ bC(i,k) = 1,\ cB(i,k) = 1.$$

Первая строка первой блочной строки,

$i = 1,\ k = 1$:

$$cC(i,k) = -4,\ bC(i,k) = 1,\ cB(i,k) = 1.$$

Последняя строка первой блочной строки,

$i = 1,\ k = M$:

$$cC(i,k) = -4,\ aC(i,k) = 1,\ cB(i,k) = 1.$$

Внутренние строки последней блочной строки,

$i = N,\ 2 \le k \le M-1$:

$$cC(i,k) = -3,\ aC(i,k) = 1,\ bC(i,k) = 1,\ cA(i,k) = 1.$$

Первая строка последней блочной строки,

$i=N,\ k=1$:

$$cC(i,k)=-4,\ \ bC(i,k)=1,\ \ cA(i,k)=1.$$

Последняя строка последней блочной строки,

$i=N,\ k=M$:

$$cC(i,k)=-4,\ \ aC(i,k)=1,\ \ cA(i,k)=1.$$

Правая часть системы (1): $fslae(i,k)=0$, $1\le i\le N$, $1\le k\le M$. Решение системы:

$$\left(y_{1,1},...,y_{1,M},\ ...\ ,y_{N,1},...,y_{N,M}\right)=(1,\ 1,\ ...,\ 1).$$

Расширенная матрица системы (1) для данного примера изображена на рисунке для случая $N=4$, $M=4$ (рис 2.2):

$$\begin{bmatrix}
-4 & 1 & & & 1 & & & & & & & & & & & \\
1 & -3 & 1 & & & 1 & & & & & & & & & & \\
 & 1 & -3 & 1 & & & 1 & & & & & & & & & \\
 & & 1 & -4 & & & & 1 & & & & & & & & \\
1 & & & & -5 & 1 & & & 1 & & & & & & & \\
 & 1 & & & 1 & -4 & 1 & & & 1 & & & & & & \\
 & & 1 & & & 1 & -4 & 1 & & & 1 & & & & & \\
 & & & 1 & & & 1 & -5 & & & & 1 & & & & \\
 & & & & 1 & & & & -5 & 1 & & & 1 & & & \\
 & & & & & 1 & & & 1 & -4 & 1 & & & 1 & & \\
 & & & & & & 1 & & & 1 & -4 & 1 & & & 1 & \\
 & & & & & & & 1 & & & 1 & -5 & & & & 1 \\
 & & & & & & & & 1 & & & & -4 & 1 & & \\
 & & & & & & & & & 1 & & & 1 & -3 & 1 & \\
 & & & & & & & & & & 1 & & & 1 & -3 & 1 \\
 & & & & & & & & & & & 1 & & & 1 & -4
\end{bmatrix}$$

Рисунок 2.2 – Пример матрицы при $N, M = 4$

Метод GMRES гарантирует, что для блочно-трехдиагональной матрицы с размерами блоков N, M, можно получить точное решение за $N * M$ шагов алгоритма, но при больших размерах, требуемая память не меньше, чем $m^2 * N * M$, поэтому метод перезапускается с начальным приближением, полученным после прошлой работы алгоритма. Для начала покажем работу алгоритма без перезапусков и количеством шагов равным размеру $N * M$ (рис. 2.3):

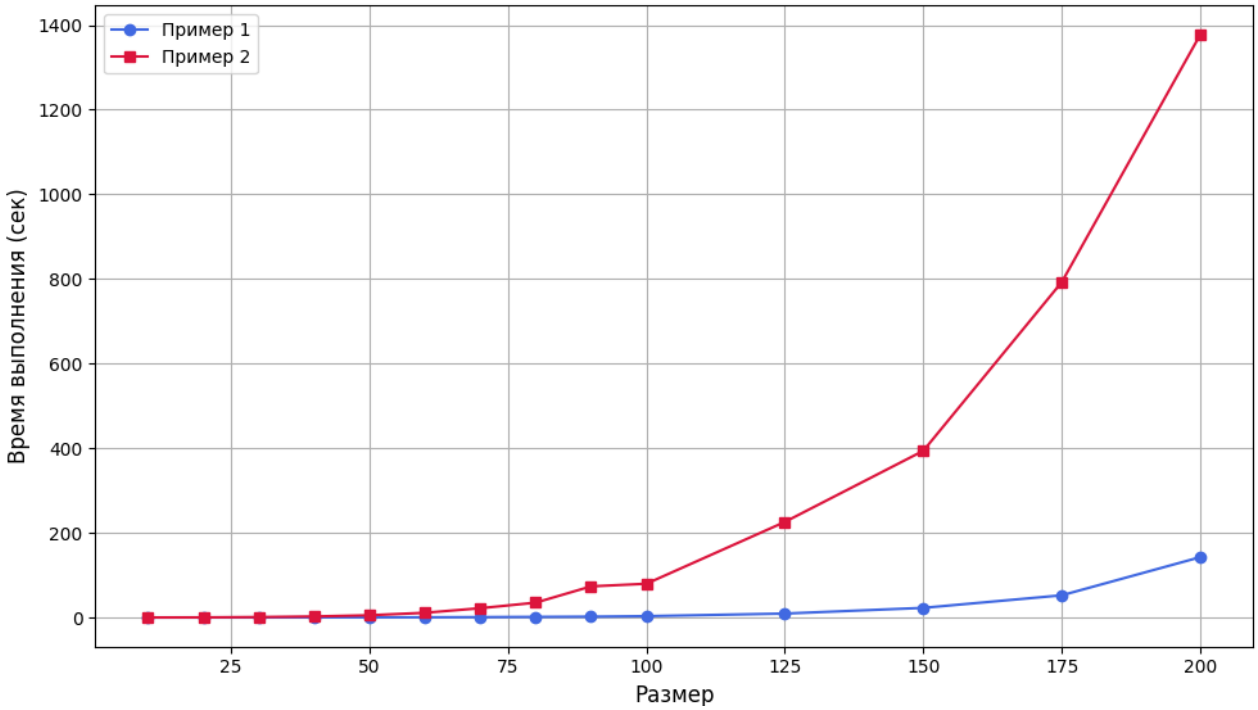

Рисунок 2.3 – Время работы алгоритма

По результатам графика видно, что время выполнения слишком велико, поэтому необходимо останавливать алгоритм, когда невязка будет меньше требуемой точности, например: 10^{-2}, $10^{-4}, 10^{-6}$. Покажем скорость сходимости алгоритма на примерах 1,2 при $N, M = 150$ (рис. 2.4, рис. 2.5):

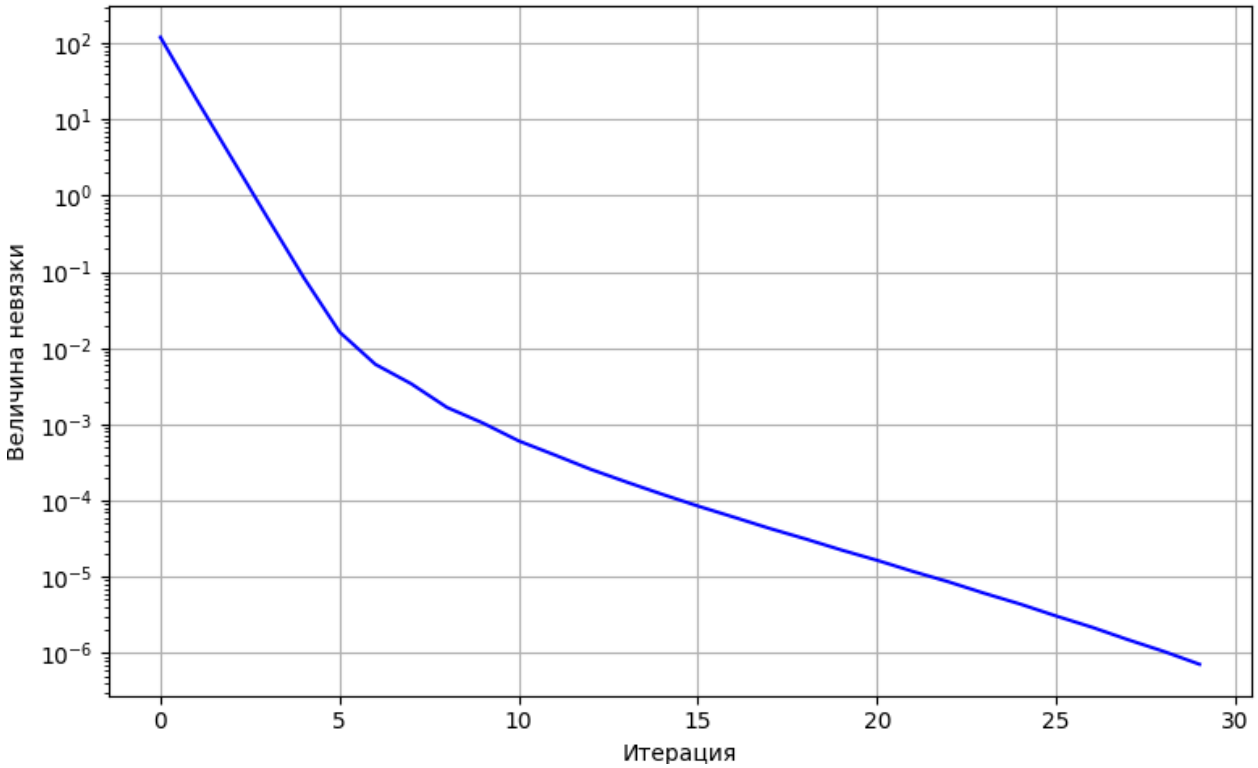

Рисунок 2.4 – Уменьшение невязки по итерациям пример 1

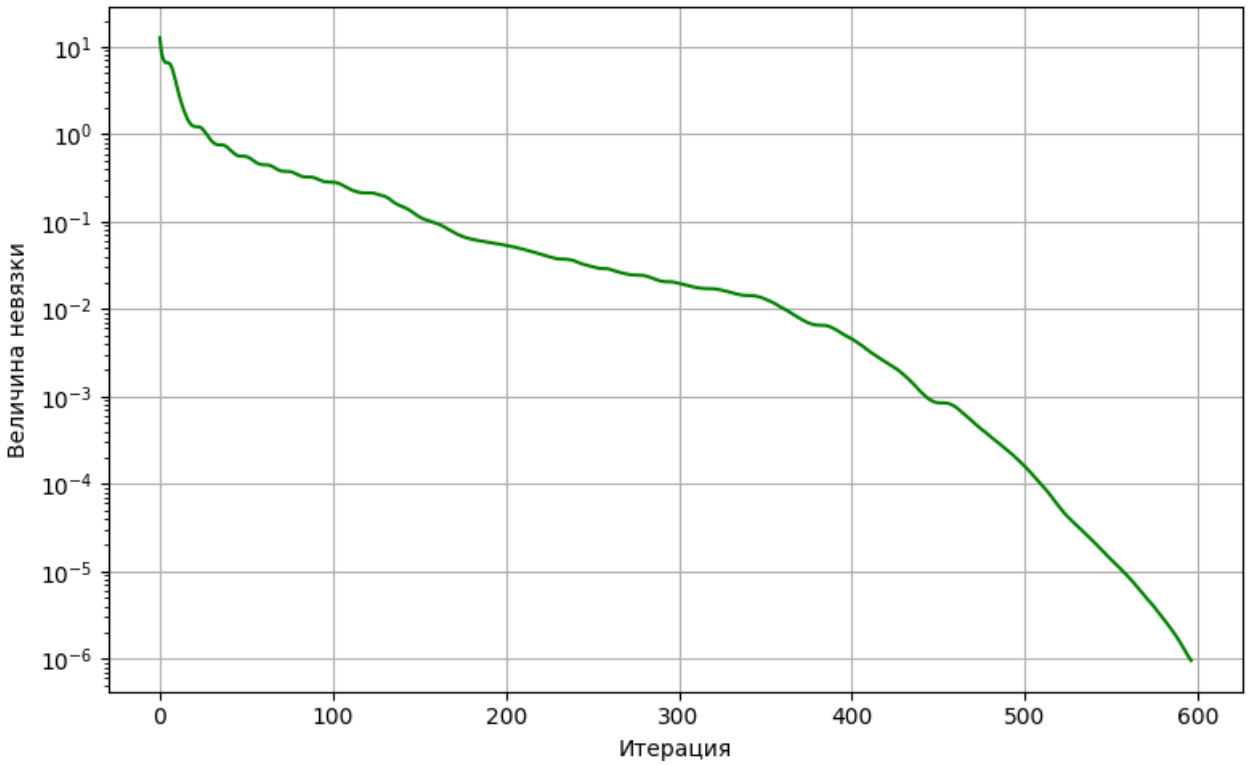

Рисунок 2.5 – Уменьшение невязки по итерациям пример 2

Из рисунка 2.4 и рисунка 2.5 видно, что количество шагов алгоритма для достижения точности 10^{-6} меньше чем $m * m$. Рассмотрим пример 1:

Покажем время работы при перезапусках алгоритма и количестве итераций равных m=5, 10, 20, точность(10^{-6}):

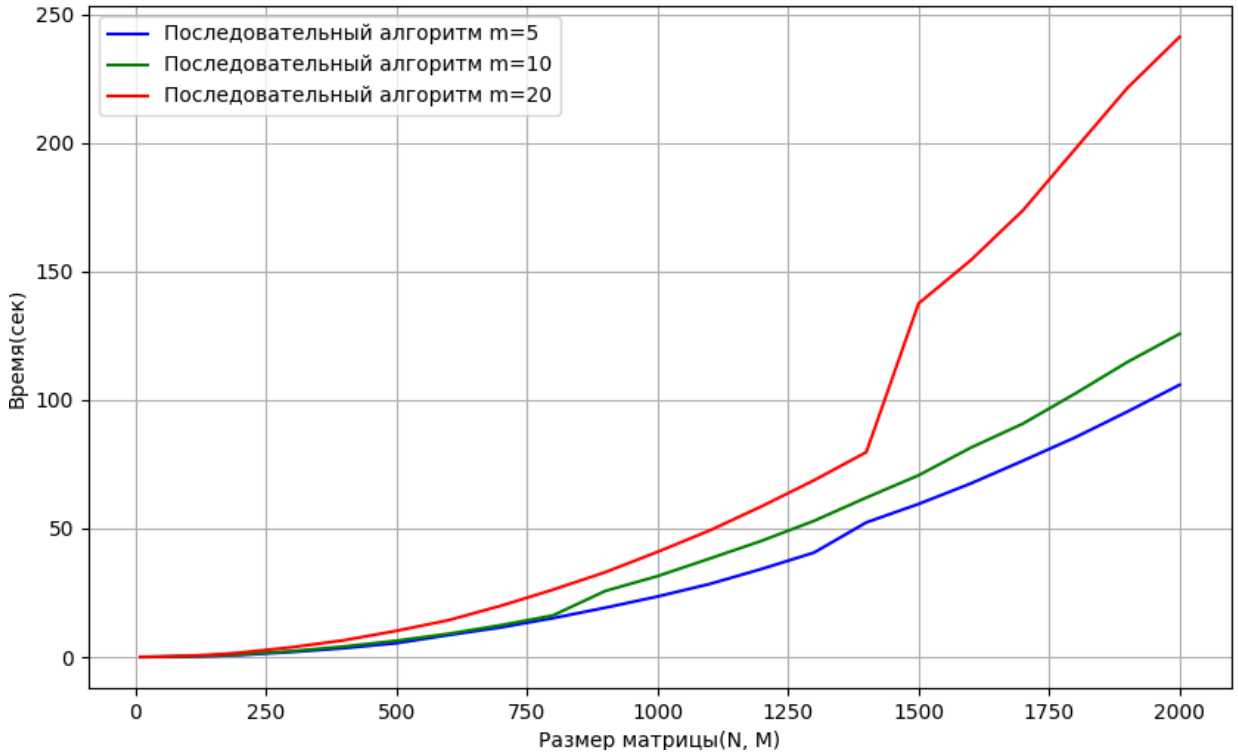

Рисунок 2.6 – Зависимость времени работы от размера матрицы при перезапусках

Количество рестартов для достижения заданной точности(10^{-6}):

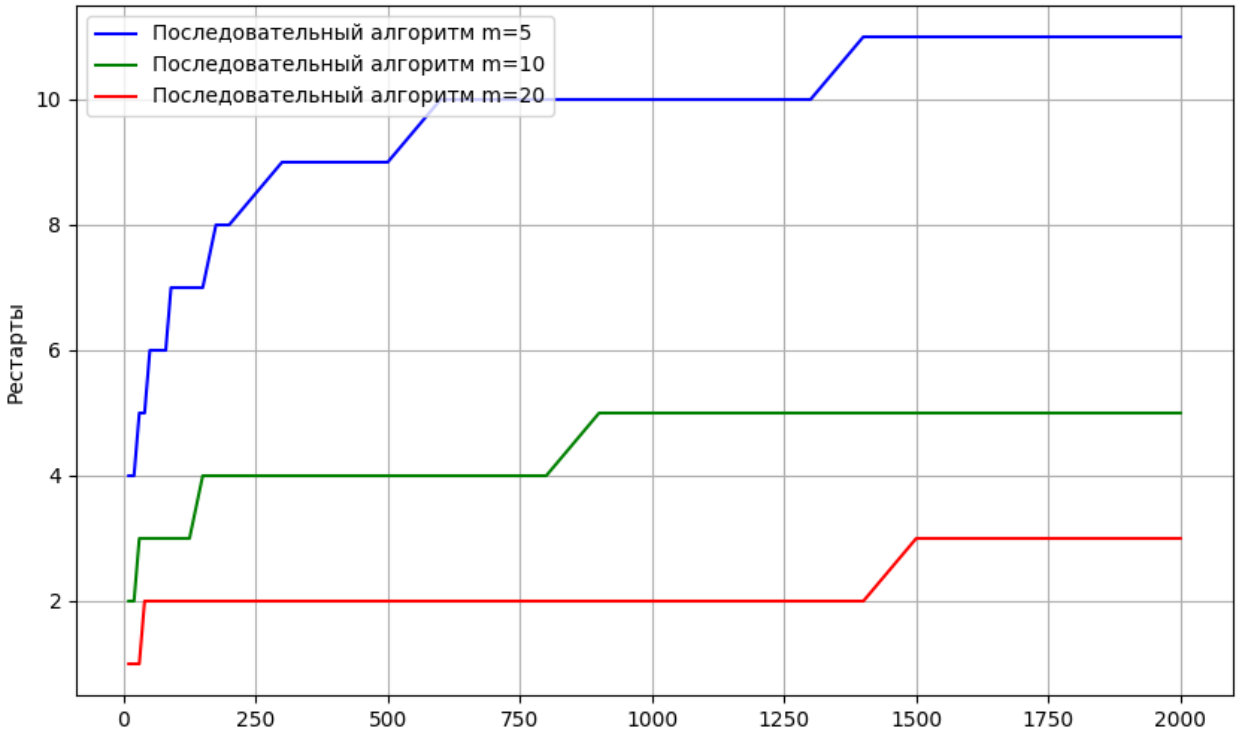

Рисунок 2.7 – Количество рестартов

Покажем влияние на время работы алгоритма, останавливая его при достижении заданной точности (10^{-6}):

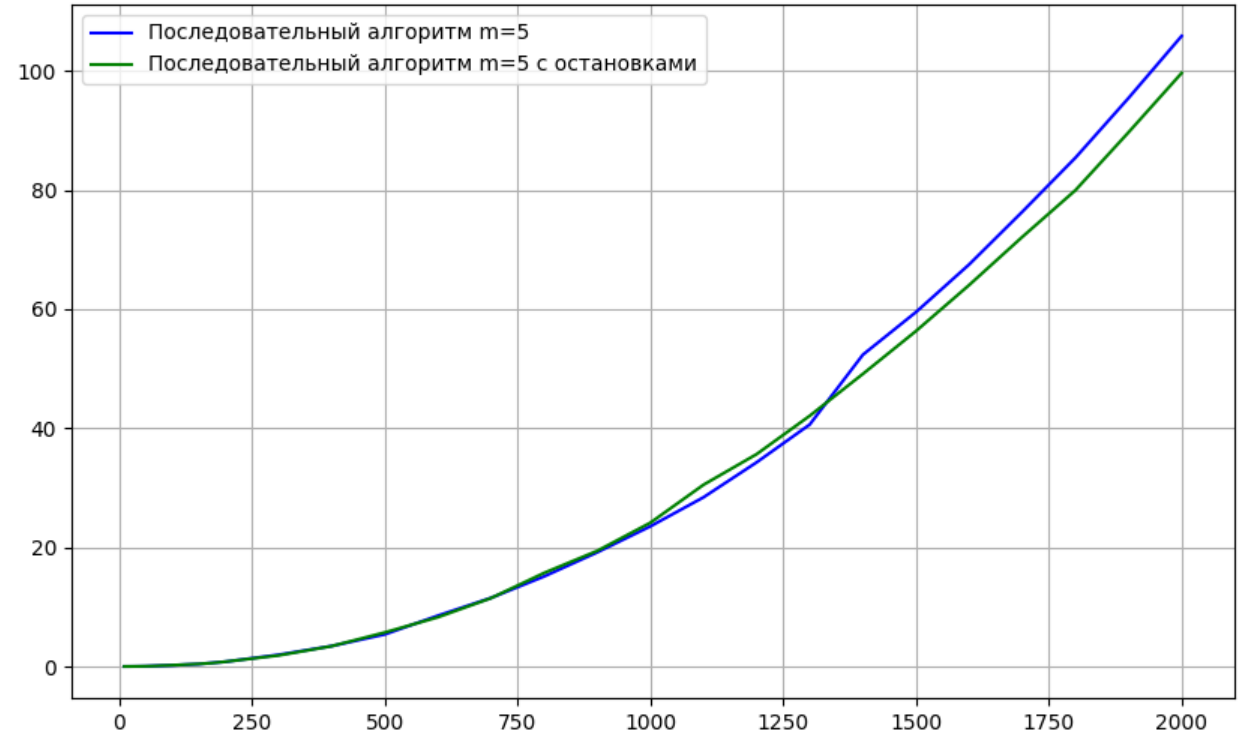

Рисунок 2.8 – Влияние достижения заданной точности на время работы алгоритма

Исследуя данные рисунки, можно сделать вывод – алгоритм быстрее всего работает при m=5 и остановках при достижении заданной точности.

Теперь рассмотрим пример 2, покажем время работы при перезапусках алгоритма и разной размерности подпространства Крылова m при N, M = 150, точность 10^{-6} (рис 2.9):

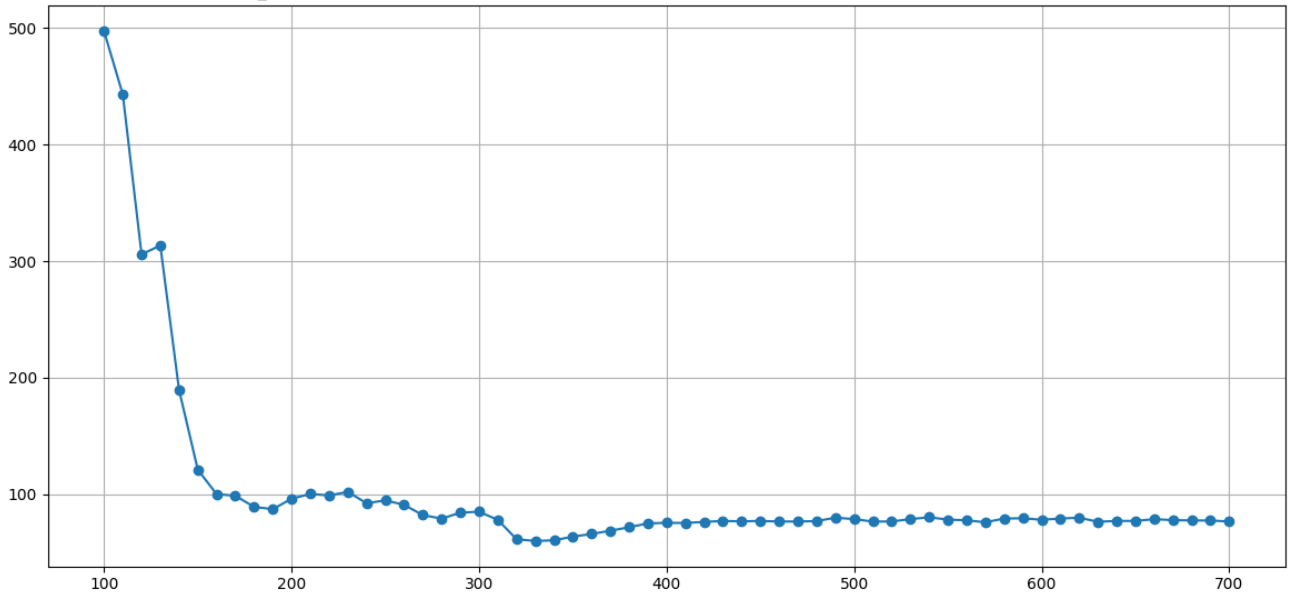

Рисунок 2.9 – Время работы при разных значениях m

Найдем наиболее эффективную размерность подпространства m при различных N, M (рис 2.10)

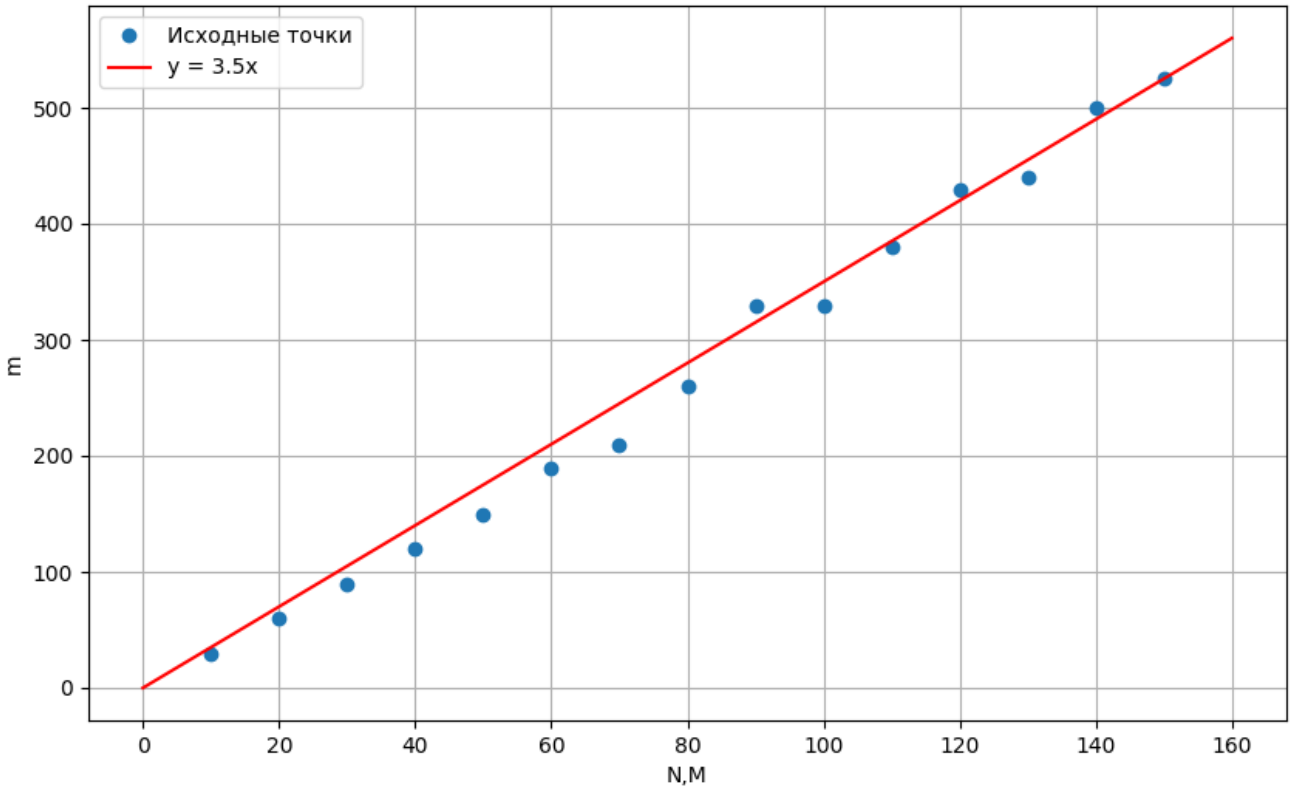

Рисунок 2.10 – Оптимальные значения m для различных N, M

Исходя из рисунка 2.9 можно заметить, что наиболее эффективный размер подпространства m = $3.5 * N$, при условии, что $N = M$.

Рассмотрев два примера, можно сделать вывод, что для матриц с большим диагональным преобладанием использование меньшей размерности подпространства Крылова является наиболее эффективным расчётом, но при уменьшении диагонального преобладания, для большей эффективности алгоритма и его сходимости, следует выбирать больший размер подпространства.

2.4 Параллельный алгоритм GMRES

Параллельный алгоритм – это алгоритм, способный одновременно выполнять обработку нескольких инструкций с использованием множества процессов (или потоков), работающих на различных вычислительных устройствах. Результаты этих параллельных вычислений затем объединяются для получения итогового решения.

Использование параллелизма в алгоритмах позволяет значительно повысить производительность вычислений. Параллельное программирование способствует более эффективному использованию вычислительных ресурсов.

Большинство современных вычислительных систем оснащены многопроцессорными или многоядерными компонентами, а также поддержкой многопоточности. Это позволяет запускать несколько потоков одновременно и тем самым раскрывать потенциал оборудования на полную мощность, обеспечивая ускоренное выполнение задач.

Однако, параллельные алгоритмы сложнее внедрить, их сложнее отладить или доказать правильность, и при малой сложности задачи они часто работают хуже, чем их последовательные аналоги, из-за накладных расходов на связь и координацию потоков.

Самой высоко нагруженной частью алгоритма являются действия умножения матрицы на вектор, вектор на вектор, вектор на константу, матрицу базисов на вектор.

Обновим леммы для параллельного выполнения:

Лемма 1*. Пусть A – матрица системы вида (1), где B_0, A_N – нулевые, а C_0, C_N – единичные матрицы, X – вектор вида (2). Тогда элементы вектора $Z=AX$,

$Z = (Z_1, ..., Z_N) = (z_{1,1}, ..., z_{1,M}, \ ... \ , z_{N,1}, ..., z_{N,M})$ могут быть вычислены следующим образом:

$Z_0 = X_0$

dopar $i = 1, N$

if $i=1$ do

$z_{i,1} = c_1^{C_i} x_{i,1} + b_1^{C_i} x_{i,2} + c_1^{B_i} x_{i+1,1}$

dopar $k = 2, M–1$

$z_{i,k} = a_k^{C_i} x_{i,k-1} + c_k^{C_i} x_{i,k} + b_k^{C_i} x_{i,k+1} + c_k^{B_i} x_{i+1,k}$

enddo

$z_{i,M} = a_M^{C_i} x_{i,M-1} + c_M^{C_i} x_{i,M} + c_M^{B_i} x_{i+1,M}$

endif($i=1$)

if $i\neq1, i\neq N$ do

$z_{i,1} = c_1^{A_i} x_{i-1,1} + c_1^{C_i} x_{i,1} + b_1^{C_i} x_{i,2} + c_1^{B_i} x_{i+1,1}$

dopar $k = 2, M–1$

$z_{i,k} = c_k^{A_i} x_{i-1,k} + a_k^{C_i} x_{i,k-1} + c_k^{C_i} x_{i,k} + b_k^{C_i} x_{i,k+1} + c_k^{B_i} x_{i+1,k}$

enddo

$z_{i,M} = c_M^{A_i} x_{i-1,M} + a_M^{C_i} x_{i,M-1} + c_M^{C_i} x_{i,M} + c_M^{B_i} x_{i+1,M}$

endif($i\neq1, i\neq N$)

if $i=N$ do

$z_{i,1} = c_1^{A_i} x_{i-1,1} + c_1^{C_i} x_{i,1} + b_1^{C_i} x_{i,2}$

dopar $k = 2, M–1$

$z_{i,k} = c_k^{A_i} x_{i-1,k} + a_k^{C_i} x_{i,k-1} + c_k^{C_i} x_{i,k} + b_k^{C_i} x_{i,k+1}$

enddo

$z_{i,M} = c_M^{A_i} x_{i-1,M} + a_M^{C_i} x_{i,M-1} + c_M^{C_i} x_{i,M}$

endif($i=N$)

enddo

Лемма 2*. Скалярное произведение векторов вида (2), (3) может быть вычислено следующим образом:

dopar $i =1, N$

$s_i = x_{i,1} z_{i,1}$

do $k = 2, M–1$

$s_i = s_i + x_{i,k} z_{i,k}$

enddo

enddopar

$xz = s_1$

dopar $i = 1, N$

$xz = xz + s_i$

enddopar

Лемма 3*. Умножение вектора вида (2) на константу c может быть вычислено следующим образом:

dopar $i = 1, N$

do $k = 1, M$

$z_{i,k} = x_{i,k} * c$

enddopar

enddopar

Лемма 4*. Пусть V – матрица размера $NM \times m$, y – m-мерный вектор. Тогда элементы вектора $t=Vy$,

$$(T_1,...,T_N) = (t_{1,1},...,t_{1,M}, \ ... \ ,t_{N,1},...,t_{N,M}),$$

могут быть вычислены следующим образом:

Вход:

трёхмерный массив $v(\beta,i,j)$, $1 \le \beta \le m$, $0 \le i \le N$, $1 \le k \le M$

Выход:

двумерный массив $t(i,j)$, $0 \le i \le N$, $1 \le k \le M$

dopar $i = 0, N$

dopar $k = 1, M$

$t(i,j) = 0$

do $\beta = 1, m$

$t(i,k) = t(i,k) + v(\beta,i,k)y(\beta)$

enddo

enddopar

enddopar

2.5 OpenMP

OpenMP представляет собой интерфейс прикладного программирования (API), предназначенный для организации многопоточного параллелизма в приложениях с общей памятью[5]. Он основан на использовании директив препроцессора, встроенных непосредственно в код программы, что позволяет компилятору автоматически распараллеливать выполнение задач.

OpenMP реализует модель параллелизма «fork-join». Программа начинает выполнение в одном потоке. При достижении параллельной области главный поток создаёт группу дополнительных потоков – это фаза

fork. Эти потоки совместно выполняют параллельный участок кода, при этом главный поток также участвует в вычислениях. После завершения параллельного участка все потоки синхронизируются и выполнение продолжается в одном потоке – это фаза *join*.

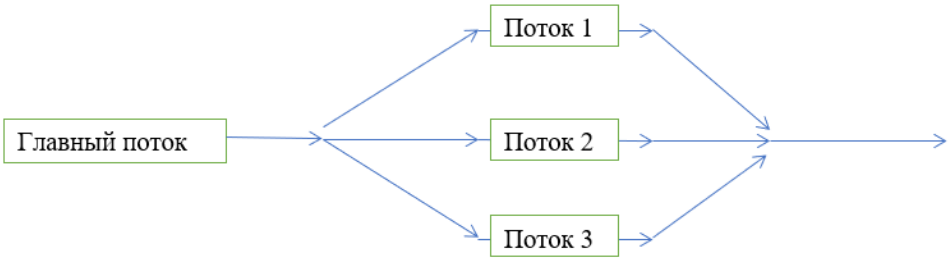

Рисунок 2.11 – Модель параллелизм

Создание новых потоков затрачивает время и ресурсы, поэтому при выборе количество параллельных процессов необходимо учитывать сложность решаемой задачи. Также для высокого показателя эффективности программы необходимо, чтобы каждый из потоков был нагружен равномерной работой. Необходимо избегать наличия данных, к которым осуществляется одновременный несогласованный доступ, чтобы избежать конфликтов при распределении данных.

ГЛАВА 3
ВЫЧИСЛИТЕЛЬНЫЕ ЭКСПЕРИМЕНТЫ

Покажем зависимость времени выполнения алгоритма от размеров матрицы системы при применении параллелизма, рассматривая матрицу из примера 1 (рис 3.1):

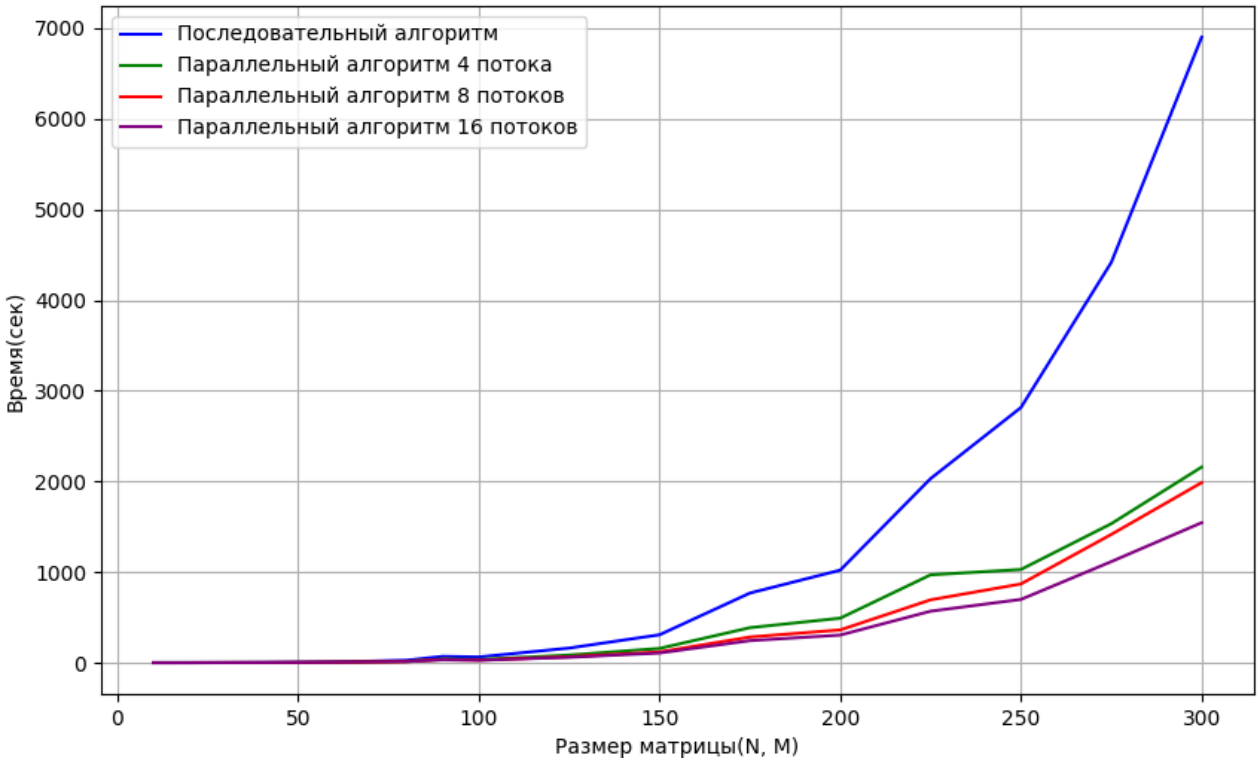

Рисунок 3.1 – Время работы алгоритма при N, M от 10 до 3000

Полный результат работы алгоритмов предоставлен в приложении В.

Покажем ускорение, которое обеспечивает параллельное выполнение (рис 3.2)

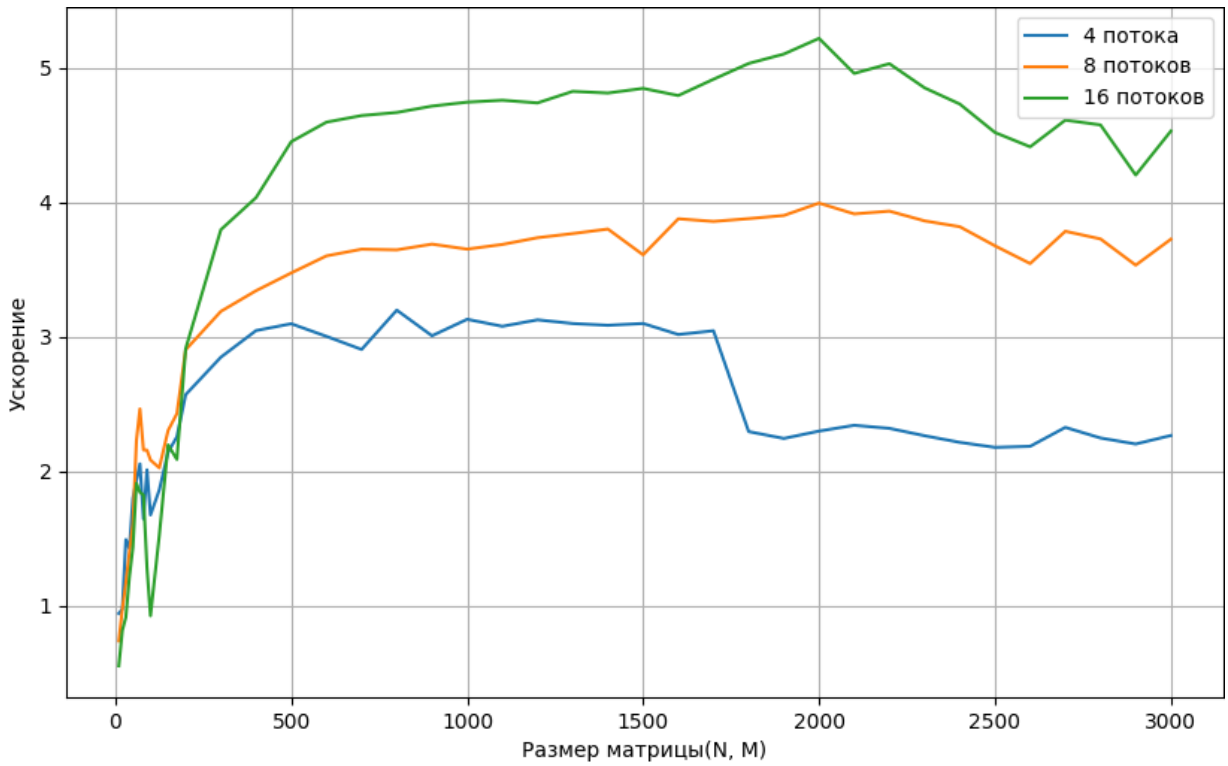

Рисунок 3.2 – Ускорение алгоритма при разном количестве потоков

Покажем зависимость времени выполнения алгоритма от размеров матрицы системы при применении параллелизма, рассматривая матрицу из примера 2(рис 3.4):

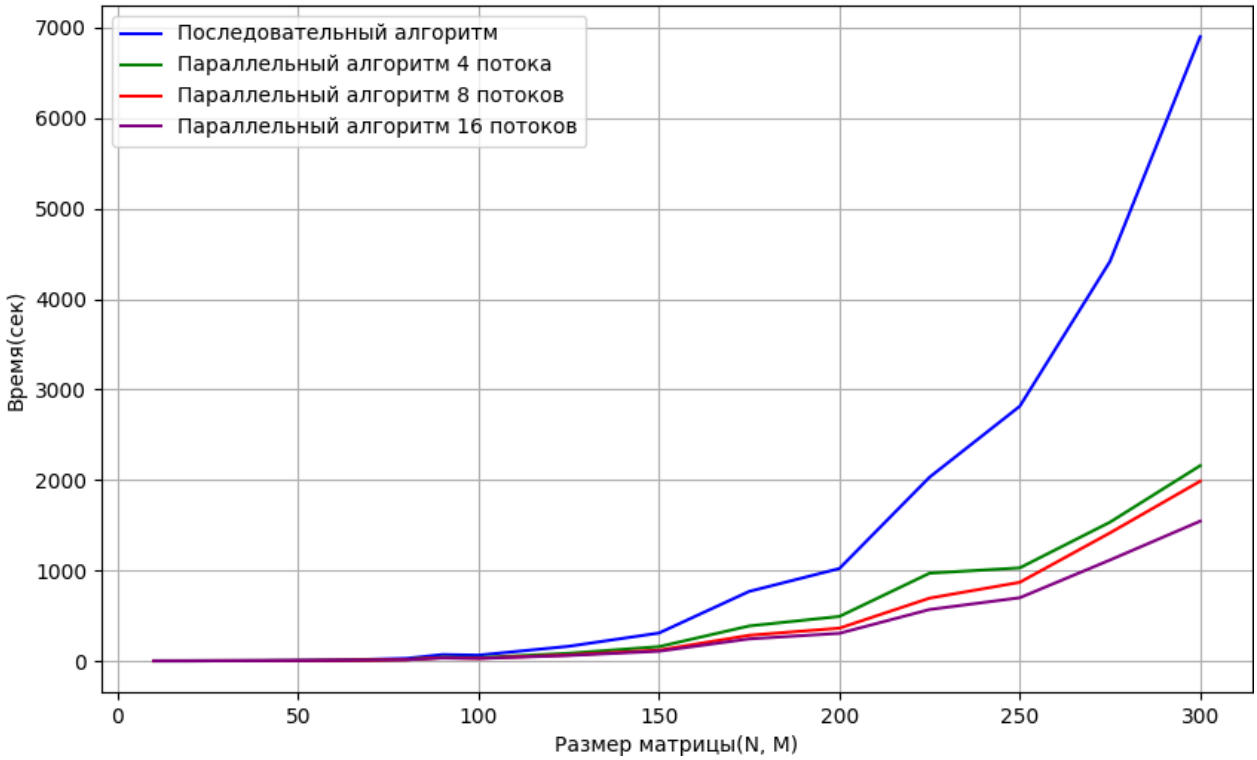

Рисунок 3.3 – Время работы алгоритма при N, M от 10 до 300

Полный результат работы алгоритмов предоставлен в приложении С.

Покажем ускорение, которое обеспечивает параллельное выполнение (рис 3.5)

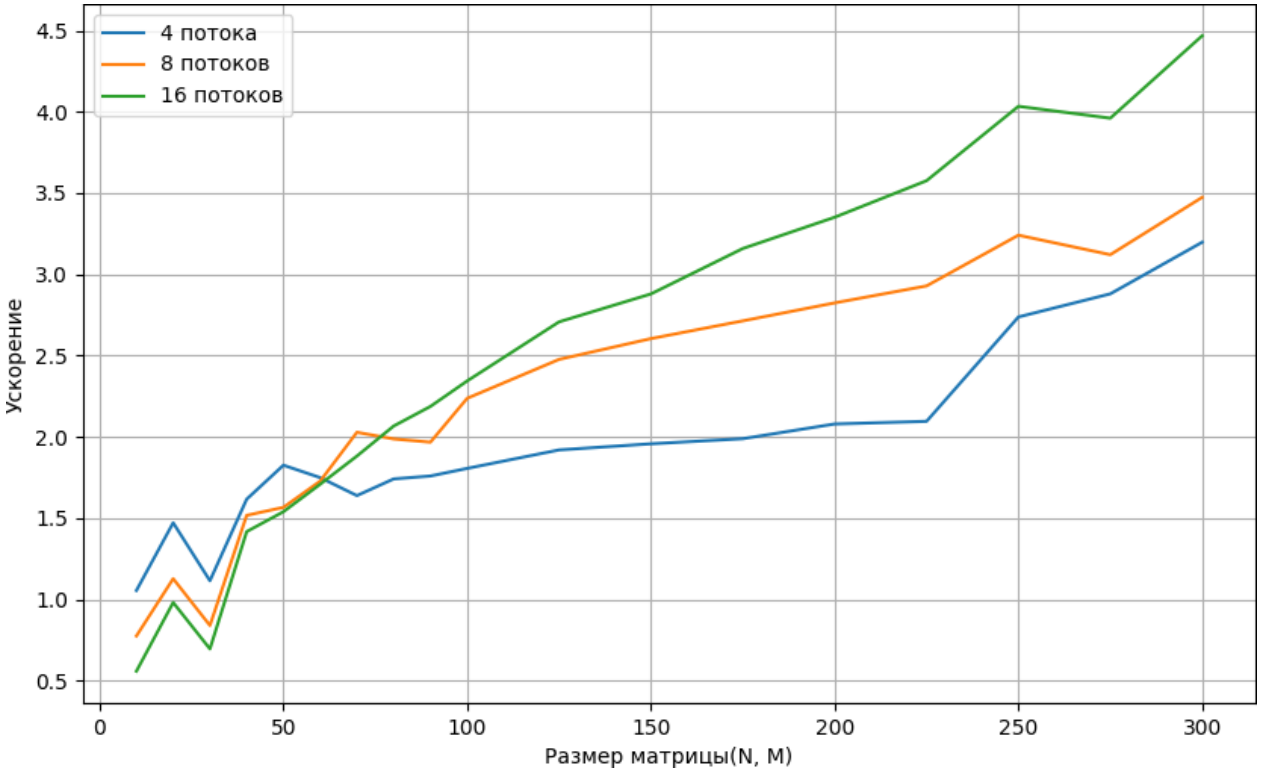

Рисунок 3.4 – Ускорение алгоритма при разном количестве потоков

Из рисунков видно, что применение параллелизма не эффективно при малых размерах блоков N, M. Это связано с накладными расходами при создании потоков. Применение параллелизма в алгоритме с помощью 4 и 8 потоков показало значительное ускорение при размерах блоков ($N, M > 200$), параллелизм с 8 потоками становится эффективнее параллелизма с 4 потоками при ($N, M > 125$), однако параллелизм с 16 потоками всё ещё уступает алгоритму с 8 потоками при $N, M < 100$.

ЗАКЛЮЧЕНИЕ

В ходе выполнения дипломной работы был исследован и реализован обобщенный метод минимальных невязок (GMRes) для решения систем уравнений специального вида, представленных блочно-трёхдиагональными матрицами с трёхдиагональными блоками.

На основе изучения литературных источников для таких систем уравнений был разработан алгоритм GMRes, который затем был программно реализован в виде последовательной программы. Проведены вычислительные эксперименты, в результате которых было выявлено время выполнения алгоритма при различных его параметрах.

Кроме того, были изучены основные принципы параллельного программирования для многоядерных процессоров, а также разработан и реализован параллельный алгоритм GMRes с использованием технологии OpenMP. Проведены дополнительные вычислительные эксперименты, позволившие оценить время выполнения и ускорение вычислений при различных параметрах алгоритма.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Саад, Ю. Итерационные методы для разреженных линейных систем / Ю. Саад. – Москва : Изд-во Моск. ун-та, 2013. – 324 с.

2. Vorst, H. A. van der. Iterative Krylov methods for large linear systems / H. A. van der Vorst. – New York : Cambridge Univ. Press, 2003. – 221 p.

3. Гергель, В. П. Высокопроизводительные вычисления для многопроцессорных многоядерных систем / В. П. Гергель. – Москва : Б-ка Нижегор. гос. ун-та им. Н. И. Лобачевского, 2010. – 539 с.

4. Скалярное произведение векторов, вещественная версия, последовательно-параллельный вариант [сайт]. – URL: http://algowiki-project.org (дата обращения: 05.05.2025).

5. The OpenMP API specification for parallel programming [сайт]. – URL: https://www.openmp.org/ (дата обращения: 10.05.2025).

```
vector<vector<double>> MatrixVectorMultiplication(Matrix& mat,
vector<vector<double>>& X)//lemma1
{
   vector<vector<double>> Z(X.size(), vector<double>(X[0].size(), 0));
#pragma omp parallel for num_threads(global_th)
      for (int j = 1; j <= mat.M - 1; j++) {
         Z[0][j] = X[0][j];
         Z[mat.N][j] = X[mat.N][j];
      }


#pragma omp parallel for num_threads(global_th)
      for (int i = 1; i <= mat.N - 1; i++) {
         Z[i][1] = mat.cA_vector[i][1] * X[i - 1][1] + mat.cC_vector[i][1] * X[i][1]
+ mat.bC_vector[i][1] * X[i][2] + mat.cB_vector[i][1] * X[i + 1][1]
+mat.bA_vector[i][1] * X[i - 1][2] + mat.bB_vector[i][1] * X[i + 1][2];

         for (int j = 2; j <= mat.M - 2; j++) {
            Z[i][j] = mat.cA_vector[i][j] * X[i - 1][j] + mat.aC_vector[i][j] * X[i][j
- 1] + mat.cC_vector[i][j] * X[i][j] + mat.bC_vector[i][j] * X[i][j + 1] +
mat.cB_vector[i][j] * X[i + 1][j] +mat.bB_vector[i][j] * X[i + 1][j + 1] +
mat.aA_vector[i][j] * X[i - 1][j - 1] + mat.bA_vector[i][j] * X[i - 1][j + 1] +
mat.aB_vector[i][j] * X[i + 1][j - 1];
         }

         Z[i][mat.M - 1] = mat.cA_vector[i][mat.M - 1] * X[i - 1][mat.M - 1] +
mat.aC_vector[i][mat.M - 1] * X[i][mat.M - 2] + mat.cC_vector[i][mat.M - 1] *
X[i][mat.M - 1] + mat.cB_vector[i][mat.M - 1] * X[i + 1][mat.M - 1]
+mat.aB_vector[i][mat.M - 1] * X[i + 1][mat.M - 2] + mat.aA_vector[i][mat.M -
1] * X[i - 1][mat.M - 2];
      }


   return Z;

}

double ScalarProduct(vector<vector<double>>& X, vector<vector<double>>&
Z)//lemma2
{
   vector<double> s(X.size());
```

```
    double XZ;


#pragma omp parallel for num_threads(global_th) collapse(2)
        for (int i = 0; i < X.size(); i++) {
            s[i] = X[i][1] * Z[i][1];

            for (int j = 2; j < Z[0].size(); j++) {
                s[i] += X[i][j] * Z[i][j];
            }
        }

    XZ = s[0];
#pragma omp parallel for reduction(+:XZ)
    for (int i = 1; i < s.size(); i++) {
        XZ += s[i];
    }
    return XZ;
}

vector<vector<double>> VectorConst(double a, vector<vector<double>>& b)
{//lemma3
    vector<vector<double>> z(b.size(), vector<double>(b[0].size()));

#pragma omp parallel for num_threads(global_th) collapse(2)
        for (int i = 0; i < b.size(); i++)
        {
            z[i][1] = b[i][1] * a;

            for (int j = 2; j < b[i].size(); j++)
            {
                z[i][j] = a * b[i][j];
            }
        }

    return z;
}


vector<vector<double> > MatrixVxy(vector<vector<vector<double>>>& v,
vector<double> y,  int m) {

    vector<vector<double> >w(v[0].size(), vector<double>(v[0][0].size(), 0));
```

```
#pragma omp parallel for num_threads(global_th)
        for (int i = 0; i <= v[0].size() - 1; i++)
        {

            for (int j = 1; j <= v[0][0].size() - 1; j++)
            {
                w[i][j] = 0;
                for (int beta = 1; beta <= m; beta++) {
                    w[i][j] = w[i][j] + v[beta][i][j] * y[beta];
                }
            }
        }

    return w;
}


vector<vector<double>> operator - (vector<vector<double>>& a,
vector<vector<double>>& b) {//X-X(разность векторов)
    vector<vector<double>> result(b.size(), vector<double>(b[0].size()));
    if (a.size() != b.size()) {
        throw "bad size";
    }
#pragma omp parallel for num_threads(global_th) collapse(2)
    for (int i = 0; i < a.size(); i++)
        for (int j = 0; j < b[i].size(); j++)
            result[i][j] = a[i][j] - b[i][j];

    return result;
}
vector<vector<double>> operator + (vector<vector<double>>& a,
vector<vector<double>>& b) {//X-X(разность векторов)
    vector<vector<double>> result(b.size(), vector<double>(b[0].size()));
    if (a.size() != b.size()) {
        throw "bad size";
    }
#pragma omp parallel for num_threads(global_th) collapse(2)
    for (int i = 0; i < a.size(); i++)
        for (int j = 0; j < b[i].size(); j++)
            result[i][j] = a[i][j] + b[i][j];

    return result;
}
```

```
void GMRES(Matrix& A, vector<vector<double>>& xm,
vector<vector<double>>& fslae, int& m, int N, int M, double& rfinish, int&
maxstart, double epsilon) {

   int start = 0;
START:
   start = start + 1;
   while (start <= maxstart) {

      vector<vector<double> > h(m + 2, vector<double>(m + 2, 0));
      vector<vector<double> > r(m + 2, vector<double>(m + 2, 0));
      vector<vector<vector<double>>> v(m + 2, vector<vector<double>>(N + 1,
vector<double>(M, 0)));
      vector<double > b(m + 2, 0);
      vector<vector<double>> tempv1 = MatrixVectorMultiplication(A,
xm);//вектор A*xm
      vector<vector<double>> tempv2 = tempv1 - fslae; // r = Ax - f

      vector<double>  c(m + 2, 0);
      vector<double>  s(m + 2, 0);
      int nr = 0;
      double scal_tempv2_tempv2 = ScalarProduct(tempv2, tempv2);//(r0,r0)
скалярное произведение

      double enres = sqrt(scal_tempv2_tempv2);//евклидова норма невязки

      b[1] = enres;
      double enres1 = 1 / enres;
      v[1]= VectorConst(enres1, tempv2);




      for (int beta = 1; beta <= m; beta++)
      {
         tempv2 = MatrixVectorMultiplication(A, v[beta]);
         for (int alpha = 1; alpha <= beta; ++alpha)
         {
            double scal_v_alpha_tempv2 = ScalarProduct(tempv2, v[alpha]);
            h[alpha][beta] = scal_v_alpha_tempv2;
            tempv1 = VectorConst(scal_v_alpha_tempv2, v[alpha]);
            tempv2 = tempv2 - tempv1;
         }
```

```
            double scal_tempv2_tempv2 = ScalarProduct(tempv2, tempv2);
            h[beta + 1][beta] = sqrt(scal_tempv2_tempv2);
            double h1 = 1 / h[beta + 1][beta];

            v[beta + 1] = VectorConst(h1, tempv2);//v(beta+1)= w_beta/
h(beta+1,beta)
            r[1][beta] = h[1][beta];
            for (int alpha = 2; alpha <= beta; ++alpha) {
               double temp = c[alpha - 1] * r[alpha - 1][beta] + s[alpha - 1] *
h[alpha][beta];
               r[alpha][beta] = -s[alpha - 1] * r[alpha - 1][beta] + c[alpha -
1]*h[alpha][beta];
               r[alpha - 1][beta] = temp;
            }
           double delta = sqrt(r[beta][beta] * r[beta][beta] + h[beta + 1][beta] * h[beta
+ 1][beta]);
            c[beta] = r[beta][beta] / delta;
            s[beta] = h[beta + 1][beta] / delta;
            r[beta][beta] = c[beta] * r[beta][beta] + s[beta] * h[beta + 1][beta];
            b[beta + 1] = -s[beta] * b[beta];
            b[beta] = c[beta] * b[beta];
           if (abs(b[beta + 1]) <= epsilon){
               nr = beta;  m = beta;
               goto Gauss;
               cout << m << endl;
            }

         }
         nr = m;

      Gauss:
         vector<double> y(nr + 1, 0);
         y[nr] = b[nr] / r[nr][nr];
         for (int alpha = nr - 1; alpha >= 1; alpha--) {
            y[alpha] = b[alpha];
            for (int beta = alpha + 1; beta <= nr; beta++) {
               y[alpha] = y[alpha] - r[alpha][beta] * y[beta];
            }
            y[alpha] = y[alpha] / r[alpha][alpha];
         }
         tempv1= MatrixVxy(v, y, nr);

         xm = xm - tempv1;
         rfinish = abs(b[nr + 1]);
```

```
        goto START;

    }

}
#include<vector>
#include <random>
using namespace std;
double element(int i, int k, int N, int M)
{
        return (2.0 * (double)i + (double)k) / (2.0 * (double)N + (double)M);
}

class GemoDin
{
public:
        int N;
        int M;
        vector<vector<double> >  cC_vector;
        vector<vector<double> > aC_vector;
        vector<vector<double> > bC_vector;
        vector<vector<double> > cA_vector;
        vector<vector<double>> cB_vector;
double element2(int i, int k, int N, int M)
{
        return (2.0 * (double)i + (double)k) / (2.0 * (double)N + (double)M);
}

        //Формирование матрицы
        GemoDin(int N, int M)
        {
                this->N = N;
                this->M = M;
                //cC
                cC_vector.resize(N + 1, vector<double>(M, 0));
                for (int i = 0; i <= N; i++) {
                        for (int j = 1; j <= M - 1; j++)
                        {
                                        cC_vector[i][j] = 8;
```

```
                }
            }

            //aC
            aC_vector.resize(N + 1, vector<double>(M, 0));
            for (int i = 0; i <= N; i++) {
                for (int k = 2; k <= M - 1; k++)
                {

                    aC_vector[i][k] = element2(i, k, N, M);
                }
            }

            //bC
            bC_vector.resize(N + 1, vector<double>(M - 1, 0));
            for (int i = 0; i <= N; i++) {
                for (int k = 1; k <= M - 2; k++)
                {
                    bC_vector[i][k] = element2(i, k, N, M);

                }
            }

            //cA
            cA_vector.resize(N + 1, vector<double>(M, 0));
            for (int i = 1; i <= N; i++) {
                for (int k = 1; k <= M - 1; k++)
                {

                    cA_vector[i][k] = element2(i, k, N, M);

                }
            }
        //cB
            cB_vector.resize(N, vector<double>(M, 0));
            for (int i = 0; i <= N - 1; i++) {
                for (int k = 1; k <= M - 1; k++)
                {

                        cB_vector[i][k] = element2(i, k, N, M);
                }
            }
        }
};
```

ПРИЛОЖЕНИЕ Б

Результаты вычислительных экспериментов

N, M	Время исполнения (последовательный алгоритм), с	Время исполнения (параллельный алгоритм, 4 потока), с	Время исполнения (параллельный алгоритм, 8 потоков), с	Время исполнения (параллельный алгоритм, 16 потоков), с
10	0.004738	0.004826	0.004569	0.006749
20	0.008906	0.00633	0.008783	0.010428
30	0.016037	0.011451	0.013808	0.017963
40	0.029304	0.018498	0.017125	0.031474
50	0.04287	0.024851	0.026384	0.03437
60	0.066908	0.039033	0.03043	0.04324
70	0.093559	0.052577	0.034679	0.046
80	0.154156	0.066796	0.05536	0.084765
90	0.163504	0.081188	0.079549	0.155076
100	0.21639	0.129991	0.137871	0.119744
125	0.318694	0.177891	0.15574	0.162824
150	0.423082	0.243659	0.185669	0.18904
200	0.801242	0.364738	0.311106	0.260791
300	1.85165	0.615805	0.597458	0.479384
400	3.41157	1.10831	1.009	0.829418
500	5.70759	1.7512	1.60794	1.24728
600	8.26449	1896738	2.24939	1.93239
700	11.4729	5.0213	3.28356	2.44248
800	15.7041	6.84194	4.1578	3.20713
900	19.4114	8.48894	5.44509	4.08435
1000	24.1352	10.3749	6.56632	5.29729
1100	30.5079	12.3995	7.9743	6.16754
1200	35.6489	15.6523	9.4582	7.41747
1300	42.0919	14.0646	11.2023	8.42161
1400	49.1227	21.5589	13.0594	10.0954
1500	56.3241	26.723	14.738	11.4255
1600	64.0433	29.9216	16.4477	12.8107
1700	72.1222	31.5482	18.6147	14.1316
1800	79.9602	35.0849	20.8491	15.8462
1900	89.6875	30.2196	23.2047	17.7987
2000	99.6494	42.9552	25.6923	19.4017
2100	109.979	43.8547	29.8798	23.4578
2200	121.092	38.2966	31.9113	25.0141
2300	131.164	41.2687	34.6881	27.1873
2400	145.675	45.9193	38.8453	32.2068
2500	159.895	52.2898	44.0751	40.453
2600	173.85	58.0016	48.54	39.6167
2700	184.946	58.9428	48.2095	38.7634
2800	198.59	64.3583	52.4408	42.4683
2900	218.158	72.9143	59.8803	48.6888
3000	228.996	74.4226	60.3184	48.1755

ПРИЛОЖЕНИЕ В

Результаты вычислительных экспериментов

N, M	Время исполнения (последовательный алгоритм), с	Время исполнения (параллельный алгоритм, 4 потока), с	Время исполнения (параллельный алгоритм, 8 потоков), с	Время исполнения (параллельный алгоритм, 16 потоков), с
10	0.019839	0.018805	0.025596	0.035527
20	0.214111	0.145525	0.189923	0.218319
30	0.762951	0.684411	0.908703	1.09706
40	2.12471	1.31439	1.40067	1.49952
50	4.58635	2.51236	2.92811	2.97941
60	8.99744	5.1439	5.20795	5.2665
70	16.4779	10.0565	8.12492	8.75598
80	26.7407	15.3576	13.456	12.9395
90	69.4645	39.4887	35.3102	31.761
100	62.8221	34.7871	28.082	26.8019
125	161.262	84.0125	65.1187	59.563
150	307.765	157.241	118.174	106.892
175	767.909	386.142	282.991	243.117
200	1020.76	490.934	361.447	304.613
225	2030.74	969.104	693.433	567.913
250	2814.68	1028.16	868.556	697.807
275	4415.51	1533.29	1414.92	1114.89
300	6899.61	2157.87	1986.36	1544.39