РІВНЕНСЬКИЙ ДЕРЖАВНИЙ ГУМАНІТАРНИЙ УНІВЕРСИТЕТ

Факультет математики та інформатики

Кафедра інформатики та прикладної математики

«До захисту допущено»

Завідувач кафедри

_____ Батишкіна Ю.В.

«___»_____20___ р.

Дипломний проект (робота)

ступеня «Магістр»

з напряму підготовки (спеціальності) 122 – Комп'ютерні науки

на тему: Методи структурування програм на прикладі реалізації числових алгоритмів

Виконав: студент _2_ курсу, групи ___М-КН-21___

_____Шут Андрій Анатолійович_____ _____

Керівник: _к.ф.-м.н., доц. Мороз І. П._____ _____

Консультант: _____ _____

Рецензент: _____ _____

Рівне – 2021 року

ЗМІСТ

ВСТУП

Актуальність дослідження. Розвиток індустрії створення програмних засобів і все більш широке використання різних програм для задоволення інформаційних потреб людини істотно підвищують вимоги до надійності програмного виробу, тобто до зменшення кількості невиявлених помилок у програмі, що залишилися, і таких неврахованих ситуацій, при виникненні яких програма може видати невизначений результат чи припиняє своє нормальне функціонування.

Аналіз праць відомих фахівців у галузі програмування В. Афанасової, В. Бахтизіна, У. Вазирані, В. Воронінної, Л. Глухової, С. Дасгупти, Дж. Клейнберга, В. Колдаєва, І. Куличенко, О. Меркулової, А. Нікітіної, А. Овсянникова, Х. Пападимітріу, В. Пароджанова, Ю. Пікмана, В. Полякова, В. Потопахіна, І. Семушина, В. Скорубського, Р. Стівенса, Е. Тардос, В. Угарова, О. Федорова, Ю. Циганової, А. Шевченка засвідчив, що для вирішення виниклих при цьому проблем у практиці програмування вироблено низку прийомів і методів, які прийнято називати методами структурування програм. При прочитанні програми в її наступних фрагментах має чітко простежуватися логіка її роботи, тобто не повинно бути «стрибків» на фрагменти програми, які представлені десь в іншому місці програми. Значення слова «структурувати» – це розташовувати частини, елементи чогось у вигляді структури.

У наукових розробках М. Бабенко, М. Більфельд, Р. Бьорд, А. Затонський, М. Левін, Дж. Макконнелл, Н. Парфілова, А. Пилькін, О. Русакова, І. Семакін, Е. Тарунін, Б. Трусов, В. Шелест, А. Шкарапута, правильний вибір структури програмного продукту дає можливість зосередитися на одній осяжній частині програмного продукту кожному етапі розробки та забезпечити реалізацію його різних елементів одночасно;

Відомо, що від 50 до 100% часу програміста витрачається на виправлення та модифікацію програм. Тому сучасна індустрія програмування пропонує системні підходи до програмування, використання яких зменшує ймовірність

помилок у програмах, спрощує їхнє розуміння, полегшує модифікацію. Саме тому було обрано тему дослідження «Методи структурування програм на прикладі реалізації числових алгоритмів».

Об’єкт дослідження – методи структурування програм.

Предмет дослідження – особливості застосування методів структурування програм до реалізації числових алгоритмів.

Мета дослідження – теоретично обґрунтувати та розробити програму реалізації числових алгоритмів засобами методів структурування.

Завдання дослідження:

- висвітлити структурування програм як наукову проблему;

- обґрунтувати сутність та зміст методів структурування програм;

- запропонувати програмну реалізацію структурування числових алгоритмів.

Структура дослідження. Дипломна робота складається зі вступу, двох розділів, списку використаних джерел (32 найменування) та 7 додатків. Перший розділ присвячений теоретичним засадам структурування програм на прикладі реалізації числових алгоритмів. У другому розділі представлено програмну реалізацію структурування числових алгоритмів на прикладі алгоритмів розв’язання систем лінійних алгебраїчних рівнянь. Загальний обсяг дослідження – 60 сторінок, основний зміст дослідження представлений на 46 сторінках.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ЗАСАДИ СТРУКТУРУВАННЯ ПРОГРАМ НА ПРИКЛАДІ РЕАЛІЗАЦІЇ ЧИСЛОВИХ АЛГОРИТМІВ

1.1. Ефективність сучасних алгоритмів у розв’язуванні задач

Вивчення явищ із різних наукових галузей передусім повинне опиратися на понятійний апарат, а тому важливо визначити ключові поняття дослідження.

Алгоритм – це одне із базових понять у інформатиці, а тому можна зустріти численні його означення.

Відомий вчений у галузі програмування Д. Кнут вказує, що алгоритм – це кінцевий набір правил, який визначає послідовність операцій для розв’язання конкретної множини задач і володіє п'ятьма важливими рисами: скінченність, визначеність, ввід, вивід, ефективність [10].

За визначенням видатного математика XX століття Андрія Колмогорова алгоритм – це будь-яка система обчислень, що виконуються за строго певними правилами, яка після певної кількості кроків свідомо призводить до розв’язання поставленої задачі [30].

Відомий математик Андрій Марков писав у свій час, що алгоритм – це точний припис, що визначає обчислювальний процес, який йде від вихідних даних, що варіюються, до шуканого результату [15].

У «Новітньому філософському словнику» алгоритм трактується як точний припис про виконання в певному порядку деякої системи операцій, що ведуть до розв’язання всіх задач певного типу [17].

За словами Родда Стівенса алгоритм – це точний набір інструкцій, які описують порядок дій виконавця задля досягнення результату розв'язання задачі кінцевий час [27].

А. Шевченко розглядає алгоритм як коректно визначену обчислювальну процедуру, на вхід (input) якої подається деяка величина чи набір величин і результат виконання якої є вихідною (output) величиною чи набором значень [31]. Таким чином, алгоритм є послідовністю обчислювальних кроків, що перетворюють вхідні величини у вихідні.

М. Бабенко та М. Левін акцентують увагу на тому, що алгоритм також можна означити як інструмент, призначений на розв’язання коректно поставленої обчислювальної задачі (computational problem). У постановці задачі загалом задаються відношення між входом і виходом. У алгоритмі описується конкретна обчислювальна процедура, з допомогою якої вдається домогтися виконання зазначених відношень [1].

Автори С. Дасгупта, Х. Пападімітріу та У. Вазирані підкреслюють, що сучасний алгоритм має бути правильним, надійним та ефективним. Якщо він не справляється із поставленим завданням, від нього мало користі. Немає сенсу в ньому і тоді, коли він дає неправильний розв’язок [5].

Різні алгоритми, розроблені на розв’язання однієї й тієї ж задачі, часто дуже відрізняються за своєю ефективністю. Ці відмінності можуть бути набагато значнішими за ті, які спричинені застосуванням неоднакового апаратного та програмного забезпечення.

На думку Р. Стівенса, якщо алгоритм не є надійним, використовувати його у програмі недоцільно. По-справжньому хороший код має бути простим, інтуїтивно зрозумілим та вичерпним. Тільки в цьому випадку можна отримати правильний результат або внести необхідні зміни до структури програми. Якщо алгоритм складний і заплутаний, то виникнуть проблеми не лише з його реалізацією, а й усуненням можливих помилок. Під сумнів можна буде поставити сам результат [27].

Дж. Клейнберг , Е. Тардос звертають увагу на той факт, що більшість розробників багато часу приділяють ефективності алгоритмів тому, що правильний результат, легкість реалізації та можливість виправити помилки – все це зведеться до мінімуму, якщо алгоритм закінчить свою роботу через великий інтервал часу або вимагатиме обсяг пам'яті більший, ніж є в комп'ютері [8].

При визначенні ефективності алгоритму виникає питання про те, як зміниться продуктивність коду, якщо скоригувати розмірність задачі, наприклад, якщо подвоїти кількість змінних у алгоритмі, чи не подвоїться час

його роботи? А якщо звести початкову кількість змінних до четвертого степеня?

Щоб отримати уявлення про те, як складність завдання впливає продуктивність, програмісти користуються поняттям «асимптотична складність».

Томас Кормен вказує, що асимптотична складність (продуктивність) визначається функцією, яка вказує, наскільки погіршується робота алгоритму з ускладненням поставленого завдання. Таку функцію записують у круглих дужках, передуючи великою літерою O [12].

Наприклад, $O(N^2)$ означає, що зі збільшенням кількості вхідних даних час роботи алгоритму (використання пам'яті чи інший вимірюваний параметр) зростає квадратично. Якщо даних стане вдвічі більше, продуктивність алгоритму сповільниться приблизно чотири рази. При збільшенні кількості вхідних даних утричі вона зменшиться в дев'ять разів.

Вираз $O(N^2)$ читається як «порядок N у квадраті». Наприклад, алгоритм швидкого сортування має найгіршу продуктивність порядку N^2.

Р. Стівенсом представлено п'ять основних правил для розрахунку асимптотичної складності алгоритму.

1. Якщо для математичної функції/алгоритму необхідно виконати певні дії $f(N)$ разів, для цього йому знадобиться зробити $O(f(N))$ кроків.

2. Якщо алгоритм виконує одну операцію, що складається з $O(f(N))$ кроків, а потім другу операцію, що включає $O(g(N))$ кроків, то загальна продуктивність алгоритму для функцій f і g становитиме $O(f(N) + g(N))$.

3. Якщо алгоритму необхідно зробити $O(f(N) + g(N))$ кроків та область значень N функції $f(N)$ більша, ніж у $g(N)$, то асимптотичну складність можна спростити до виразу $O(f(N))$.

4. Якщо алгоритму всередині кожного кроку $O(f(N))$ однієї операції доводиться виконувати ще $O(g(N))$ кроків іншої операції, то загальна продуктивність алгоритму становитиме $O(f(N) \times g(N))$.

5. Постійними множниками (константами) можна знехтувати. Якщо є константа, то O(З × *f*(N)) або O(*f*(С × N)) можна записати як O(*f*(N)) [27].

Наведені правила здаються ддещо формальними через абстрактні функції *f*(N) і *g*(N), але насправді ними дуже легко користуватися практично.

Як правило, основна частина роботи алгоритму припадає на цикли. Іншого способу, що дозволяє зробити більше N кроків з фіксованою кількістю рядків коду, немає.

Важливо відзначити, що асимптотична складність дає уявлення про теоретичну поведінку алгоритму. Практичні результати можуть відрізнятись.

Алгоритм з продуктивністю O(1) виконується за той самий відрізок часу, незалежно від складності завдання. Як правило, йдеться про обмежене коло команд, оскільки за O(1) часу неможливо навіть переглянути всі вхідні дані.

Теорія дозволяє зрозуміти, як змінюється робочий цикл алгоритму, проте не менш важливі й практичні моменти. За словами Дж. Макконнелла, наприклад, під час аналізу алгоритму вважається, що ці кроки займають однакову кількість часу, навіть якщо це відповідає дійсності. Взяти хоча б створення та видалення нових об'єктів – на них потрібно набагато більше часу, ніж на переміщення цілих чисел з однієї частини масиву до іншої. У такому разі алгоритм з масивами працюватиме краще, ніж той, що використовує солідну кількість об'єктів і виглядає краще в асимптотичному позначенні [14].

Крім того, багато середовищ програмування надають доступ до функцій операційної системи, які є ефективнішими, ніж основні технології алгоритмізації. Наприклад, частина алгоритму, який виконує сортування, вимагає переміщення елементів масиву на одну позицію вниз, щоб можна було поставити новий елемент. Цей досить повільний процес призводить до того, робота алгоритму визначається як $O(N^2)$. Однак багато програм можуть використовувати особливу функцію, що переміщає блоки пам'яті (наприклад, RtlMoveMemory у програмах.NET-програмах і MoveMemory в програмах Windows C++. Таким чином, замість переміщення одного елемента при

проходженні через масив програма змушує функції пересувати послідовності величин, значно прискорюючи процес.

Навіть якщо алгоритм має оптимальну асимптотичну складність, на думку Р. Стівенса, при збільшенні продуктивності допустимо використовувати інструменти, які пропонує середовище програмування. Наприклад, багато бібліотек містять програми сортування і відмінно підходять для роботи з масивами даних. Так, Microsoft .NET Framework, що використовується в C# і Visual Basic, включає метод Array.Sort [27].

Подібні специфічні бібліотеки можуть бути загальнодоступними і корисними при виконанні певних завдань. Наприклад, ефективно використовувати бібліотеку мережевого аналізу, замість того, щоб самому розробляти деякі інструменти, і заощадити час, припустимо, на побудові дерев або сортуванні. Звичайно, можна виконувати ці завдання власноруч, але звернутися до готової бази даних набагато простіше.

Слушним є зауваження Р. Стівенса, згідно з яким, якщо інструменти програмування мають функції, що замінюють роботу будь-якого алгоритму, обов'язково потрібно скористатися ними. Так можна досягти кращої продуктивності і витратити менше ресурсів на налагодження [27].

А. Шевченко констатує, що для дуже великих завдань найефективніший алгоритм не завжди є найшвидшим. Припустимо, сортується довгий список чисел. І тут алгоритм швидкого сортування спрацює добре. Але якщо всього три числа, простий ряд операторів If, можливо, виявить себе набагато краще. Навряд чи матиме значення, впорається програма із завданням за 1 або за 2 мс. Якщо не планується виконувати сортування багаторазово, раціональніше вдатися до простого алгоритму, який легше піддається налагодженню, а не до складного, який заощадить лише 1 мс [31].

Якщо у процесі створення програми користуються бібліотеками, які аналогічні тим, що описані вище, можливо, не доведеться розробляти всі алгоритми самостійно, але розуміти принцип їхньої дії вкрай необхідно, щоб

отримати якомога більше користі з інструментів, які застосовуються для їх реалізації.

Як вважає В. Потопахін, щоб отримати максимальну користь від алгоритмів, необхідно розуміти принцип їхньої дії та мати уявлення про основні характеристики продуктивності [23].

Таким чином, ми розглянули асимптотичну складність і дізналися, як на її основі передбачити поведінку того чи іншого алгоритму при зміні розмірності задачі. Отже, ефективність алгоритму прийнято визначати за допомогою терміну «асимптотична складність» (продуктивність), що визначається функцією, що відображає наскільки погіршується робота алгоритму з ускладненням поставленого завдання.

На практиці алгоритми з поліноміальним часом роботи виявляються відносно швидкими, тому до них зручно звернутися для розв’язання помірно складних питань, на відміну від алгоритмів з експоненціальними чи факторіальними робочими циклами, у яких продуктивність змінюється дуже швидко та до яких краще апелювати у нетрудомістких процесах.

1.2. Структурування програм як наукова проблема

Одним з методів удосконалення програм є структурне програмування (СП). Л. Глухова та В. Бахтизін вважають, що його призначення полягає у організації проектування програм і процесу кодування, з метою запобігання більшості логічних помилок і виявлення уже допущених [4].

Розглянемо переваги структурного програмування.

На думку Р. Бьорда, структурне програмування дозволяє значно скоротити кількість варіантів побудови програми з однієї і тієї ж специфікації, що у результаті значно знижує складність програми [2].

А. Затонський та М. Більфельд відзначають, що у структурованих програмах логічно пов'язані оператори знаходяться візуально ближче, а слабко пов'язані знаходяться далі, що дозволяє обходитися без блок-схем та інших графічних форм зображення алгоритмів [6].

На думку А. Овсянникова та Ю. Пікмана, завдяки структуруванню суттєво спрощується процес тестування і налагодження програм [18].

В. Поляков та В. Скорубський зауважують, що структурованим програмам притаманна вища продуктивність роботи за рахунок того, що дія кожної керуючої структури добре відома і немає необхідності її обмірковувати [21].

Серед переваг структурного програмування слід вказати зрозумілість та чіткість програм, їх вищу ефективність за рахунок глобальної оптимізації програми.

Н. Парфілова, А. Пилькін, Б. Трусов констатують, що структурування програм зосереджується на одному з найбільш схильних до помилок аспектів програмування – логіки програми та включає складові.

1. проектування згори – вниз (низхідне проектування) – подібно до написання статті згори – вниз. Процес написання статті має ієрархічну структуру і починається з вершини ієрархії, тобто з короткого огляду. Розробку проекту зазвичай починають з дослідження цілей і визначення основних завдань, що ведуть до досягнення цих цілей. Якщо проект дуже великий, то необхідно провести його розбиття. Спочатку необхідно здійснити опис завдання засобами природної мові. Якщо записати завдання природною мовою неможливо, то навряд чи вдасться скласти програму. Отже, важливо формулювати завдання правильно на стадіях проектування, щоб не виправляти її пізніше на стадіях програмування і налагодження [20].

В. Потопахін відзначає, що метод проектування згори – вниз передбачає спочатку визначення завдання, а потім поступове уточнення структури шляхом внесення більш дрібних деталей. Проектування являє собою послідовність кроків такого уточнення. На кожному кроці необхідно виявити функції, які потрібно втілити, тобто це завдання розбивається на ряд, кожному з них буде відповідати один модуль. Саме такий традиційний і по суті ієрархічний підхід застосовується при створенні складних структур в інших галузях (в техніці, в серійному виробництві) [22].

2. Далі слід описати дані, вказуючи їх структуру та основні процеси обробки. Цей опис має включати ретельно відпрацьовані приклади, які переконливо демонструють функції системи і їх найбільш суттєві варіанти – такі приклади будуть корисні пізніше на стадії тестування.

За словами В. Паронджанова, при описі модуля повинні бути описані його тестові дані. Тестування програми неминуче, тому виявлення вимог до тестування, заздалегідь на стадіях проектування, є хорошою практикою. Логічна перевірка фрагментів програми повинна зменшити необхідність тестування кінцевої програми [19].

Основна перевага такого методу роботи полягає у тому, що він забезпечує створення документації.

Ще одним з важливих методів структурування програм, на думку В. Шелеста, є модульне програмування. Щоб досягти успіху в структурному програмуванні програма має бути представлена у вигляді модулів [32].

Б. Страуструп додає, що модульне програмування є процесом поділу програми на логічні частини, які називаються модулями, і послідовне програмування кожної частини [28].

Якщо велика єдина задача ділитися на підзадачі, то значно простіше прочитати і зрозуміти програму. Якщо здійснюється програмування всієї задачі згори – вниз, то вона природно розбивається на підзадачі для можливих модулів.

О. Меркулова, А. Нікитіна та О. Федоров підкреслюють, що при цьому переслідується дві мети:

- необхідність домогтися того, щоб програмний модуль був правильним і не залежав від контексту, в якому він буде використовуватися;

- слід прагнути до того, щоб з модулів можна було формувати великі програми без будь-яких попередніх визначень внутрішньої роботи модуля [7].

Б. Страуструп вважає, що оптимальний розмір модуля складає 60 рядків [28].

Слід дотримуватися до незалежності між модулями або програмами. Для досягнення цієї мети, щоб модуль не залежав від:

1) джерела вхідних даних;

2) місця призначення вихідних даних;

3) від передісторії [28].

При розбитті програми на функціональні блоки, незалежні модулі можна забезпечувати певну самостійність останніх. Хоча певна залежність все таки існує. Кожен модуль повинен мати своє призначення, яке відрізняється від призначення інших модулів. Це повинен бути замкнутий блок, вхід і вихід якого чітко визначені. Прагнення до незалежності добре тим, що менш ймовірно, що зміни в одній підпрограмі впливатимуть на решту програми [28].

Б. Страуструп дотримується думки, що вплив зміни в одному модулі на іншу частину програми називається хвильовим ефектом. Цей ефект можна зменшити, звівши до мінімуму зв'язок між модулями, тобто скоротити кількість шляхів уздовж яких зміни або помилки можуть проникнути в інші частини. Простий шлях зменшення хвильового ефекту – уникати використання глобальних змін і робити модуль невеликим [28].

Мінімізація взаємозв'язку між модулями – це модульне зчеплення, яке відбувається за рахунок посилення зв'язків між елементами одного модуля (модульна міцність). Таким чином, тісно пов'язані елементи треба старатися помістити в один модуль [28].

В. Потопахін зауважує, що використання модулів призводить до зменшення складності, чинники складності при цьому включають три складові:

1) функціональна складність – обумовлена тим, що один модуль виконує дуже багато функцій;

2) розподілена складність – це складність ідентифікації спільної функції розподіленої між декількома модулями, за рахунок чого втрачається можливість зменшення складності всієї програми при модульному програмуванні;

3) складність зв'язку визначається складністю взаємодії модулів, при використанні спільних даних [22].

На думку Р. Бьорда, структурне кодування – це метод написання добре структурованих програм, який дозволяє отримувати програми більш зручні для тестування, модифікації та використання. Програми довільних розмірів і складності можуть бути написані на основі обмеженої множини базисних структур [2].

Цей принцип покладено в основу проектування схем, де будь-яка логічна структура може бути створена з елементарних структур:

- структура послідовності – це формалізація того, що оператори програми виконуються в порядку їх появи в програмі, поки щось не змінить їх послідовність.

- структура вибору – це вибір однієї з двох дій виходячи з виконаної деякої умови.

- структура повторення – використовується для повторного виконання групи команд до тих пір, поки не виконається деяка умов [2]а.

А. Затонський та М. Більфельд відзначають, що отримання правильної програми шляхом заміни операторів програми на керуючі логічні структури називається вкладенням структур [6].

Н. Парфілова, А. Пилькін, Б. Трусов слушно зауважують, що в основу структурування покладено наступні положення:

1. програма повинна складатися з дрібних кроків, розмір кроку визначається кількістю рішень, які застосовуються програмістом на цьому кроці.

2. складне завдання розбивається на прості частини, які легко сприймаються, кожна з яких має тільки один вхід і один вихід.

3. логіка програми повинна опиратися на мінімальну кількість простих базових керуючих структур [20].

Л. Глухова та В. Бахтизін вважають, що фундаментом структурного програмування є теорема про структуризацію. Згідно з теоремою, яка б не була

складна задача, блок-схема програми може бути представлена з використанням дуже обмеженої кількості елементарних керуючих структур. Ці елементарні структури можуть поєднуватися між собою, утворюючи більш складні структури, при цьому вони можуть являти собою досить складні блок-схеми з одним входом і з одним виходом [4].

На думку А. Овсянникова та Ю. Пікмана, щоб не звертатись до довільних передач управління в програмі, там де це можливо, краще не використовувати оператори GOTO, RETURN. Характерною особливістю структурованої програми є не стільки відсутність цих операторів, а скільки наявність жорсткої структури її організації [18].

Б. Страуструп погоджується, що, якщо текст програми буде займати кілька десятків сторінок, то сприймання такої програми буде ускладненим, тому рекомендується вести структурування тексту програми (наприклад, програма складається з 80 сторінок вихідного тексту, необхідно цей вихідний текст замінити на текст, в якому відображаються найбільш важливі, в функціональному смислі, фрази у вигляді сегментів). Якщо розписати весь програмний комплекс великими сегментами, то опис всього комплексу може зайняти всього одну сторінку, а це наочно і зручно [28].

Графічне представлення програмних алгоритмів має широке поширення завдяки наочності. Граф-схема являє собою граф, що складається з вершин, з'єднаних спрямованими стрілками-дугами.

1. вершина – 

2. функціональний оператор -

3. умовний або об'єднувальний оператори – 

4. кінцевий оператор

5. дуги, що визначають послідовність виконання дій [19].

Програма називається простою, якщо її граф-схема задовольняє таким умовам:

1) один вхід і один вихід;

2) через кожну її вершину проходить хоча б один шлях від входу до виходу [19].

Складна програма зазвичай має більше, ніж один вхід або вихід.

Якщо комплекс програм великий і блок-схема програми велика, то необхідно виділити в ній підграфи, досить прості і зрозумілі за структурою.

Розглянемо як відбувається структурування програмних компонент. При ієрархічній структурі комплексу програм важливе значення мають обсяг, складність компонентів для кожного рівня ієрархії. За принципами побудови, мови опису, обсягом і іншими характеристикам в структурі комплексу програм О. Меркулова, А. Нікитіна та О. Федоров виділили наступні ієрархічні рівні:

- операторів і операндів програм, відповідних компонентів тексту програм на мові програмування;

- програмних модулів, які оформляються, як закінчені компоненти тексту програм;

- функціональні групи програм або пакета прикладних програм;

- комплексу програм, який оформлюється як завершений програмний засіб певного цільового призначення [7].

Багаторівневий ієрархічний підхід висхідного і спадного проектування дозволяє проектувати складні комплекси програм за принципом згори – вниз з позиції призначення і найкращого розв’язання основної мети, завдання всієї системи [7].

Іноді основне проектування згори – вниз супроводжує розробка компонент знизу – вгору. Розробка починається з модулів нижнього рівня, далі переходять до розробки модулів наступного рівня тощо.

Перевагою цього принципу, на думку О. Меркулової, А. Нікитіної та О. Федорова, є те, що при переході до розробки модулів більш високого рівня ієрархії, модулі нижніх рівнів можна вважати готовими і підключати їх до модулів верхнього рівня на стадії налагодження [7].

Розглянемо загальні правила структурної побудови програмних засобів. Якщо комплекс програм будується на основі модульно-ієрархічної структури,

що складається з програмних та інформаційних модулів, то на початкових етапах розробки комплексу програм формується його структура та загальні правила взаємодії компонент.

Ці правила, за словами Н. Парфілової, А. Пилькіна та Б. Трусова, полягають у наступному.

1. Правило зв'язку програмних модулів з управлінням.

Передача управління викликаному модулю завжди здійснюється через перший оператор або команду, а вихід з викликаного модуля завжди відбувається через його природне закінчення, тобто через останній оператор або команду. Після закінчення випробування викликаного модуля управління передається в викликаний модуль на оператор, який слідує безпосередньо за оператором виклику. Модулі нижніх рівнів або одного рівня ієрархії можуть викликатися для виконання тільки модулями вищих рівнів, тобто модулі нижніх рівнів не можуть викликати модулі вищих рівнів, а модулі одного рівня – викликати один одного. [20]

2. Правило зв'язку програмних модулів за інформацією.

Н. Парфілова, А. Пилькін та Б. Трусов вказують, що інформація зон глобальних змінних доступна для використання будь-якими модулями, що входять до комплексу програм або групи програм, відповідно до сфери дії зони глобальних змінних, тобто глобальні змінні можуть бути доступні не для всього комплексу програм, а лише для зазначеної в описі групи модулів [20].

Локальні змінні доступні лише в межах того модуля, в якому вони визначені. Для взаємодії модулів створюються зони обмінних змінних, інформація з яких доступна лише модулям безпосередньо пов'язаним з управлінням. За словами Н. Парфілової, А. Пилькіна та Б. Трусова Забороняється використання для обміну інформацією між модулями регістри і комірки пам'яті, які використовуються як регістри, тобто після закінчення роботи викликаного модуля вважається, що регістри не містять інформації. Інформація, що знаходиться в регістрах модуля, який викликається, при

зверненні повинна бути збережена на період виконання цього модуля і відновлена при поверненні до початкового модуля [20].

3. Типова структура програмного модуля.

Під структурою програмного модуля Н. Парфілова, А. Пилькін та Б. Трусов розуміють сукупність смислових частин, що утворюють модуль і використовуються для різних цілей при його розробці та застосуванні [20].

Типова структура модуля в загальному випадку включає:

1) заголовок;

2) опис змінних;

3) тіло модуля;

4) точки входу і виходу [20].

На думку А. Овсянникова та Ю. Пікмана, будь-яку програму можна синтезувати на основі елементарних базових конструкцій трьох типів: проста обчислювальна послідовність, альтернатива, ітерація [18].

Проста обчислювальна послідовність, полягає в перетворенні або переміщенні сукупності вихідних змінних. Елементарні конструкції слідують одна за одною, причому кінець попереднього оператора замикається з початком наступного.

Альтернатива полягає в перевірці деякої умови.

А. Шевченко відзначає, що ітерація являє собою структуру, в якій при кожному зверненні один з декількох операторів повторюється більше одного разу. Для структурування програм число ітерацій має бути заданим до входу в цикл, а не визначатися обчисленнями всередині циклу. В результаті чого виключається можлива невизначеність функціонування програми [31].

Т. Кормен вказує, що простота вихідних конструкцій структурного програмування запобігає появі складних інформаційних зв'язків і заплутаних передач управління [12].

Д. Кнут вважає, що структурованими вважаються програми, які не мають циклів з декількома виходами, не мають переходів всередину циклів або

умовних операторів і не мають виходів з внутрішньої частини циклів або умовних операторів [10].

Найбільш відомими методами, що дозволяють виконати структурування програм, є метод дублювання кодів програми, метод введення змінної стану і метод булевих ознак.

Отже, структурне програмування – одна з найпопулярніших методик, фундаментом якої є теорема про структурування. Якою б складною не була задача, блок-схема відповідної програми (алгоритму) завжди може бути представлена з використанням обмеженої кількості елементарних керуючих структур (послідовність, розгалуження, цикл).

Ідея структурування програми ґрунтується на тому, що слід здійснити перетворення кожної частини алгоритму на одну з трьох основних структур або їх комбінацію. Після достатньої кількості таких перетворень частина, що залишилася неструктурованою, або зникне, або стане непотрібною, в результаті вийде алгоритм, еквівалентний вихідному, у якому використовуються лише 3 керуючі структури.

Мета структурного програмування полягає у виборі структури програми шляхом розкладання вихідного завдання підзавдання (декомпозиція). Програми мають мати просту структуру. Розробка алгоритму спрощується кожному рівні крок за кроком.

1.3. Сутність та зміст методів структурування програм

Алгоритми – це основа програмування, що визначає, яким чином програмне забезпечення використовуватиме структури даних.

Як уже було сказано вище, ефективність алгоритму – це властивість алгоритму, яка пов’язана з обчислювальними ресурсами, які використовує алгоритм. М. Бабенко та М. Левін наголошують на тому, що алгоритм має бути проаналізованим з метою визначення необхідних алгоритму ресурсів . Ефективність алгоритму можна розглядати як аналог виробничої продуктивності повторюваних чи неперервних процесів [1].

А. Шевченко відзначає, що у загальному випадку довільна програма не може бути перетворена в структуровану програму, яка реалізує той же алгоритм, побудована із застосуванням тих же конструкцій і не використовує додаткових змінних. Таке перетворення можливе при використанні трьох відомих методів:

- дублювання кодів програми;
- введення змінної стану;
- метод булевих ознак [31].

Розглянемо застосування методу дублювання кодів програми на прикладі неструктурованої програми типу «решітка».

Алгоритм такої програми схематично представляє рис. 1.1.

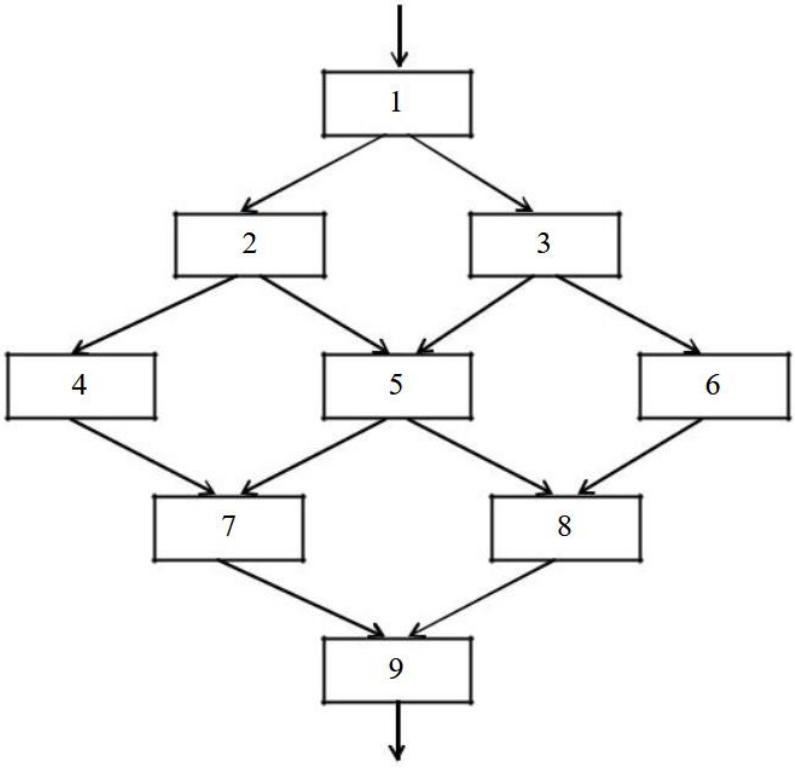

Рис. 1.2. Алгоритм неструктурованою програми типу «решітка»

Стрілки, що з'єднують блоки на цій схемі алгоритму, є операторами переходу. Стрілки, що з'єднують блоки на цій схемі алгоритму, є операторами переходу.

Як вважає В. Потопахін, програма, що реалізує цей алгоритм, не є структурованою, тому що не задовольняє умові «один вхід – один вихід» [23].

Розглянемо сутність методу дублювання кодів. У програмі дублюються ті модулі вихідного алгоритму або програми, в які можна увійти з декількох місць (крім останнього блоку).

Відповідно до цього у вихідній схемі необхідно дублювати модулі 5 (в нього можна увійти з модулів 2 і 3), 7 (в нього можна увійти з модулів 4 і 5) і 8 (в нього можна увійти з модулів 5 і 6). Це призводить вихідну схему до виду, який ілюструє рис. 1.2 [23].

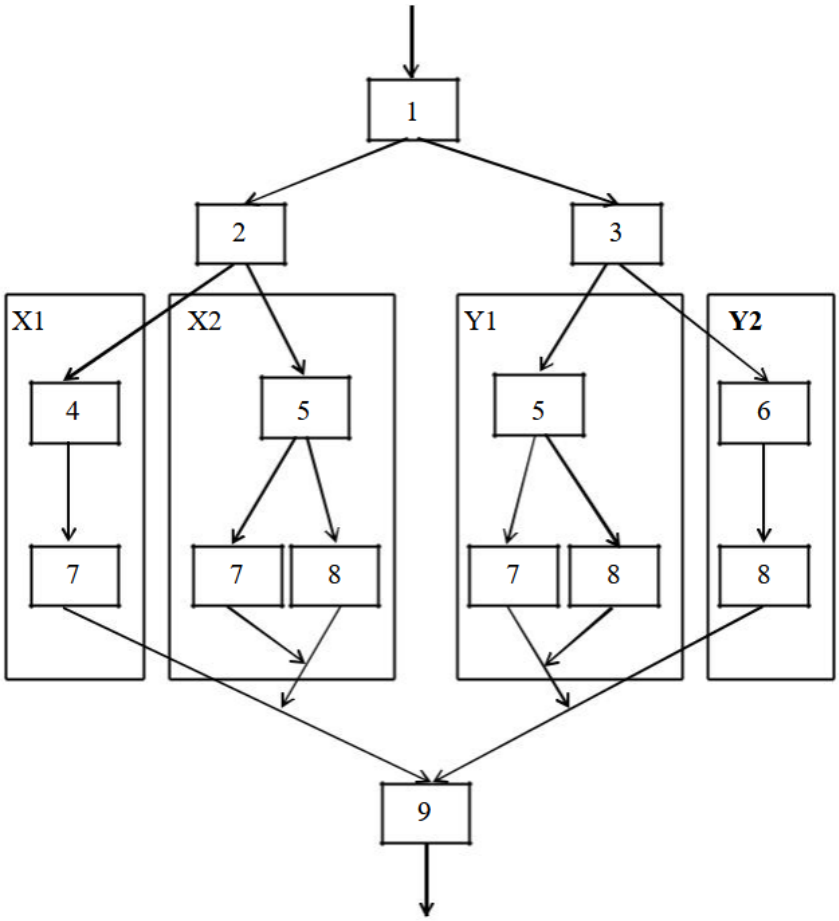

Рис. 1.2. Перетворена за методом дублювання кодів схема алгоритму

Отримана схема алгоритму є структурованою.

Щоб довести це, необхідно скористатися перетвореннями Бома-Джакопіні, тобто послідовно перетворити схему до одного функціонального блоку з одним входом і одним виходом. Для цього необхідно виконати кілька кроків (за В. Потопахіним) [23].

Крок 1 – перетворення. На вищенаведеною схемою модулі 4 і 7 є конструкцією слідування. Відповідно до перетворень Бома-Джакопіні вони можуть бути зведені до одного функціонального блоку X_1. Модулі 5, 7, 8 представляють собою конструкцію If-Then-Else з одним входом і одним виходом. Вони також можуть бути перетворені до одного функціонального блоку (X_2 і Y_1) [23].

Аналогічні міркування справедливі для конструкції слідування, що складається з модулів 6, 8 – вона зводиться до функціонального блоку Y_2. У результаті попередня схема набуває вигляду, який представляє рис. 1.3.

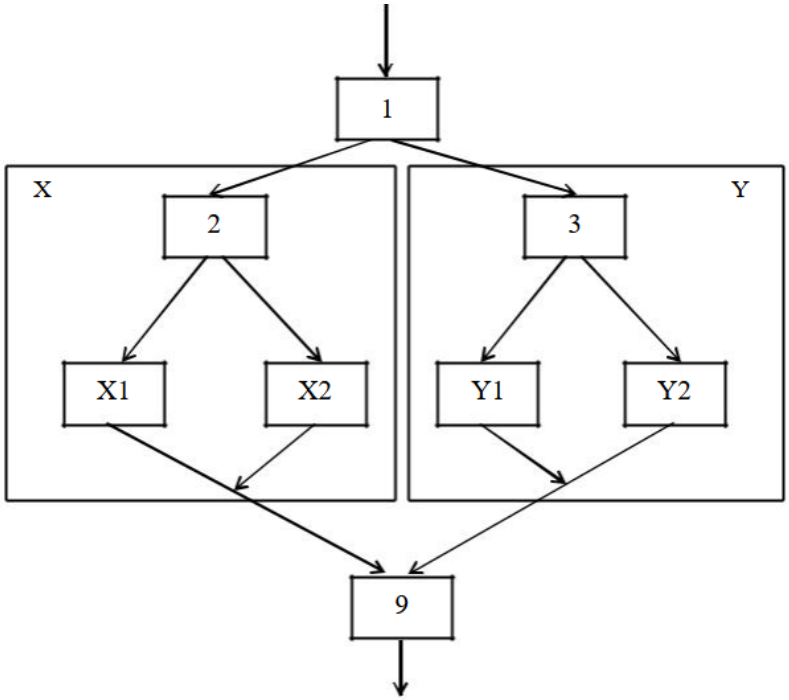

Рис. 1.3. Схема алгоритму після першого кроку перетворення Бома-Джакопіні

Крок 2 перетворення. В отриманій в результаті виконання кроку 1 схема група модулів 2, X1, X2 і група модулів 3, Y_1, Y_2 являють собою конструкції If-Then- Else з одним входом і одним виходом. Відповідно до перетворень Бома-Джакопіні їх можна представити у вигляді функціональних блоків (блоків X і Y відповідно). В результаті схема набуває вигляду, який ілюструє рис. 1.4 [23].

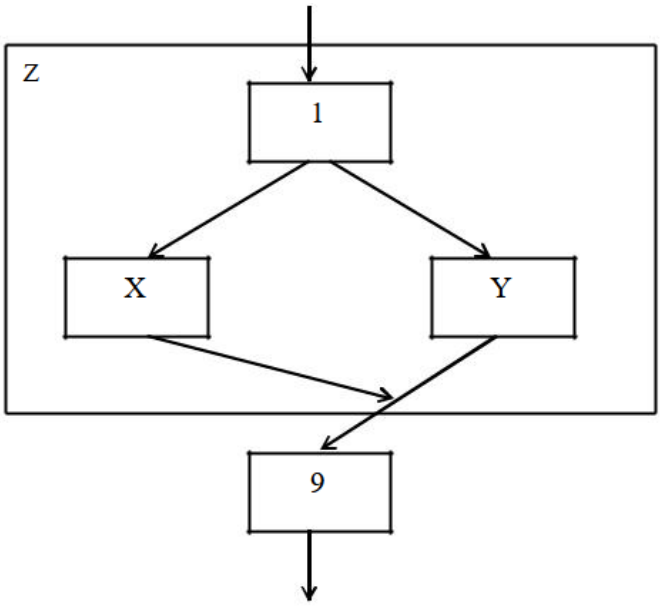

Рис. 1.4. Схема алгоритму після другого кроку перетворення Бома-Джакопіні

Крок 3 перетворення. В отриманій в результаті виконання кроку 2 схемі група модулів 1, X, Y являє собою конструкцію If-Then-Else с одним входом і одним виходом. Відповідно до перетворень Бома-Джакопіні її можна представити у вигляді функціонального блоку Z. В результаті схема набуває вигляду конструкції проходження (рис. 1.5) [23].

Крок 4 перетворення. Відповідно до перетвореннями Бома-Джакопіні конструкцію проходження можна представити у вигляді функціонального блоку R (див. рис. 1.5) [23].

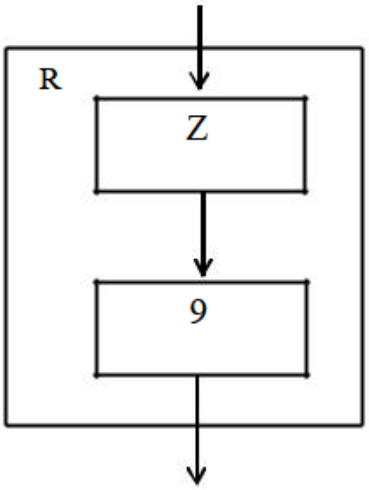

Рис. 1.5. Схема алгоритму після третього кроку перетворення Бома-Джакопіні

Таким чином, за допомогою чотирьох кроків перетворення доведено, що отримана в результаті застосування методу дублювання кодів схема алгоритму (див. рис. 1.2) є структурованою [23].

Перевагою методу дублювання кодів є те, що його зручно використовувати при низхідному проектуванні програм. Вихідну задачу укрупнено можна представити у вигляді одного функціонального блоку, а потім її поступово розукрупнювати через проміжні схеми алгоритму до результуючої структурованої схеми.

Л. Глухова та В. Бахтизін вважають, що є й недоліки методу дублювання кодів:

1) непридатність до програм з циклами;

2) додаткові витрати пам'яті для зберігання дубльованих модулів. Тому метод використовується, якщо дубльовані модулі містять незначну кількість операторів. Якщо модулі великі, то замість дублювання кодів необхідно використовувати виклик підпрограм з формальними параметрами [4].

Метод введення змінної стану був вперше запропонований Ашкрофт і Манною. Розглянемо застосування цього методу на прикладі неструктурованої програми, алгоритм якої схематично представляє рис. 1.6. Ця схема не є структурованою, оскільки з циклу, що складається з блоків 1 і 3, існує два виходи [23].

Таким чином, порушена умова «один вхід – один вихід», якій повинні задовольняти структуровані схеми.

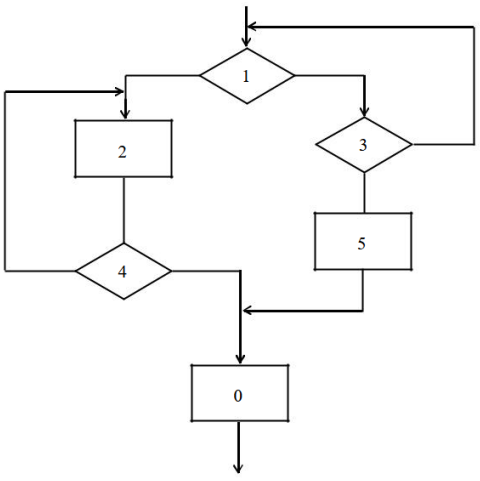

Рис. 1.6. Вихідна схема неструктурованого алгоритму

За словами Дж. Макконнелла процес перетворення програми в структуровану складається з наступної послідовності кроків.

1) Кожному блоку неструктурованою схеми приписується номер. Зазвичай першому блоку присвоюється 1, останньому – 0.

2) В програму вводиться додаткова змінна цілого типу (наприклад, *J*), яка називається змінною стану.

3) Функціональні блоки вихідної схеми замінюються блоками, які виконують крім основних функцій перетворення змінної *J*: змінної *J* присвоюється значення, рівне номеру блоку-приймача в вихідній схемі.

4) Аналогічно перетворюються логічні блоки. При цьому, якщо в логічному блоці умова істинно, то це відповідає одному значенню *J*, якщо хибна – іншому.

5) Початкова схема перебудовується до виду, запропонованого Ашкрофт-Манною (рис. 1.7) [14].

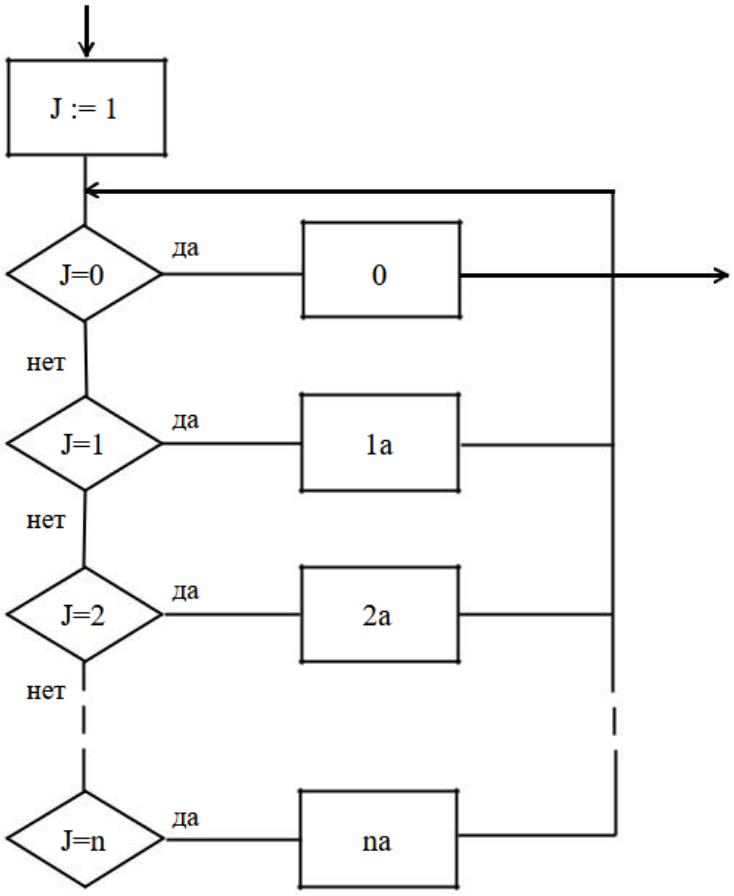

Рис. 1.7. Узагальнений вид схеми алгоритму, запропонований Ашкрофт-Манною

На цій схемі блоки 1а – nа є аналогами відповідних блоків вихідної схеми і, крім цього, присвоюють значення змінної *J* [14].

В результаті перетворень Ашкрофта-Манни вихідна неструктурована схема розглянутого прикладу (див. рис. 1.6) приймає структурований вигляд, який представляє рис. 1.8 [14].

При виконанні алгоритму, реалізованого за методом Ашкрофта-Манни, змінна стану *J* встановлюється в початкове значення, рівне номеру першого блоку неперетвореної схеми (як правило, це одиниця). Потім здійснюється послідовне опитування змінної *J*, починаючи з нуля і закінчуючи максимальним номером блоку вихідної схеми (в нашому прикладі він дорівнює п'яти).

Виконується той блок вихідної схеми, номер якого відповідає поточному значенню *J*. Крім цього в *J* заноситься значення, рівне номеру того блоку вихідної схеми, який повинен виконуватися за поточним блоком.

Коли значення *J* стане рівним нулю, виконується останній блок неперетвореної схеми (блок з номером нуль) і здійснюється вихід з алгоритму.

Отримана за методом Ашкрофта-Манни схема алгоритму є структурованою. За словами Л. Глухової та В. Бахтизіна для доказу цього достатньо послідовно перетворити цю схему до одного функціонального блоку [4].

Крок 1 перетворення.

Конструкції 1а, 3а і 4а чвляють собою конструкції If-Then-Else з одним входом і одним виходом, конструкції 2а, 5а є конструкціями слідування (див. рис. 1.8). Отже, вони можуть бути перетворені до відповідних функціональних блоків [4].

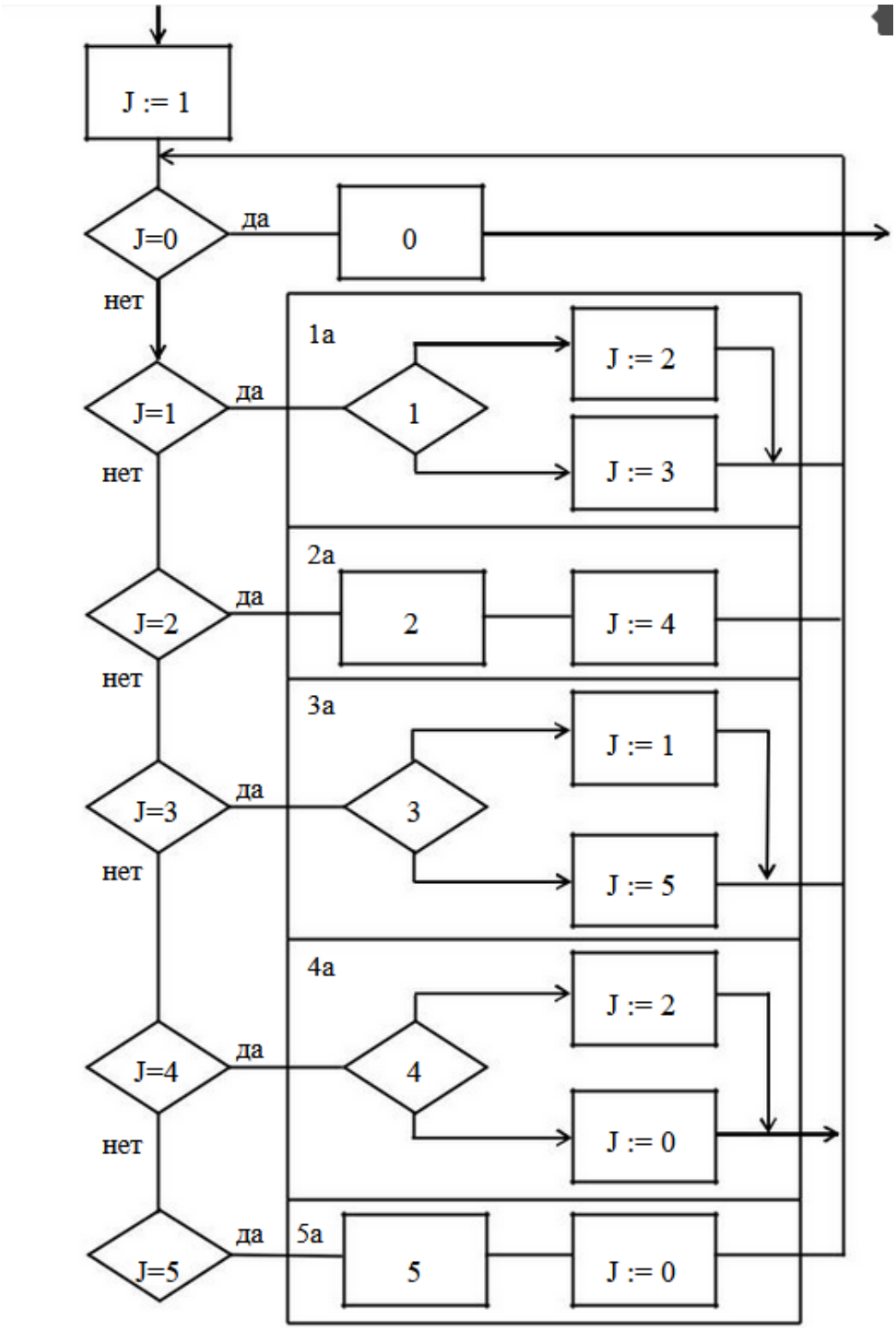

Рис. 1.8. Структурована форма вихідної схеми

Цей крок перетворень і всі наступні кроки пояснює рис. 1.9.

Наступні кроки перетворень необхідно проводити знизу-вгору схеми (див. рис. 1.9) [4].

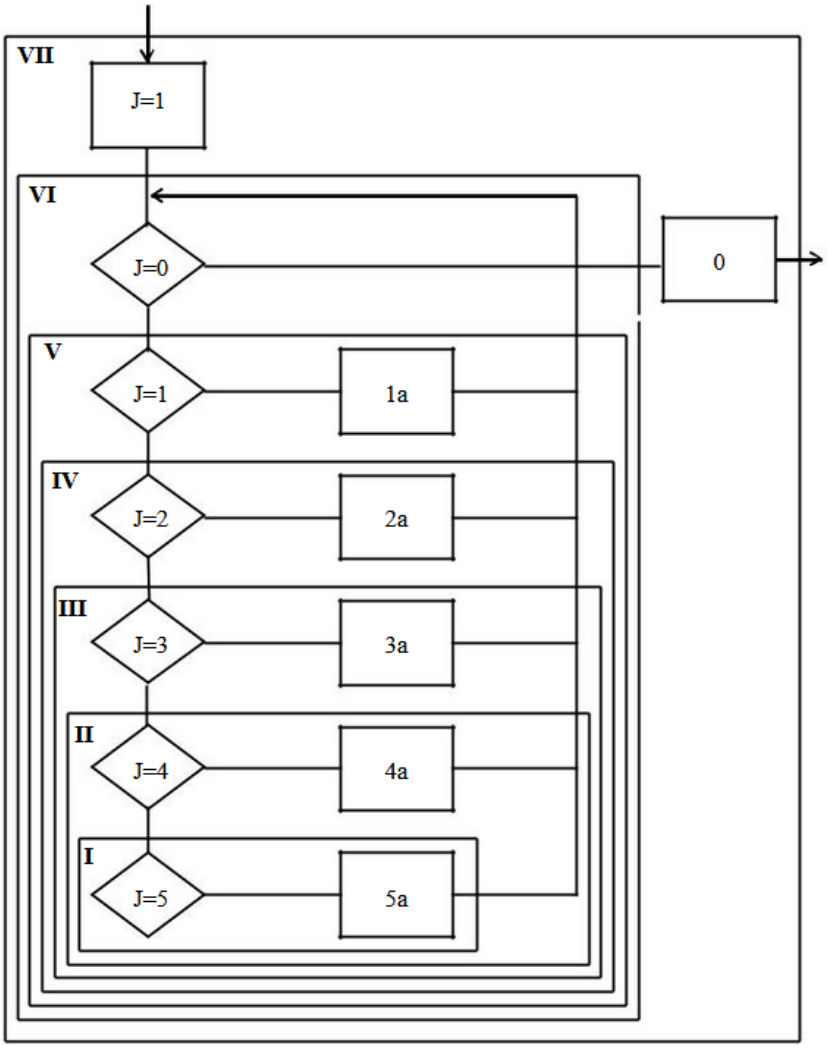

Рис. 1.9. Кроки перетворень Бома-Джакопіні для структурованої схеми

Крок 2 перетворення. Символ «Розв'язок» з перевіркою умови $J = 5$ і блок 5а являють собою конструкцію If-Then-else (з однією гілкою) з одним входом і одним виходом. Тому цей символ «Розв'язок» і блок 5а можуть бути замінені функціональним блоком I (див. Рис. 1.9).

Друга гілка даного символу «Розв'язок» може бути використана для підвищення надійності програми і на цьому рисунку не показана (ця вітка забезпечує можливість контролю непотрапляння значень J в діапазон 0 – n, де n – максимальний номер блоку у вихідній схемі).

По суті, без перевірки умови $J = 5$ в алгоритмі можна обійтися, перейшовши на виконання блоку 5а по гілці «ні» перевірки умови $J = 4$.

Крок 3 перетворення. Символ «Розв'язок» з перевіркою умови $J = 4$ і блоки I і 4а являють собою конструкцію If-Then-Else з одним входом і одним виходом. Тому вони замінюються функціональним блоком II (див. Рис. 1.9).

Крок 4 перетворення. Символ «Розв'язок» з перевіркою умови $J = 3$ і блоки II і 3а є конструкцією If-Then-Else з одним входом і одним виходом. Тому вони замінюються функціональним блоком III (див. рис. 1.9).

Крок 5 перетворення. Символ «Розв'язок» з перевіркою умови $J = 2$ і блоки III і 2а являють собою конструкцію If-Then-Else з одним входом і одним виходом. Тому вони замінюються функціональним блоком IV (див. рис. 1.9).

Крок 6 перетворення. Символ «Розв'язок» з перевіркою умови $J = 1$ і блоки IV та 1а являють собою конструкцію If-Then-Else з одним входом і одним виходом. Тому вони замінюються функціональним блоком V (див. рис. 1.9).

Крок 7 перетворення. Символ «Розв'язок» з перевіркою умови $J = 0$ і блок V являють собою конструкцію узагальненого циклу з одним входом і одним виходом. Тому вони замінюються функціональним блоком VI (див. рис. 1.9).

Крок 8 перетворення. Функціональний блок початкової установки $J = 1$, блок VI і блок 0 являють собою конструкцію слідування з одним входом і одним виходом. Тому вони замінюються функціональним блоком VII (див. рис. 1.9). [4]

Таким чином, за вісім кроків перетворень Бома-Джакопіні вихідна схема, побудована за методом Ашкрофта-Манни, перетворена в один функціональний блок з одним входом і одним виходом. Це підтверджує, що вона є структурованою.

Л. Глухова та В. Бахтизін визначають такі переваги методу введення змінної стану:

1) процес перетворення програми відрізняється наочністю і чіткістю;

2) будь-якому блоку вихідної схеми відповідає певний стан програми, що допомагає виконувати тестування і налагодження програми;

3) метод можна застосовувати до програм будь-якої структури (розгалужених і циклічних);

4) можливе автоматичне застосування цього методу [4].

Також вони називають і недоліки методу:

1) структурована форма схеми алгоритму сильно відрізняється від топології вихідної схеми, що ускладнює її розуміння;

2) додаткові витрати часу на аналіз і установку значень змінної стану;

3) громіздкість результуючої схеми [4].

Розглянемо тепер сутність методу булевої ознаки, вона полягає в наступному.

У програму, яка містить цикли, вводиться деяка ознака. Початкове значення ознаки задається до циклу. Цикл виконується, поки ознака зберігає своє початкове значення. Значення ознаки змінюється при наявності деяких умов всередині циклу [23].

Розглянемо застосування методу булевої ознаки на прикладі перетворення неструктурованої схеми, яку представляє рис. 1.10.

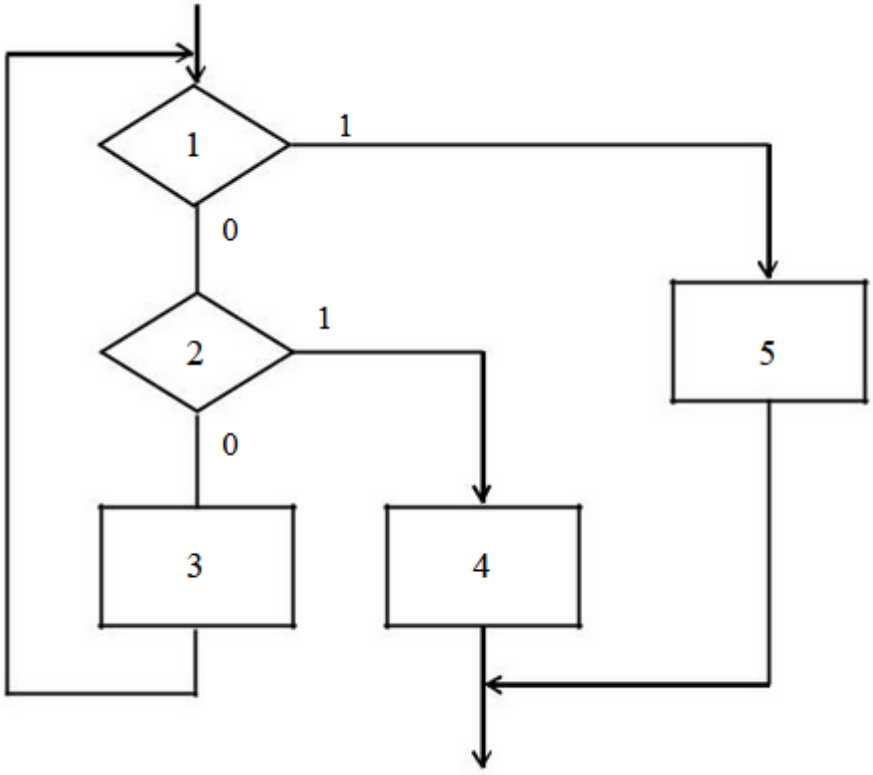

Рис. 1.10. Вихідна неструктурована схема

Схема алгоритму (див. рис. 1.10) не є структурованою, оскільки до неї входить цикл (блоки 1, 2, 3), що містить один вхід і два виходи. На цій схемі в блоках 1 і 2 записані деякі умови, що визначають виконання тієї чи іншої ділянки обчислень. Ці умови позначені як номери блоків 1 і 2. Значення 1 і 0 біля виходів символів «Розв'язок» відповідають логічним значенням «так» і «ні» [23].

Відповідно до розглянутих методів у вихідну схему вводиться ознака (наприклад, *J*). Схема набуває структурованого вигляду і легко реалізується конструкціями узагальненого циклу і прийняття двійкового розв'язку (рис. 1.11) [23].

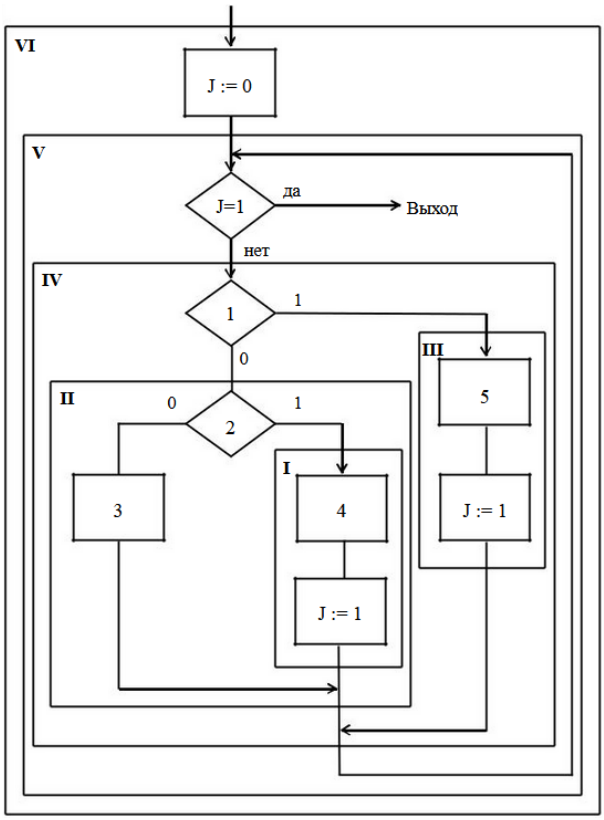

Рис. 1.11. Структурована форма вихідної схеми, перетворена за методом булевої ознаки

Рис. 1.11 схематично ілюструє кроки послідовності перетворень Бома-Джакопіні:

1) блок 4, функціональний блок *J*: = 1 → блок I;

2) блоки 2, 3, I → блок II;

3) блок 5, функціональний блок *J*: = 1 → блок III;

4) блоки 1, II, III → блок IV;

5) умовний блок (Розв'язок) *J* = 1, блок IV → блок V;

6) блоки *J*: = 0, V → блок VI.

Кожен з цих кроків перетворень на схемі (див. рис. 1.11) показаний пунктирним функціональним блоком.

А. Шевченко наводить переваги методу булевої ознаки:

1) компактність, економічність;

2) топологія вихідної схеми змінюється незначно [31].

Недолік методу булевої ознаки полягає у тому, що метод призначений для використання лише в циклах.

Іноді, на думку М. Бабенко та М. Левіна, можна обійтися без спеціального ознаки, використовуючи ті умови, які вже є у вихідній схемі [1].

Наприклад, розглянуту вихідну неструктуровану схему можна представити так, як ілюструє рис. 1.12.

На цьому рисунку умова «(1 і 2) = 0» означає одночасну рівність нулю умов, записаних в блоках 1 і 2 вихідної неструктурованої схеми (див. рис. 1.10). Таким чином, тіло циклу 3 буде виконане в тому випадку, якщо обидві умови 1 і 2 цієї статті не виконуються.

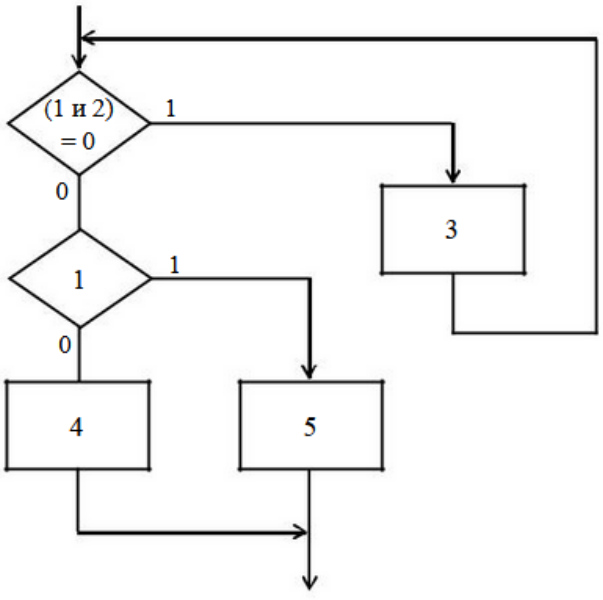

Рис. 1.12. Структурована форма вихідної схеми, перетворена без використання додаткового ознаки

Результуюча схема (див. рис. 1.12) містить узагальнений цикл з одним входом і одним виходом і конструкцію If-Then-Else, тобто є структурованою.

Слід мати на увазі, що перетворення схеми до структурованого вигляду без додаткової ознаки можливе і ефективне лише при невеликій кількості умов.

Отже, розглянуто такі методи структурування програм як метод дублювання кодів, метод перетворень Ашкрофта-Манни та метод булевої ознаки.

Виявлено, що методом дублювання кодів зручно застосовувати при низхідному проектуванні програм. Він полягає у тому, що в програмі дублюються ті модулі вихідного алгоритму або програми, в які можна увійти з декількох місць. Використання методу перетворень Ашкрофта-Манни (змінної стану) дозволяє трансформувати вихідну неструктуровану схему в структурований вигляд. Отже, сутність методу булевої ознаки полягає у тому, що до програми, що містить цикли, вводиться деяка ознака. Початкове значення ознаки задається до циклу. Цикл виконується доти, поки ознака не змінює своє початкове значення. За наявності певних умов всередині циклу значення ознаки змінюється.

Висновки до першого розділу

Розгляд структурування програм як наукової проблеми засвідчив, що воно дає можливість суттєво скоротити кількість варіантів побудови програми з однієї і тієї ж специфікації, що у результаті значно знижує її складність.

Отже, структурне програмування – одна з найпопулярніших методик, фундаментом якої є теорема про структурування. Якою б складною не була задача, блок-схема відповідної програми (алгоритму) завжди може бути представлена з використанням обмеженої кількості елементарних керуючих структур (послідовність, розгалуження, цикл).

Ідея структурування програми ґрунтується на тому, що слід здійснити перетворення кожної частини алгоритму на одну з трьох основних структур або їх комбінацію. Після достатньої кількості таких перетворень частина, що залишилася неструктурованою, або зникне, або стане непотрібною, в результаті вийде алгоритм, еквівалентний вихідному, у якому використовуються лише 3 керуючі структури.

Виявлено, що у структурованих програмах логічно пов'язані оператори знаходяться візуально ближче, а слабко пов'язані знаходяться далі, що дає змогу обходитися без блок-схем та інших графічних форм зображення алгоритмів.

Зрозумілість, чіткість програм, їх вища ефективність за рахунок глобальної оптимізації програми – це істотні переваги структурного програмування. Структурування програм сконцентроване на її логіці та включає проектування згори – вниз (низхідне проектування) та опис даних з вказівкою їх структури, основних процесів обробки.

Серед методів структурування програм важливе модульне програмування, згідно з яким, для досягнення успіху програма має бути представлена у вигляді модулів. По суті модульне програмування є процесом поділу програми на логічні частини, які називаються модулями, і послідовне програмування кожної частини. Використання модулів призводить до зменшення складності.

З’ясовано, що отримання правильної програми шляхом заміни операторів програми на керуючі логічні структури називається вкладенням структур, а базисом структурного програмування є теорема про структуризацію.

Метод дублювання кодів, який зручно використовувати при низхідному проектуванні програм, полягає у тому, що в програмі дублюються ті модулі вихідного алгоритму або програми, в які можна увійти з декількох місць. Метод перетворень Ашкрофта-Манни (змінної стану) дозволяє перетворити вихідну неструктуровану схему в структурований вигляд.

Отже, сутність методу булевої ознаки полягає у тому, що до програми, що містить цикли, вводиться деяка ознака. Початкове значення ознаки задається до циклу. Цикл виконується доти, поки ознака не змінює своє початкове значення. За наявності певних умов всередині циклу значення ознаки змінюється.

РОЗДІЛ 2. ПРОГРАМНА РЕАЛІЗАЦІЯ СТРУКТУРУВАННЯ ЧИСЛОВИХ АЛГОРИТМІВ

2.1. Технічне завдання та його програмна реалізація

Нам необхідно написати програму, яка виконує розв’язки системи лінійних рівнянь сімома різними методами. Результатом роботи програми має бути:

• Вектор x – розв’язок рівняння

• Час роботи кожного алгоритму

Після запуску програми, користувач повинен ввести матрицю коефіцієнтів. Після введення натиснути кнопку «Solve». Після цього з'являться значення вектора *x* і час роботи кожного алгоритму.

Мінімальне значення матриці — 2.

Можна також розв’язати усіма способами, якщо це можливо

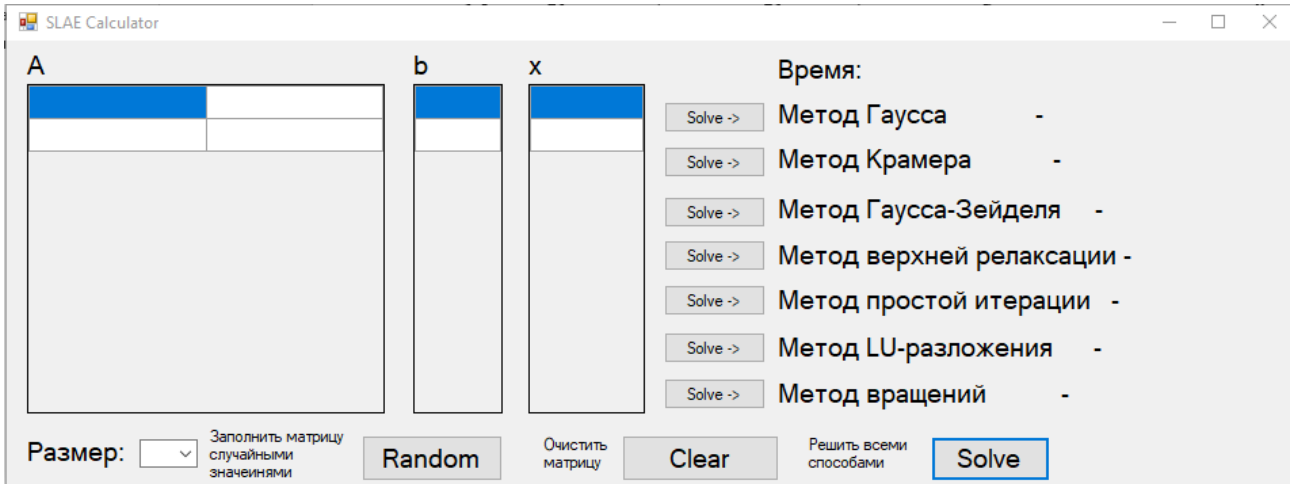

Рис 2.1. Програма до введення матриці

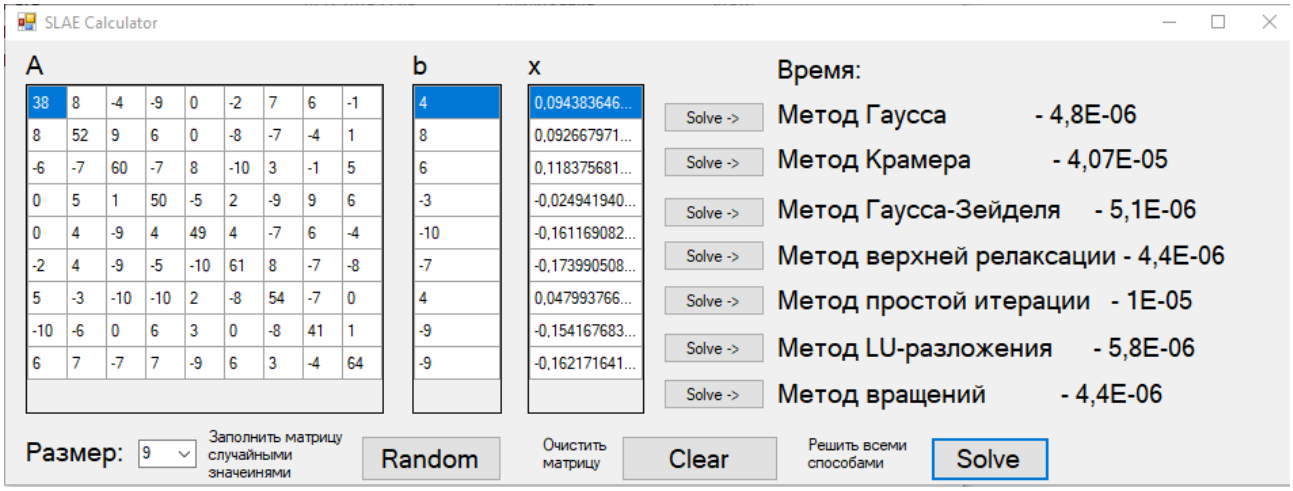

Рис. 2.2. Результат роботи програми

2.2. Опис реалізації числових алгоритмів

Метод Гауса – класичний метод розв'язання системи лінійних алгебраїчних рівнянь (СЛАР). Названий на честь німецького математика Карла Фрідріха Гауса. Цей метод передбачає послідовне виключення змінних, коли за допомогою елементарних перетворень система рівнянь зводиться до рівносильної системи трикутного вигляду, з якої послідовно, починаючи з останніх (за номером), знаходяться усі змінні системи.

Метод Гауса вимагає $O(n^3)$ арифметичних операцій.

Метод Крамера (правило Крамера) – спосіб розв'язання систем лінійних рівнянь алгебри з кількістю рівнянь рівним кількості невідомих з ненульовим головним визначником матриці коефіцієнтів системи (причому для таких рівнянь розв’язок існує і він єдиний).

Алгоритм для системи n лінійних рівнянь із n невідомими (над довільним полем)

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases}$$

та визначником матриці системи, відмінним від нуля, розв’язок записується у вигляді

$$x_i = \frac{1}{\Delta}\begin{vmatrix} a_{11} & \ldots & a_{1,i-1} & b_1 & a_{1,i+1} & \ldots & a_{1n} \\ a_{21} & \ldots & a_{2,i-1} & b_2 & a_{2,i+1} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ a_{n-1,1} & \ldots & a_{n-1,i-1} & b_{n-1} & a_{n-1,i+1} & \ldots & a_{n-1,n} \\ a_{n1} & \ldots & a_{n,i-1} & b_n & a_{n,i+1} & \ldots & a_{nn} \end{vmatrix}$$

Метод Крамера вимагає обчислення n + 1 визначників розмірності n x n. При використанні методу Гауса для обчислення визначників метод має складність за елементарними операціями додавання порядку $O(n_4)$, що складніше, ніж метод Гауса при прямому розв’язку системи.

Метод простих ітерацій є ітераційним, тому для запуску алгоритму, крім системи лінійних алгебраїчних рівнянь, необхідно задати необхідну точність розв’язку.

Алгоритм виглядає так:

1. Встановити вектор початкового наближення. Він дорівнює вектору вільних членів.

2. Розрахувати таке наближення за наступною формулою:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{i \neq j} a_{ij} x_j^{(k)}) \qquad (6.1)$$

7. Якщо норма вектора нев'язки вища за задану, перейти на п.2 інакше – розв’язок задано.

Метод Гауса-Зейделя – є класичним ітераційним методом розв'язання системи лінійних рівнянь.

Метод Зейделя є деякою модифікацією методу простої ітерації. Основна його ідея полягає в тому, що при обчисленні (k+1)-го наближення невідомої x_i враховуються вже обчислені раніше (k+1) – наближення невідомих x_1, x_2, ..., x_{i-1}.

У цьому методі, як і методі простої ітерації, необхідно привести систему до виду, щоб діагональні коефіцієнти були за модулем, і перевірити умови збіжності.

Якщо умови збіжності не виконуються, необхідно провести елементарні перетворення. Нехай дано система із трьох лінійних рівнянь. Виберемо довільно початкові наближення коренів: $x_1(0)$, $x_2(0)$, $x_3(0)$, намагаючись, щоб вони певною мірою відповідали шуканим невідомим. За нульове наближення можна прийняти стовпець вільних членів, тобто x(0) = b (тобто $x_1(0)=b_1$, $x_2(0)=b_2$, $x_3(0)=b_3$).

$$\begin{cases} x_1^{(1)} = \alpha_{12} x_2^{(0)} + \alpha_{13} x_3^{(0)} + \beta_1, \\ x_2^{(1)} = \alpha_{21} x_1^{(1)} + \alpha_{23} x_3^{(0)} + \beta_2, \\ x_3^{(1)} = \alpha_{31} x_1^{(1)} + \alpha_{32} x_2^{(1)} + \beta_3 \end{cases}$$

Слід звернути увагу до особливість методу Зейделя, що у тому, що отримане у першому рівнянні значення $x_1^{(l)}$ відразу ж у другому рівнянні, а значення $x_1^{(1)}$, $x_2^{(1)}$ – у третьому рівнянні тощо. Тобто, всі знайдені значення $x_1^{(1)}$ підставляються в рівняння для знаходження $x_{i+1}^{(1)}$.

Робочі формули для методу Зейделя для системи трьох рівнянь мають такий вигляд:

$$\begin{cases} x_1^{(k+1)} = \alpha_{12} x_2^{(k)} + \alpha_{13} x_3^{(k)} + \beta_1, \\ x_2^{(k+1)} = \alpha_{21} x_1^{(k+1)} + \alpha_{23} x_3^{(k)} + \beta_2, \\ x_3^{(k+1)} = \alpha_{31} x_1^{(k+1)} + \alpha_{32} x_2^{(k+1)} + \beta_3 \end{cases}$$

Запишемо у загальному вигляді для системи n-рівнянь робочі формули:

$$\begin{cases} x_1^{(k+1)} = \alpha_{12} x_2^{(k)} + \alpha_{13} x_3^{(k)} + \ldots + \alpha_{1n} x_n^{(k)} + \beta_1, \\ x_2^{(k+1)} = \alpha_{21} x_1^{(k+1)} + \alpha_{23} x_3^{(k)} + \ldots + \alpha_{2n} x_n^{(k)} + \beta_2, \\ \ldots \\ x_n^{(k+1)} = \alpha_{n1} x_1^{(k+1)} + \alpha_{n2} x_2^{(k+1)} + \ldots + \alpha_{n,n-1} x_{n-1}^{(k+1)} + \beta_n \end{cases}$$

Зауважимо, що теорема збіжності для методу простої ітерації справедлива й у методу Зейделя.

Задамо певну точність розв’язку e, після досягнення якої ітераційний процес завершується, тобто розв’язування триває до того часу, доки буде виконано умова для всіх рівнянь:

$\left| x_i^{(k+1)} - x_i^{(k)} \right| \le \varepsilon,$ де i=1,2,3,…,n.

Метод верхньої релаксації – ітераційний метод розв'язання систем лінійних рівнянь алгебри.

Система лінійних рівнянь

$$\begin{cases} a_{11} x_1 + \ldots + a_{1n} x_n = b_1 \\ a_{21} x_1 + \ldots + a_{2n} x_n = b_2 \\ \ldots \\ a_{n1} x_1 + \ldots + a_{nn} x_n = b_n \end{cases}$$

зводиться до вигляду

$$\begin{cases} b_{11}x_1 + b_{12}x_2 + \ldots + b_{1n}x_n + c_1 = 0 \\ \ldots \\ b_{n1}x_1 + b_{n2}x_2 + \ldots + b_{nn}x_n + c_n = 0 \end{cases},$$

Де $b_{ij} = -\frac{a_{ij}}{a_{ii}}, c_i = \frac{b_i}{a_{ii}}$

Знаходяться нев'язки:

$$\begin{cases} R_1^{(0)} = c_1 - x_1^{(0)} + \sum_{j=2}^{n} b_{1j}x_j^{(0)} \\ R_2^{(0)} = c_2 - x_2^{(0)} + \sum_{j=1, j\neq 2}^{n} b_{2j}x_j^{(0)} \\ \ldots \\ R_n^{(0)} = c_n - x_n^{(0)} + \sum_{j=1}^{n-1} b_{nj}x_j^{(0)} \end{cases}$$

Вибирається початкове наближення $X^{(0)} = 0$.

На кожному кроці необхідно перетворити в нуль максимальну нев'язку: $R_s^{(k)} = \delta x_s^{(k)} \Rightarrow R_s^{(k+1)} = 0, R_i^{(k+1)} = R_i^{(k)} + b_{is}\delta x_s^{(k)}$

Умова зупинки: $|R_j^{(k)}| < \varepsilon, \forall j = \overline{1,n}$.

Відповідь знаходиться за формулою: $x_i \approx x_i^{(0)} + \sum_j \delta x_i^{(j)}$.

Алгоритм:

1. Отримаємо формули для відшукання $x_{(s+1)}$ за попереднім наближенням $x^{(s)}$ у явному вигляді.

$$(D + \omega L)(x^{(s+1)} - x^{(s)}) + \omega A x^{(s)} = \omega b,$$

$$Dx^{(s+1)} + \omega L x^{(s+1)} - Dx^{(s)} - \omega L x^{(s)} + \omega A x^{(s)} = \omega b,$$

$$Dx^{(s+1)} = -\omega L x^{(s+1)} + Dx^{(s)} - \omega(A - L)x^{(s)} + \omega b.$$

З урахуванням того, що A-L=R+D отримуємо

$$Dx^{(s+1)} = -\omega L x^{(s+1)} + (1 - \omega)Dx^{(s)} - \omega R x^{(s)} + \omega b.$$

Далі неважко записати явні формули для відшукання компонентів нового вектора $x^{(s+1)}$:

$$a_{ii}x_i^{(s+1)} = -\omega\sum_{j=1}^{i-1}a_{ij}x_j^{(s+1)} + (1-\omega)a_{ii}x_i^{(s)} - \omega\sum_{j=i+1}^{n}a_{ij}x_j^{(s)} + \omega b_i$$

Як слідує з формули, при підрахунку i-ї компоненти нового наближення всі компоненти, індекс яких менший за i, беруться з нового наближення $x^{(s+1)}$, а всі компоненти, індекс яких більший або дорівнює i – зі старого наближення $x^{(s)}$. Таким чином, після того, як i-а компонента нового наближення обчислена, i-о компонента старого наближення ніде не використовуватиметься. Навпаки, для підрахунку наступних компонентів вектора $x^{(s+1)}$ компоненти з індексом, меншим або рівним i, будуть використовуватися «в новій версії». У силу цієї обставини для реалізації методу достатньо зберігати лише одне (поточне) наближення $x^{(s)}$, а при розрахунку наступного наближення $x^{(s+1)}$ використовувати формулу для всіх компонентів по порядку та поступово оновлювати вектор $x^{(s)}$.

Метод LU – розкладання зводить розв’язування системи лінійних рівнянь до розв’язання двох простих систем. Якщо відомо LU-розкладання матриці, вихідна система може бути записана так:

$$LUx = b, \tag{5.1}$$

Скориставшись формулою множення матриць та прирівнявши отримане до вихідної матриці A, отримаємо формули для обчислення невідомих коефіцієнтів:

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ki}^2} \tag{5.3}$$

$$l_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} l_{ki}l_{kj}}{l_{ii}} \tag{5.4}$$

Алгоритм розв’язку СЛАР буде виглядати так:

1. Виконати LU-розкладання матриці A, скориставшись формулами (5.3) та (5.4).

2. Розв’язати систему прямою підстановкою

3. Розв’язати систему зворотною підстановкою (аналогічно зворотному ходу методу Гауса)

Метод Жордана-Гауса розв'язання системи лінійних алгебраїчних рівнянь поділено на 2 етапи: прямий та зворотний хід. Прямий хід, що приводить вихідну систему до виду (2.3), повністю ідентичний прямому ходу методу Гаусса.

Зворотній хід

У зворотному циклі за діагональними елементами матриці A:

1. Для кожного рядка обчислити коефіцієнт за такою формулою:

$$\alpha_i = \frac{a_{ji}}{a_{ii}}, \tag{3.1}$$

де i – номер рядка ведучого елемента; j – номер поточного рядка

2. Виконати елементарне перетворення рядків:

$$(j) = (j) - \alpha_i (i) \tag{3.2}$$

В результаті система буде приведена до такого вигляду:

$$\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a'_{22} & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & \cdots & a'_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{pmatrix} = \begin{pmatrix} b''_1 \\ b''_2 \\ \cdots \\ b''_n \end{pmatrix} \tag{3.3}$$

3. Обчислити компоненти вектора розв’язків за такою формулою:

$$x_i = \frac{b''_i}{a'_{ii}} \tag{3.4}$$

Ми працювали над програмною реалізацією структурування числових алгоритмів - методів розв’язання систем лінійних алгебраїчних рівнянь (методи Гауса, Крамера, простих ітерацій, Гауса-Зейделя, верхньої релаксації, LU – розкладання, Жордана-Гауса).

Метод Гауса – це класичний метод розв'язання системи лінійних алгебраїчних рівнянь, який вимагає $O(n^3)$ арифметичних операцій.

Отже, метод Крамера є способом розв'язання систем лінійних алгебраїчних рівнянь з кількістю рівнянь рівною кількості невідомих з ненульовим головним визначником матриці коефіцієнтів системи. Цей метод у програмній реалізації має складність за елементарними операціями додавання порядку $O(n_4)$,

Метод простих ітерацій є ітераційним, тому для запуску алгоритму, крім СЛАЛ, необхідно задати необхідну точність розв’язку. Метод Гауса-Зейделя – є класичним ітераційним методом розв'язання системи лінійних рівнянь. Метод верхньої релаксації – ітераційний метод розв'язання систем лінійних рівнянь алгебри. Метод LU – розкладання зводить розв’язування системи лінійних рівнянь до розв’язання двох простих систем. Метод Жордана-Гауса розв'язання системи лінійних алгебраїчних рівнянь поділено на 2 етапи: прямий та зворотний хід.

Висновки до другого розділу

У ході виконання дослідження ми застосували прийоми розв'язання систем лінійних алгебраїчних рівнянь. Величезна кількість чисельних методів ставить актуальним завданням не стільки створення нових, скільки дослідження та класифікацію старих, виявлення кращих методів. Аналіз впливу помилок показав, що найкращими методами немає принципової різниці з погляду стійкості до помилок округлення. Створення потужних комп'ютерів істотно послабило значення різниці між методами (у таких характеристиках, як об'єм пам'яті, кількість арифметичних операцій). В таких умовах найкращими стають ті методи, які дуже відрізняються від кращих за швидкістю і зручності реалізації на комп'ютерах, дозволяють розв'язувати широкий клас задач, як добре, і погано обумовлених і оцінювати точності обчислювального розв’язку.

У результаті ми застосували різні чисельні методи на практиці.

У ході виконання дослідження було проведено порівняльний аналіз чисельних методів Гауса, Гауса-Зейделя, метод верхньої релаксації, LU – розкладання, Жордана-Гауса. Зокрема, метод Зейделя у деяких випадках призводить до більш швидкої збіжності, ніж метод простих ітерацій.

ВИСНОВКИ

Отже, структурування програм дає можливість суттєво скоротити кількість варіантів побудови програми з однієї і тієї ж специфікації, що у результаті значно знижує її складність.

Отже, структурне програмування є однією з найпопулярніших методик, основою якої є теорема про структурування. Якою б складною не була задача, блок-схема відповідної програми (алгоритму) завжди може бути представлена з використанням обмеженої кількості елементарних керуючих структур (послідовність, розгалуження, цикл).

Ідея структурування програми ґрунтується на тому, що слід здійснити перетворення кожної частини алгоритму на одну з трьох основних структур або їх комбінацію. Після достатньої кількості таких перетворень частина, що залишилася неструктурованою, або зникне, або стане непотрібною, в результаті вийде алгоритм, еквівалентний вихідному, у якому використовуються лише 3 керуючі структури.

Виявлено, що у структурованих програмах логічно пов'язані оператори знаходяться візуально ближче, а слабко пов'язані знаходяться далі, що дає змогу обходитися без блок-схе м та інших графічних форм зображення алгоритмів.

Зрозумілість, чіткість програм, їх вища ефективність за рахунок глобальної оптимізації програми – це істотні переваги структурного програмування. Структурування програм сконцентроване на її логіці та включає проектування згори – вниз (низхідне проектування) та опис даних з вказівкою їх структури, основних процесів обробки.

Серед методів структурування програм важливе модульне програмування, згідно з яким, для досягнення успіху програма має бути представлена у вигляді модулів. По суті модульне програмування є процесом поділу програми на логічні частини, які називаються модулями, і послідовне програмування кожної частини. Використання модулів призводить до зменшення складності.

З’ясовано, що отримання правильної програми шляхом заміни операторів програми на керуючі логічні структури називається вкладенням структур, а базисом структурного програмування є теорема про структуризацію.

Метод дублювання кодів, який зручно використовувати при низхідному проектуванні програм, полягає у тому, що в програмі дублюються ті модулі вихідного алгоритму або програми, в які можна увійти з декількох місць. Метод перетворень Ашкрофта-Манни (змінної стану) дозволяє перетворити вихідну неструктуровану схему в структурований вигляд.

Отже, сутність методу булевої ознаки полягає у тому, що до програми, що містить цикли, вводиться деяка ознака. Початкове значення ознаки задається до циклу. Цикл виконується доти, поки ознака не змінює своє початкове значення. За наявності певних умов всередині циклу значення ознаки змінюється.

Ми працювали над програмною реалізацією структурування числових алгоритмів - методів розв’язання систем лінійних алгебраїчних рівнянь (методи Гауса, Крамера, простих ітерацій, Гауса-Зейделя, верхньої релаксації, LU – розкладання, Жордана-Гауса).

Метод Гауса – це класичний метод розв'язання системи лінійних алгебраїчних рівнянь, який вимагає $O(n^3)$ арифметичних операцій.

Метод Крамера є способом розв'язання систем лінійних алгебраїчних рівнянь з кількістю рівнянь рівною кількості невідомих з ненульовим головним визначником матриці коефіцієнтів системи. Цей метод у програмній реалізації має складність за елементарними операціями додавання порядку $O(n_4)$,

Метод простих ітерацій є ітераційним, тому для запуску алгоритму, крім СЛАЛ, необхідно задати необхідну точність розв’язку. Метод Гауса-Зейделя – є класичним ітераційним методом розв'язання системи лінійних рівнянь. Метод верхньої релаксації – ітераційний метод розв'язання систем лінійних рівнянь алгебри. Метод LU – розкладання зводить розв’язування системи лінійних рівнянь до розв’язання двох простих систем. Метод Жордана-Гауса розв'язання системи лінійних алгебраїчних рівнянь поділено на 2 етапи: прямий та зворотний хід.

У ході виконання дослідження ми застосували прийоми розв'язання систем лінійних алгебраїчних рівнянь. Величезна кількість чисельних методів ставить актуальним завданням не стільки створення нових, скільки дослідження та класифікацію старих, виявлення кращих методів. Аналіз впливу помилок показав, що найкращими методами немає принципової різниці з погляду стійкості до помилок округлення. Створення потужних комп'ютерів істотно послабило значення різниці між методами (у таких характеристиках, як об'єм пам'яті, кількість арифметичних операцій). В таких умовах найкращими стають ті методи, які дуже відрізняються від кращих за швидкістю і зручності реалізації на комп'ютерах, дозволяють розв'язувати широкий клас задач, як добре, і погано обумовлених і оцінювати точності обчислювального розв’язку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бабенко М.А., Левин М.В. Введение в теорию алгоритмов и структур данных. Электронное издание. Москва : МЦНМО, 2016. 144 с.

2. Бёрд Р. Жемчужины проектирования алгоритмов : функциональный подход / Пер. с англ. В.Н. Брагилевского и А.М. Пеленицына. Москва : ДМК Пресс, 2013. 330 с.

3. Вычислительная линейная алгебра в проектах на C# : учебное пособие / И.В. Семушин, Ю.В. Цыганова, В.В. Воронина, В.В. Угаров, А.И. Афанасова, И.Н. Куличенко. Ульяновск : УлГТУ, 2014. 429 с.

4. Глухова Л.А., Бахтизин В.В. Основы алгоритмизации и структурного проектирования программ : Учеб. пособие по курсам «Основы алгоритмизации и программирования» и «Технология разработки программного обеспечения» для студ. спец. 400101 «Программное обеспечение информационных технологий» дневной формы обучения. Минск : БГУИР, 2003. 72 с.

5. Дасгупта С., Пападимитриу Х., Вазирани У. Алгоритмы. / Пер. с англ. под ред. А. Шеня. Москва : МЦНМО, 2014. 320 с.

6. Затонский А., Бильфельд Н. Программирование и основы алгоритмизации. Москва : «Озон», 2019. 176 с.

7. Избранные вопросы теории алгоритмов : учебно-методическое пособие / сост. : О.О. Меркулова, А.Б. Никитина, О.А. Фёдоров. Южно-Сахалинск : СахГУ, 2018. 112 с.

8. Клейнберг Дж., Тардос Е. Алгоритмы: разработка и применение. Классика Computers Science / Пер. с англ. Е. Матвеева. Санкт-Петербург : Питер, 2016. 800 с.

9. Клири С. Конкурентность в C#. Асинхронное, параллельное и многопоточное программирование. 2-е межд. изд. Санкт-Петербург : Питер, 2020. 272 с.

10. Кнут Д.Э. Искусство программирования. Том 2. Получисленные алгоритмы = The Art of Computer Programming. Volume 2. Seminumerical

Algorithms / под ред. Л.Ф. Козаченко (гл. 3, разд. 4.6.4 и 4.7), В.Т. Тертышного (гл. 4) и И.В. Красикова (разд. 4.6). 3. Москва : Вильямс, 2001. Т. 2. 832 с.

11. Колдаев В.Д. Численные методы и программирование: учебное пособие. / Под ред. проф. Л.Г. Гагариной. Москва : ИД «ФОРУМ» : ИНФРА-М, 2009. 336 с.

12. Кормен Томас Х. Алгоритмы : построение и анализ, 3-е изд. : Пер. с англ. Москва : ООО «И.Д. Вильяме» 2013. 1328 с.

13. Котов О.М. Язык C# : краткое описание и введение в технологии программирования : учебное пособие. Екатеринбург : Изд-во Урал, ун-та, 2014. 208 с.

14. Макконнелл Дж. Основы современных алгоритмов. 2-е дополненное издание Москва : Техносфера, 2004. 368 с.

15. Марков А.А. Избранные труды / Сост. и общ. ред. Н.М. Нагорного. Москва : Изд-во МЦНМО, 2002. Т. 2. 648 с.

16. Мейерс С. Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов. Москва : ДМК Пресс. 298 с.

17. Новейший философский словарь : 3-е изд., исправл. / Сост. и гл. науч. ред. А.А. Грицанов. Минск : Книжный Дом. 2003. 1280 с.

18. Овсянников А.В., Пикман Ю.А. Алгоритмы и структуры данных : учебно-методический комплекс для специальности 1-310307 «Прикладная информатика (по направлениям)». Ч.1 ; БГУ, Факультет социокультурных коммуникаций, Кафндра информационных технологий. Минск : БГУ, 2015. 124 с.

19. Паронджанов В.Д. Учись писать, читать и понимать алгоритмы. Алгоритмы для правильного мышления. Основы алгоритмизации. Москва : ДМК Пресс, 2016. 520 с.

20. Парфилова Н.И, Пылькин А.Н., Трусов Б.Г. Программирование : структурирование программ и данных : учебник. 2-е издание. Москва Академия, 2016. 238 с.

21. Поляков В.И., Скорубский В.И. Основы теории алгоритмов. Санкт-Петербург : СПб НИУ ИТМО, 2012. 51 с.

22. Потопахин В. Современное программирование с нуля! Москва : ДМК Пресс, 2016. 350 с.

23. Потопахин В.В. Искусство алгоритмизации. Москва : ДМК Пресс, 2011. 320 с.

24. Потопахин В.В. Язык С. Освой на примерах. Санкт-Петербург : БХВ-Петербург, 2006. 320 с.

25. Прата С. Язык программирования С. Лекции и упражнения, 6-е изд. : Пер. с англ. Москва : ООО «И.Д. Вильямс», 2015. 928 с.

26. Семакин И.Г., Русакова О.Л., Тарунин Е.Л., Шкарапута А.П. Программирование, численные методы и математическое моделирование : учебное пособие. Москва : КНОРУС, 2020. 298 с.

27. Стивенс Р. Алгоритмы. Теория и практическое применение / [перевод с английского В.В. Кириленко, Р.В. Волошко]. 2-е издание. Москва : Эксмо, 2021. 544 с.

28. Страуструп Б. Программирование : принципы и практика с использованием C++, 2-е изд.: Пер. с англ. Москва : ООО «И.Д. Вильямс», 2016. 1328 с.

29. Тюкачев Н.А., Хлебостроев В.Г. C#. Основы программирования : Учебное пособие. 3-е изд., стер. Санкт-Петербург : Издательство «Лань», 2018. 272 с.

30. Успенский В.А. Колмогоров. Новая философская энциклопедия : в 4 т. / пред. науч.-ред. совета В.С. Стёпин. 2-е изд., испр. и доп. Москва : Мысль, 2010. 2816 с.

31. Шевченко А.В. Программирование и основы алгоритмизации : учеб. пособие. Санкт-Петербург : Изд-во СПбГЭТУ «ЛЭТИ», 2011. 144 с.

32. Шелест В.Д. Программирование. Санкт-Петербург : БХВ-Петербург, 2002. 592 с.

ДОДАТКИ

Додаток А

Код до програми розв’язання СЛАР методом Гаусса

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SLAE
{
    class GaussMethod : SLAEMethod
    {
        public override double[] Solve(double[,] matrix, double[] b)
        {
            int length = matrix.GetLength(0);
            double[] x = new double[length];

            //прямой хід рішення
            for (int k = 1; k < length; k++)
            {
                for (int i = k; i < length; i++)
                {
                    double coeff = matrix[i, k - 1] / matrix[k - 1, k - 1];

                    for (int j = 0; j < length; j++)
                    {
                        matrix[i, j] -= coeff * matrix[k - 1, j];
                    }

                    b[i] -= coeff * b[k - 1];
                }
            }

            //обратний хід рішення
            for (int i = length - 1; i >= 0; i--)
            {
                x[i] = b[i] / matrix[i, i];

                for (int j = length - 1; j > i; j--)
                {
                    x[i] -= matrix[i, j] * x[j] / matrix[i, i];
                }
            }

            return x;
        }
    }
}
```

Додаток Б

Код до програми розв'язання СЛАР методом простої ітерації, методом Гаусса-Зейделя

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SLAE
{
    class GaussSeidelMethod : SLAEMethod
    {
        private readonly double eps;

        public GaussSeidelMethod(double eps = 0.00001)
        {
            this.eps = eps;
        }

        private bool Converge(ref double[] xk, ref double[] xkp)
        {
            int length = xk.GetLength(0);

            double norm = 0.0;
            for (int i = 0; i < length; i++)
            {
                norm += (xk[i] - xkp[i]) * (xk[i] - xkp[i]);
            }

            return Math.Sqrt(norm) < eps;
        }

        public override double[] Solve(double[,] matrix, double[] b)
        {
            int length = matrix.GetLength(0);
            double[] p = new double[length];
            double[] x = new double[length];

            do
            {
                Array.Copy(x, p, length);

                for (int i = 0; i < length; i++)
                {
                    double var = 0.0;
                    for (int j = 0; j < i; j++)
                    {
                        var += matrix[i, j] * x[j];
                    }
                    for (int j = i + 1; j < length; j++)
                    {
                        var += matrix[i, j] * p[j];
                    }
                    x[i] = (b[i] - var) / matrix[i, i];
                }

            } while (!Converge(ref x, ref p));

            return x;
    }}
```

Додаток В

Код до програми розв'язання СЛАР методом Крамера

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SLAE
{
    class CramerMethod : SLAEMethod
    {
        public override double[] Solve(double[,] matrix, double[] b)
        {
            int length = matrix.GetLength(0);
            double detMatrix = Determinant(matrix);
            double[] x = new double[length];

            for (int i = 0; i < length; i++)
            {
                double[,] newMatrix = new double[length, length];
                Array.Copy(matrix, newMatrix, length * length);
                for (int j = 0; j < length; j++)
                {
                    newMatrix[j, i] = b[j];
                }

                x[i] = Determinant(newMatrix) / detMatrix;
            }

            return x;
        }

        private double Determinant(double[,] matrix)
        {
            int length = matrix.GetLength(0);

            for (int k = 1; k < length; k++)
            {
                for (int i = k; i < length; i++)
                {
                    double coeff = matrix[i, k - 1] / matrix[k - 1, k - 1];

                    for (int j = 0; j < length; j++)
                    {
                        matrix[i, j] -= coeff * matrix[k - 1, j];
                    }
                }
            }



            double det = 1.0;
            for (int i = 0; i < length; i++)
            {
                det *= matrix[i, i];
```

```
            }

            return det;
        }

        private void swap_row(ref double[,] matrix, int i, int j)
        {
            int length = matrix.GetLength(0);

            for (int k = 0; k < length; k++)
            {
                double temp = matrix[i, k];
                matrix[i, k] = matrix[j, k];
                matrix[j, k] = temp;
            }
        }
    }
}
```

Додаток Г

Код до програми розв'язання СЛАР методом LU розкладу

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SLAE
{
    class LUDecomposition : SLAEMethod
    {
        private void LUDecompose(ref double[,] matrix, out double[,] L, out double[,] U)
        {
            int length = matrix.GetLength(0);
            L = new double[length, length];
            U = new double[length, length];

            for (int j = 0; j < length; j++)
            {
                U[0, j] = matrix[0, j];
                L[j, 0] = matrix[j, 0] / U[0, 0];
            }

            for (int i = 1; i < length; i++)
            {
                for (int j = i; j < length; j++)
                {
                    double sum = 0.0;
                    for (int k = 0; k < i; k++)
                    {
                        sum += L[i, k] * U[k, j];
                    }

                    U[i, j] = matrix[i, j] - sum;
                }

                for (int j = i; j < length; j++)
                {
                    double sum = 0.0;
                    for (int k = 0; k < i; k++)
                    {
                        sum += L[j, k] * U[k, i];
```

```
                }

                L[j, i] = (matrix[j, i] - sum) / U[i, i];
            }
        }
    }

    public override double[] Solve(double[,] matrix, double[] b)
    {
        int length = matrix.GetLength(0);

        LUDecompose(ref matrix, out double[,] L, out double[,] U);

        double[] y = new double[length];
        for (int i = 0; i < length; i++)
        {
            double sum = 0.0;
            for (int j = 0; j < i; j++)
            {
                sum += L[i, j] * y[j];
            }

            y[i] = b[i] - sum;
        }

        double[] x = new double[length];
        for (int i = length - 1; i >= 0; i--)
        {
            double sum = 0.0;
            for (int j = i + 1; j < length; j++)
            {
                sum += U[i, j] * x[j];
            }

            x[i] = (y[i] - sum) / U[i, i];
        }

        return x;
    }
  }
}
```

Додаток Д

Код програми генератор матриць

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SLAE
{
    class GeneratorMatrix
    {
        Random random = new Random();

        public double[,] GenerateMatrix(int size)
        {
            double[,] matrix = new double[size, size];

            for (int i = 0; i < size; i++)
            {
                double sum = 0.0;
                for (int j = 0; j < size; j++)
                {
                    matrix[i, j] = random.Next(-10, 10);
                    sum += Math.Abs(matrix[i, j]);
                }

                matrix[i, i] = sum + random.Next(0, 10);
            }

            return matrix;
        }

        public double[] GenerateVector(int size)
        {
            double[] b = new double[size];

            for (int i = 0; i < size; i++)
            {
                b[i] = random.Next(-10, 10);
            }

            return b;
        }
    }
}
```

Додаток Е

Код до програми розв'язання СЛАР методом простої ітерації

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SLAE
{
    class SimpleIterationMethod : SLAEMethod
    {
        private readonly double eps;

        public SimpleIterationMethod(double eps = 0.00001)
        {
            this.eps = eps;
        }
        private bool Converge(ref double[] xk, ref double[] xkp)
        {
            int length = xk.GetLength(0);

            double norm = 0.0;
            for (int i = 0; i < length; i++)
            {
                norm += (xk[i] - xkp[i]) * (xk[i] - xkp[i]);
            }

            return Math.Sqrt(norm) < eps;
        }
        public override double[] Solve(double[,] matrix, double[] b)
        {
            int length = matrix.GetLength(0);
            double[] p = new double[length];
            double[] x = new double[length];

            double[,] alpha = new double[length, length];
            double[] beta = new double[length];

            for (int i = 0; i < length; i++)
            {
                beta[i] = b[i] / matrix[i, i];
                for (int j = 0; j < length; j++)
                {
                    alpha[i, j] = (i != j) ? -matrix[i, j] / matrix[i, i] : 0;
                }
            }

            Array.Copy(beta, x, length);

            do
            {
                Array.Copy(x, p, length);

                for (int i = 0; i < length; i++)
                {
                    double[] c = new double[length];
                    for (int j = 0; j < length; j++)
                    {
                        c[i] += alpha[i, j] * p[j];
                    }

                    x[i] = beta[i] + c[i];
```

```
                }

            } while (!Converge(ref x, ref p));


            return x;
        }
    }
}
```

Додаток Є

Код до програми розв’язання СЛАР методом верхньої ітерації

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SLAE
{
    class UpperRelaxationMethod : SLAEMethod
    {
        public override double[] Solve(double[,] matrix, double[] b)
        {
            int length = matrix.GetLength(0);
            double[] x = new double[length];

            //double coeff;

            GeneratorMatrix generator = new GeneratorMatrix();


            return x;
        }
    }
}
```

Додаток Ж

Код до програми розв’язання СЛАР методом Якобі

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SLAE
{
    abstract class SLAEMethod
    {
        public abstract double[] Solve(double[,] matrix, double[] b);

        public bool IsDiagonalityDominance(ref double[,] matrix)
        {
            int length = matrix.GetLength(0);

            for (int i = 0; i < length; i++)
            {
                double sumNonDiagonalElems = 0.0;
                for (int j = 0; j < length; j++)
                {
                    if (i == j)
                        continue;
                    else
                        sumNonDiagonalElems += Math.Abs(matrix[i, j]);
                }

                if (Math.Abs(matrix[i, i]) <= sumNonDiagonalElems)
                    return false;
            }

            return true;
        }
    }
}
```