МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: «РОЗРОБКА СИСТЕМИ АВТОМАТИЗАЦІЇ ІНЖЕНЕРНИХ РОЗРАХУНКІВ ЗАСОБАМИ JULIA»

Виконав: студент 2 курсу, групи 8.1213-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
Ю.Ю. Кабанов
(ініціали та прізвище)

Керівник професор кафедри програмної інженерії, доцент, к.ф.-м.н. Кудін О.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент завідувач кафедри фундаментальної та прикладної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Кабанову Юрію Юрійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка системи автоматизації інженерних розрахунків засобами Julia

керівник роботи Кудін Олексій Володимирович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 01 » березня 2024 року № 385-с

2. Строк подання студентом роботи 25.11.2024 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Технічне завдання до роботи.
3. Комплект нормативних документів.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Дослідження проблеми автоматизації інженерних розрахунків.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 07.03.2024 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	08.04.2024	
2.	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником.	15.04.2024	
3.	Аналіз існуючих методів рішення.	29.04 2024	
4.	Дослідження базових бібліотек мови програмування Julia та засобів чисельних обчислень.	13.05.2024	
5.	Дослідження спеціалізованих бібліотек мови програмування Julia та засобів чисельних обчислень.	27.05.2024	
6.	Узгодження подальших дій з науковим керівником.	03.06.2024	
7.	Розробка першого розділу.	28.06.2024	
8.	Розробка другого розділу.	30.08.2024	
9.	Розробка третього розділу.	31.10.2024	
10.	Оформлення та нормоконтроль	20.11.2024	
	кваліфікаційної роботи магістра.		
11.	Захист кваліфікаційної роботи.	12.12.2024	

Студент _____ Ю.Ю. Кабанов
(підпис) (ініціали та прізвище)

Керівник роботи _____ О.В. Кудін
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційн робота магістра «Розробка системи автоматизації інженерних розрахунків засобами Julia»: 79 с., 73 рис., 3 табл., 43 джерела.

ІНЖЕНЕРНІ РОЗРАХУНКИ, МАШИННЕ НАВЧАННЯ, НАУКОВІ ДОСЛІДЖЕННЯ, ЧИСЕЛЬНІ МЕТОДИ, DATA SCIENCE, JULIA.

Об’єктом дослідження є процес інженерних розрахунків чисельними методами.

Метою роботи є створення інженерної програмної системи, яка дозволить автоматизувати інженерні розрахунки з використанням переваг мови програмування Julia.

Методи дослідження: аналіз вимог до системи автоматизації інженерних розрахунків, розробка архітектури системи, визначення компонент, необхідних для реалізації функціональності системи, використання сучасних методів програмування та інструментів розробки.

Сучасні методи обробки наукових даних в контексті Data Science у таких галузях, як інженерія, машинне навчання, штучний інтелект вимагають від науковця вміння працювати з великими масивами інформації та програмними засобами.

Одним з таких програмних засобів є мова програмування Julia. Це відкрита вільна високопродуктивна динамічна мова високого рівня, створена спеціально для технічних (математичних) обчислень. Його синтаксис близький до синтаксису інших середовищ технічних обчислень, таких як Matlab та Octave. Він має у своєму складі складний компілятор, що забезпечує розподілене паралельне виконання інструкцій, обчислювальну точність та велику бібліотеку математичних функцій. Базова бібліотека в основному написана тією ж мовою Julia, але також включає розвинені, кращі у своєму класі відкриті бібліотеки.

SUMMARY

Master's qualifying paper "Development of a Computer Aided Engineering System Using Julia": 79 pages, 73 figures, 3 tables, 43 references.

ENGINEERING CALCULATIONS, MACHINE LEARNING, SCIENTIFIC RESEARCH, NUMERICAL METHODS, DATA SCIENCE, JULIA.

The object of the study is is the process of engineering calculations using numerical methods, which often require high computing power and efficiency for processing large amounts of data.

The aim of the study is to create a software system that will allow automating engineering calculations using the advantages of the Julia programming language.

The methods of research are analysis of requirements for the automation of engineering calculations, development of the system architecture, determination of components necessary to implement the functionality of the system, use of modern programming methods to ensure the efficiency of the system, testing and evaluation.

Modern methods of processing scientific data in the context of Data Science in such fields as engineering, machine learning, and artificial intelligence require the scientist to be able to work with large amounts of information and software tools.

One such software tool is the Julia programming language. It is an open, free, high-performance, high-level dynamic language created specifically for technical (mathematical) calculations. Its syntax is close to that of other technical computing environments, such as Matlab and Octave. It includes a sophisticated compiler that provides distributed parallel execution of instructions, computational precision, and a large library of mathematical functions. The library is mostly written in the same Julia language, but also includes advanced, best-in-class open-source libraries.

ЗМІСТ

Завдання на кваліфікаційну роботу……2
Реферат ……..4
Summary ……5
Вступ……..…8
1 Дослідження проблеми використання мов програмування в інженерно-математичних та наукових розрахунках 10
1.1 Огляд проблеми використання мов програмування в інженерно-математичних та наукових розрахунках 10
1.2 Огляд літературних джерел з використання мов програмування в інженерно-математичних та наукових розрахунках 11
1.3 Компілятори та програмні інструменти для інженерно-математичних і наукових розрахунків 14
1.4 Бібліотеки для наукових обчислень 23
1.5 Висновки з розділу 1 26
2 Дослідження засобів чисельного моделювання для інженерно-математичних та наукових розрахунків мовою Julia 27
2.1 Мова програмування Julia для рішення інженерно-математичних та наукових задач 27
2.1.1 Спеціалізовані бібліотеки мови julia 28
2.2 Висновки з розділу 2 37
3 Розробка системи автоматизації інженерних розрахунків засобами мови Julia.39
3.1 Еталонні форми кінцевих елементів у бібліотеці Ferrite.jl 39
3.2 Типи об’єктів CellValues та FacetValues 41
3.3 Ступені свободи (Degrees of Freedom, DOF) 42
3.4 Шаблон розрідженості та розріджені матриці 43

3.5 Збірка локальних на глобальних матриць 44
3.6 Граничні та початкові умови (Boundary Conditions) 46
3.7 Обмеження (Constraints) 51
3.8 Рішення лінійних та нелінійних систем у Ferrite.jl 52
3.9 Розрахункова сітка у Ferrite.jl 53
3.10 Експорт результатів розрахунків 55
3.11 Реалізація функціоналу програмних кодів для виконання инженерних розрахунків методом кінцевих елементів засобами Julia 57
3.12 Висновки з розділу 3 73
Висновки 75
Перелік посилань 76

ВСТУП

Більшість прикладних задач, що постають перед інженером чи науковцем так чи інакше зводяться або до обробки експериментальних даних, або до математичного моделювання фізичного процесу. Для вирішення обох цих задач є широкий спектр різних апаратних засобів, починаючи від персональних комп'ютерів та закінчуючи обчислювальними кластерами та суперкомп'ютерами. Що ж до програмного забезпечення, можна виділити цілий клас мов програмування, призначених спеціально для чисельних розрахунків. У порівнянні з мовами загального призначення вони забезпечують простий (інтуїтивно зрозумілий) синтаксис програм, а також велику бібліотеку спеціалізованих функцій. Всі вони інтерпретуються, що прискорює реалізацію та налагодження алгоритмів, але негативно позначається на швидкості роботи програм.

Окреме місце серед мов програмування для инженерних та наукових розрахунків займає Julia. Це відносно молода мова, перша версія якої була опублікована у лютому 2012 року. Концепція мови полягає в тому, що в рамках однієї мови можна вирішувати як задачі прототипування, так і глибокої оптимізації коду (включаючи перегляд асемблерного коду, підтримку паралельних і розподілених обчислень, а також інші можливості). Додатки, повністю написані мовою Julia, практично не поступаються у продуктивності додаткам, написаним статично компилируемых мовами на кшталт C чи C++.

Мета і завдання дослідження полягають у створенні інженерної програмної системи на мові програмування Julia, яка дозволить автоматизувати інженерні розрахунки з використанням головних переваг мови, таких як висока продуктивність, гнучкість та простота використання.

Об’єктом дослідження є інженерні розрахункі чисельними методами, які часто вимагають високої обчислювальної потужності та ефективності для обробки великих обсягів даних.

Предметом дослідження є інженерний програмний застосунок, який реалізує усі переваги мови програмування Julia на на основі чисельних методів.

Для вирішення поставленої задачі використовуються наступні методи дослідження:

– аналіз вимог до системи автоматизації інженерних розрахунків;
– розробка архітектури системи, визначення компонент та модулів, необхідних для реалізації функціональності системи;
– використання сучасних методів програмування та інструментів розробки для забезпечення ефективності та надійності системи;
– експериментування з використанням різних математичних алгоритмів;
– тестування та оцінка ефективності роботи отриманого програмного коду у порівняні з існуючими реалізаціями.

Практичне значення одержаних результатів дослідження полягає в тому, що була розроблена інженерно-програмна система математичного моделювання, яка дозволяє оптимально виконувати обчислення методами кінцевих елементів використовуючи всі переваги мови програмування Julia. З цього можна зробити висновок, що представлений у роботі підхід до створення інженерних програмних застосунків можна використовувати в якості ефективного та дешевого рішення для створення більш складних комп’ютерних систем інженерного аналізу.

Результати дослідження були представлені на XV Всеукраїнській, XXII регіональній науково-практичній конференції молодих дослідників Запорізького національного університету «Актуальні проблеми математики та інформатики».

1 ДОСЛІДЖЕННЯ ПРОБЛЕМИ ВИКОРИСТАННЯ МОВ ПРОГРАМУВАННЯ В ІНЖЕНЕРНО-МАТЕМАТИЧНИХ ТА НАУКОВИХ РОЗРАХУНКАХ

1.1 Огляд проблеми використання мов програмування в інженерно-математичних та наукових розрахунках

Наукові та інженерні розрахунки часто вимагають виконання складних математичних операцій, обробки великих обсягів даних та моделювання складних систем. Мови програмування відіграють ключову роль цих завданнях, проте їх використання пов’язані з низкою проблем. Розглянемо основні труднощі, з якими стикаються вчені та інженери під час використання мов програмування у своїх розрахунках.

Продуктивність – одна з ключових проблем при виконанні наукових та інженерних розрахунків. Багато мов загального призначення, таких як Python та R, спочатку не були розроблені з урахуванням високопродуктивних обчислень. Це призводить до необхідності використання додаткових інструментів, таких як компілятори Just-In-Time (наприклад, Numba для Python) або паралельні обчислення, що ускладнює розробку та збільшує ймовірність помилок.

Сучасні наукові завдання часто вимагають обробки величезних обсягів даних та використання суперкомп’ютерів. Масштабованість додатків – ще одна серйозна проблема. Багато традиційних мов програмування не підтримують розподілені обчислення "з коробки". Це вимагає використання додаткових фреймворків та бібліотек, таких як MPI (Message Passing Interface) або Hadoop, що додає складність у розробку та підтримку коду.

Для наукових розрахунків важлива як швидкість виконання, так і точність обчислень. Деякі мови та їх реалізації можуть вводити чисельні помилки через

особливості роботи з плаваючою комою. Необхідність контролю за точністю потребує додаткових зусиль з боку розробників.

Використання мов програмування в наукових та інженерних розрахунках неминуче пов’язано з низкою проблем, таких як вибір відповідної мови, продуктивність, масштабованість, сумісність, точність обчислень та підтримка. Розуміння цих проблем та пошук оптимальних рішень для кожного конкретного завдання є ключовими факторами успішного виконання наукових та інженерних проєктів. Для цього необхідні не тільки технічні навички, а й глибоке розуміння специфіки завдань і можливостей використовуваних інструментів.

1.2 Огляд літературних джерел з використання мов програмування в інженерно-математичних та наукових розрахунках

Чи потрібне вміння програмувати інженерам та науковим співробітникам? Відповідь на це питання не така проста. З одного боку, сьогодні є безліч спеціалізованих програм, призначених для проведення розрахунків практично у всіх галузях науки та техніки. З іншого боку, як слушно зазначається у роботі [1], завжди є прості завдання, вирішувати які з використанням наявних програм незручно. Крім того, найчастіше виникає необхідність перевірити нову ідею. Зрештою, виникає питання: хто розроблятиме наступне покоління наукових та інженерних програм для комп’ютера?

Як зазначається у статті [2], довгий час основними мовами високопродуктивних обчислень були Fortran, C і C++. Причинами цього є їх сумісність, здатність компіляції високоефективний машинний код і достатній рівень абстракції порівняно з програмуванням мовою асемблера.

Порівняно недавно було запропоновано нову мову програмування Julia (www.julialang.org) [3, 4], розробники якої спробували поєднати в одній мові

високі швидкодія обчислень і ефективність розробки програми. Мова Julia – це кросплатформна компілювана мова програмування, що вільно розповсюджується (ліцензія MIT) з динамічною типізацією, яка має ряд переваг. У Julia реалізована можливість JIT-компіляції на основі LLVM [5]. Компіляція Just-in-Time (JIT) дозволяє забезпечити одночасно виразність сучасних інтерпретованих мов та продуктивність таких мов, як C та Fortran.

Мова Julia позиціонується розробниками як мова для вирішення наукових і технічних завдань. Його широко використовують математики. У роботі [6] наведено написану мовою Julia бібліотеку Optim.jl, яка призначена для вирішення задач оптимізації. Бібліотека підтримує обчислення з використанням комплексних чисел та розрахунки із заданою точністю. У ній реалізовані методи оптимізації, що не використовують похідні функції, а також методи оптимізації першого та другого порядків. До момента написання роботи [6] в бібліотеці були реалізовані головним чином методи оптимізації без обмежень, які використовувалися для вирішення наукових та інженерних завдань.

У роботі [7] йдеться про відкриту предметно-орієнтовану вбудовану в Julia мову моделювання JuMP, за допомогою якого дослідники можуть вирішувати завдання оптимізації (лінійні, нелінійні, з обмеженнями і без). JuMP дозволяє вирішувати завдання оптимізації дуже ефективно завдяки особливостям мови Julia.

У статті [8] описана бібліотека оптимізаційного моделювання Convex, завдання якої полягає в перекладі проблеми, представленої на зрозумілому досліднику мовою, в абстрактне синтаксичне дерево, що описує проблему, що дозволяє Convex перевірити, чи виконуються правила дисциплінованого опуклого програмування і передати вирішення проблеми. Ці операції виконуються в Julia з використанням множинної диспетчеризації. Вибір вирішувача здійснюється в Convex автоматично. Показано, що завдання математичної оптимізації можна із зручної для читання людиною форми

перетворити на абстрактне синтаксичне дерево, що представляє це завдання, з використанням набору функцій. Наприклад, Convex може перевірити, чи задача є опуклою, застосовуючи правила дисциплінованого опуклого програмування [9].

У роботі [10] наведено програмну платформу StochasticProgram5.jl з відкритим вихідним кодом, написану мовою Julia і призначену для стохастичного програмування. Платформа включає інструменти для моделювання та алгоритми оптимізації.

Бібліотека DifferentialEquations.jl для вирішення диференціальних рівнянь Julia описана в [11]. Перелік розв'язуваних з її допомогою завдань включає, зокрема, звичайні, стохастичні, алгебраїчні та гібридні диференціальні рівняння, а також диференціальні рівняння приватних похідних.

Алгоритм методу розщеплення конічних операторів (COSMO) для квадратичної цільової функції і конічних обмежень, а також його програмна реалізація, що вільно розповсюджується, на мові Julia представлені в [12]. Особливості алгоритму дозволяють використовувати його для вирішення завдань оптимізації великої розмірності, наприклад, напіввизначених програм оптимізації портфеля, теорії графів і робастного управління.

У статті [13] наводиться опис RAFF.jl – відкритої бібліотеки мовою Julia, призначеної для робастної апроксимації експериментальних даних. Особливості алгоритму, реалізованого бібліотеці, дозволяють визначати можливі помилкові дані (викиди), завдяки чому вдається зменшити похибку апроксимації.

У [14] розказано про розширення можливостей моделювання з компонентом, створені з використанням мови програмування Julia. Модель середовища складається із структури даних та набору функцій, призначених для обробки цих даних.

Стаття [15] присвячена опису навчальної бібліотеки щодо гідродинамічних розрахунків мовою Julia. Розглянуто різні концепції, пов'язані з просторовою та

тимчасовою дискретизаціями, явними та неявними числовими схемами, багатокроковими числовими схемами, чисельними методами та ітераційними вирішувачами у обчислювальній гідродинаміці.

JuliaFEM – бібліотека з відкритим вихідним кодом мовою Julia, призначена для проведення паралельних розрахунків методом кінцевого елемента на комп'ютерному кластері [16]. При розробці JuliaFEM було використано бібліотеку автоматичного диференціювання ForwardDi [17] для розрахунку градієнтів, якобіанів та гесіанів. Приклад використання бібліотеки JuliaFEM для модельного розрахунку власних частот кронштейна наведено в [18].

Роботи [19] присвячені бібліотеці GaussianProcesses.jl та , яка призначена для чисельного моделювання гаусових процесів. Для оптимізації параметрів процесу використано бібліотеку Optim.jl, яка надає кілька ефективних алгоритмів оптимізації без обмежень.

Розглянуто ряд джерел на деякі прикладні бібліотеки мови програмування Julia, призначені для вирішення наукових та технічних задач. Проведений аналіз свідчить про доцільність використання мови програмування Julia як у навчальному процесі при підготовці фахівців інженерних та природничих спеціальностей, так і безпосередньо в інженерній практиці.

1.3 Компілятори та програмні інструменти для інженерно-математичних і наукових розрахунків

Сучасні інженерно-математичні та наукові розрахунки вимагають потужних інструментів, здатних обробляти великі обсяги даних та виконувати складні математичні операції. Розглянемо деякі основні програмні платформи та технології, що використовуються в наукових та інженерних розрахунках.

MATLAB (Matrix Laboratory) – одна з найпопулярніших платформ для інженерних та наукових розрахунків. MATLAB [20] надає широкий набір інструментів для матричних операцій, побудови графіків та виконання чисельних методів.

Компілятор MATLAB використовується для перетворення програм MATLAB у додатки та бібліотеки, які можуть працювати незалежно від системи. Використовується для створення автономних C та C++ додатків на платформах Windows, UNIX та Macintosh, C і C++ бібліотек спільного використання (підключаються бібліотек, або dll, на Windows). Конфігурація зовнішнього компілятора відбувається автоматично під час виконання команди mbuild -setup. Файл опцій містить параметри налаштування та прапори, які керують роботою зовнішнього C/C++ компілятора. Для створення файлу опцій система MATLAB має готові конфігуровані файли опцій, які наведені нижче на рисунку 1.

lcccompp.bat	Lcc C, Version 2.4.1 (включен в MATLAB)
msvc60compp.bat	Microsoft Visual C/C++, Version 6.0
msvc71compp.bat	Microsoft Visual C/C++, Version 7.1
msvc80compp.bat	Microsoft Visual C/C++, Version 8.0
bcc55freecompp.bat	Borland C/C++ (free command-line tools) Version 5.5
bcc55compp.bat	Borland C++ Builder 5
bcc56compp.bat	Borland C++ Builder 6

Рисунок 1 – Файли опцій компілятора MATLAB

Чисельні методи представлено в MATLAB через базові операції розв’язання систем лінійних та нелінійних рівнянь розглянуто функції обчислення кінцевих різниць, чисельного диференціювання, чисельного інтегрування, тріангуляції та апроксимації Лапласіана, а також рішення звичайних диференціальних рівнянь. Векторна обробка даних забезпечує високу швидкість обчислень, і, в більшості випадків, рятує користувача від написання циклів і гарантує необхідну точність.

Переваги MATLAB:

– простота використання та інтуїтивно зрозумілий інтерфейс;

– велика кількість вбудованих функцій та інструментів;

– підтримка різних типів моделювання та симуляцій.

Недоліки MATLAB:

– висока ціна ліцензії;

– обмежена продуктивність під час обробки великих обсягів даних.

Python – універсальна інтерпретована мова програмування, яка завдяки бібліотекам NumPy та SciPy стала популярним інструментом для наукових та інженерних розрахунків [21]. NumPy забезпечує підтримку масивів та матриць, а SciPy надає додаткові функції для чисельних методів та наукових обчислень. Містить високооптимізовані реалізації, написані мовами низького рівня, такими як Fortran, C і C++. Насолоджуйтеся гнучкістю Python зі швидкістю скомпільованого коду.

Еталонною реалізацією Python є інтерпретатор CPython, який підтримує більшість платформ, що активно використовуються, є стандартом де-факто мови. Він розповсюджується під вільною ліцензією Python Software Foundation License, що дозволяє використовувати його без обмежень у будь-яких додатках, включаючи пропрієтарні.

Сучасна бібліотека SfePy (www.sfepy.org) – це програмне забезпечення для розв’язування систем пов’язаних диференціальних рівнянь із частинними похідними (PDE) методом скінченних елементів у 1D, 2D та 3D постановці. Його можна розглядати як розв’язувач PDE із чорною скринькою, який можна використовувати для створення спеціальних програм. Приклад реалізації задачі гідродінаміки чисельними метдами наведено на рисунку 2.

SfePy – це безкоштовне програмне забезпечення, випущене за новою ліцензією BSD і спирається на бібліотеки NumPy та SciPy.

Переваги Python:

– безкоштовність та відкритий вихідний код;

– величезна спільнота та велика документація;

– легкість інтеграції з іншими мовами та інструментами;

– має потужні бібліотеки чисельних методів на основі кінцевих елементів.

Недоліки Python:

– найменша продуктивність у порівнянні з компілюваними мовами;

– необхідність додаткових налаштувань для високопродуктивних обчислень.

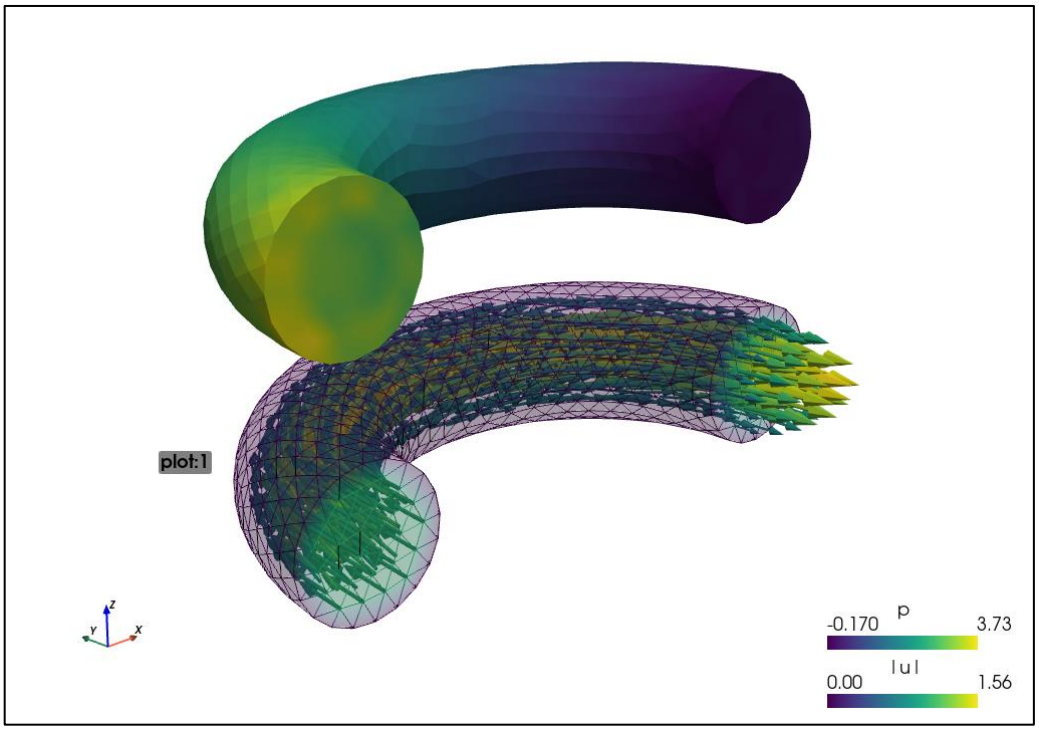


Рисунок 2 – Реалізація задачі Навье-Стокса в Python
за допомогою бібліотеки SfePy

R – інтерпретована мова програмування та середовище розробки, спеціально створені для статистичних обчислень та графіки. R широко використовується в біоінформатиці, соціології та інших науках, що потребують обробки більших обсягів статистичних даних [22].

Мова та середовище доступні під ліцензією GNU GPL; поширюються як вихідних кодів, і навіть відкомпільованих додатків під низку операційних систем: FreeBSD, Solaris та інші дистрибутиви Unix і Linux, Windows, macOS.

У базову поставку R включений основний набір пакетів, а за станом на 2024 доступно більше 20304 пакетів.

На мові R доступна реалізація методу скінченних елементів для моделювання балочних конструкцій і 2D геометрії з використанням трикутників постійної деформації. Застосовуються властивості матеріалу та граничні умови (навантаження та обмеження) для створення моделі кінцевих елементів. Модель в R створює напруги, деформації та вузлові зміщення. На рисунку 3 наведено приклад реалізації створення повної кінцево-елементної моделі, використовуючи напругу для заданої двовимірної сітки за заданих граничних умов (обмеження та навантаження).

```
data(triMesh)
data(dime)

meshP = triMesh$MeshPts$p
meshT = triMesh$MeshPts$T
centroid = triMesh$Centroids
Y = matrix(20e9, nrow = NROW(meshT))
Nu = matrix(0.45, nrow = NROW(meshT))
Thick = 0.001
DOF = 6
BoundConx = BoundCony = numeric(NROW(meshP))
BoundConx[1:NROW(meshP)] = BoundCony[1:NROW(meshP)] = 1
BoundConx[c(10, 11, 12)] = BoundCony[c(10, 11, 12)] = 0
SFShear = 0
SFTensile = 0
Length = dime$TrussLength
area = dime$Area
Fx = 10
Fy = 10

fea_stress = FEMStress.2d(meshP, meshT, centroid, BoundConx, BoundCony, SFShear, SFTensile,
            Length, area, Fx, Fy, Y, Nu, Thick)
```

Рисунок 3 – Код створення повної кінцево-елементної моделі в R

Переваги мови програмування R:

– спеціалізація на статистиці та аналітиці даних;
– багато пакетів для різних статистичних методів;
– безкоштовність та відкритий вихідний код;
– реалізована можливість розрахунків на основі метода кінцевих елементів.

Недоліки:

– відносно повільна продуктивність;

– обмежені можливості загальних інженерних розрахунків.

Julia – сучасна об’єктно-орієнтовна інтерпретована мова програмування, розроблена спеціально для високопродуктивних чисельних обчислень [23]. Вона поєднує зручність використання інтерпретованих мов, таких як Python, з продуктивністю компілюваних мов, таких як C і Fortran. Julia з лютого 2012 року розробляється в Массачусетському технологічному інституті, який за багатьма рейтингами є безперечним світовим лідером у галузі IT-технологій. Автори – Стефан Карпінськи, Джефф Безансон та Вірал Шах – прагнули створити інструмент для проведення великого об’єму обчислень із максимально можливою продуктивністю. З цієї причини у Julia передбачена підтримка хмар та паралельних обчислень.

Чисельні методи у Julia представлено рядом бібліотек JuliaFEM, Ferrite, Gridap, Trixi та інших, що постійно розвиваються. Геометрична модель може бути створена за допомогою відомих CAD-пакетів (CATYA, Autodesk Inventor, SolidWork). Повністю реалізовано можливість створення власних кінцевих елементів, граничних умов та рішення лінійних алгебраїчних рівняннь та діференційних рівнянь у приватних похідних (PDE), також є ряд інструментів для створення власних сіткових моделей, або імпорту вже існуючих із зовнішніх джерел. На рисунку 4 показано результат обчислення обтікання циліндра у потоці Incompressible Navier-Stokes equations).

Синтаксис Julia близький до синтаксису інших середовищ технічних обчислень, таких як Matlab та Octave. Базова бібліотека мови в основному написана тією ж мовою Julia, але також включає розвинені, кращі в своєму класі відкриті бібліотеки для лінійної алгебри, генерації випадкових чисел, обробки сигналів і рядкових змінних, написані на C і FORTRAN.

Також є можливість швидко встановити через вбудований менеджер низку зовнішніх пакетів, підготовлених спільнотою розробників Julia. Модуль мови IJulia, розроблений спільно співтовариствами Jupyter та Julia, забезпечує потужний вебінтерфейс так званого графічного блокнота (graphical notebook), добре знайомий користувачам Python, який поєднує код, форматований текст, математичні обчислення та мультимедійні ресурси в одному документі.

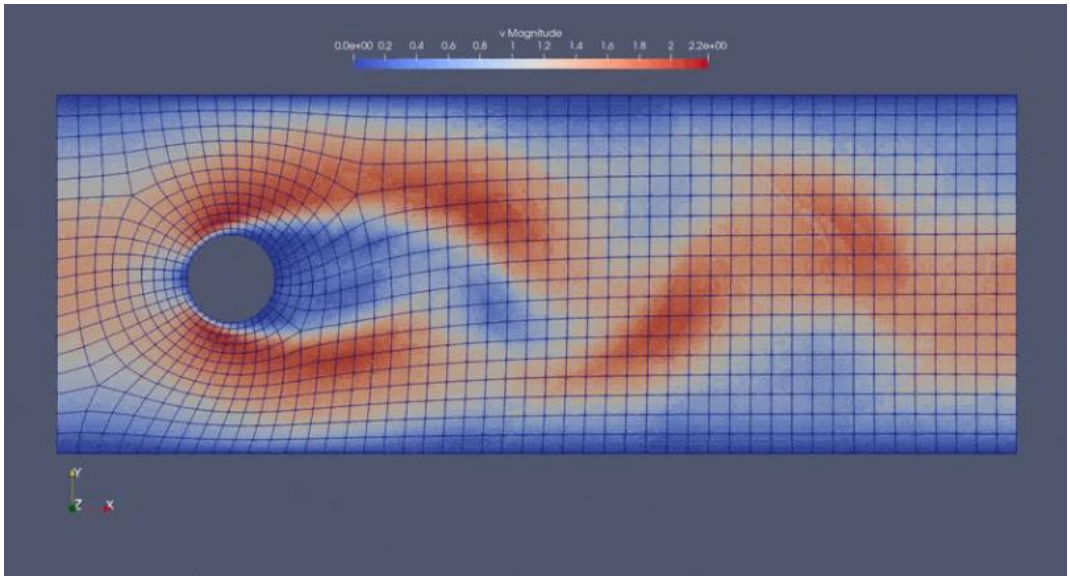


Рисунок 4 – Обчислення рівняннь Навье-Стокса в біліотеці Ferrite мови Julia

Основні переваги Julia:

– множинна диспетчеризація (мультиметод): забезпечення можливості визначати поведінку функції за різних комбінацій типів аргументів;
– динамічна типізація;
– продуктивність, що наближається до продуктивності статичних мов;
– вбудований менеджер пакунків;
– макроси та інші об’єкти метапрограмування;
– функції викликів Python: PyCall;
– функції прямої обробки дзвінків C без надбудов;
– потужні можливості оболонки для керування іншими процесами;

– є можливості забезпечення паралелізму та розподілених обчислень;
– ефективна підтримка кодування Unicode;
– ліцензія MIT: поширюється безкоштовно разом із вихідними кодами;
– велика кількість бібліотек для реалізації метода кінцевих елементів.

Недоліки Julia:

– відносно молода екосистема з меншою кількістю бібліотек порівняно з Python або R;
– обмежена підтримка в науковому співтоваристві.

FORTRAN – одна з найстаріших компільованих мов програмування, яка досі широко використовується в наукових розрахунках, особливо в задачах чисельного моделювання та високопродуктивних обчислень [24].

Мову було створено у період з 1954 по 1957 рік групою програмістів під керівництвом Джона Бекуса у корпорації IBM. Одна з переваг сучасного FORTRAN – велика кількість написаних на ньому програм та бібліотек підпрограм [25] – переважно на старих версіях мови, різних математичних бібліотек для матричної алгебри та розв’язання систем лінійних рівнянь, бібліотеки для вирішення диференціальних рівнянь, інтегральних рівнянь та їх систем, апроксимації функцій, спеціальних функцій, швидких перетворень Фур’є статистики та інших математичних дисциплін. Ці бібліотеки постачаються, зазвичай, з компілятором. Ряд таких пакетів створювався протягом десятиліть і популярний у науковому середовищі до сьогодення, наприклад – Міжнародна математична бібліотека підпрограм (IMSL) [26].

Стандарт ANSI X3.9-1966 [27] визначив мову FORTRAN (заснований на FORTRAN IV фірми IBM, яка служила стандартом де-факто). Згодом на нього почали посилатися як на FORTRAN66, хоча багато хто продовжував називати його FORTRAN IV, мову, якою стандарт був значною мірою заснований.

Чисельні методи у FORTRAN представлено низкою підпрограм (subroutines) та компілятором FORTRAN90 Compiler, реалізовано методи

Гальоркіна, моделі пластичності матеріалів, рішення диференційних рівняннь у приватних похідних (PDE) та інші. На рисунку 5 наведено рішення вітрового обтікання будівлі за допомогою підпрограм FORTRAN.

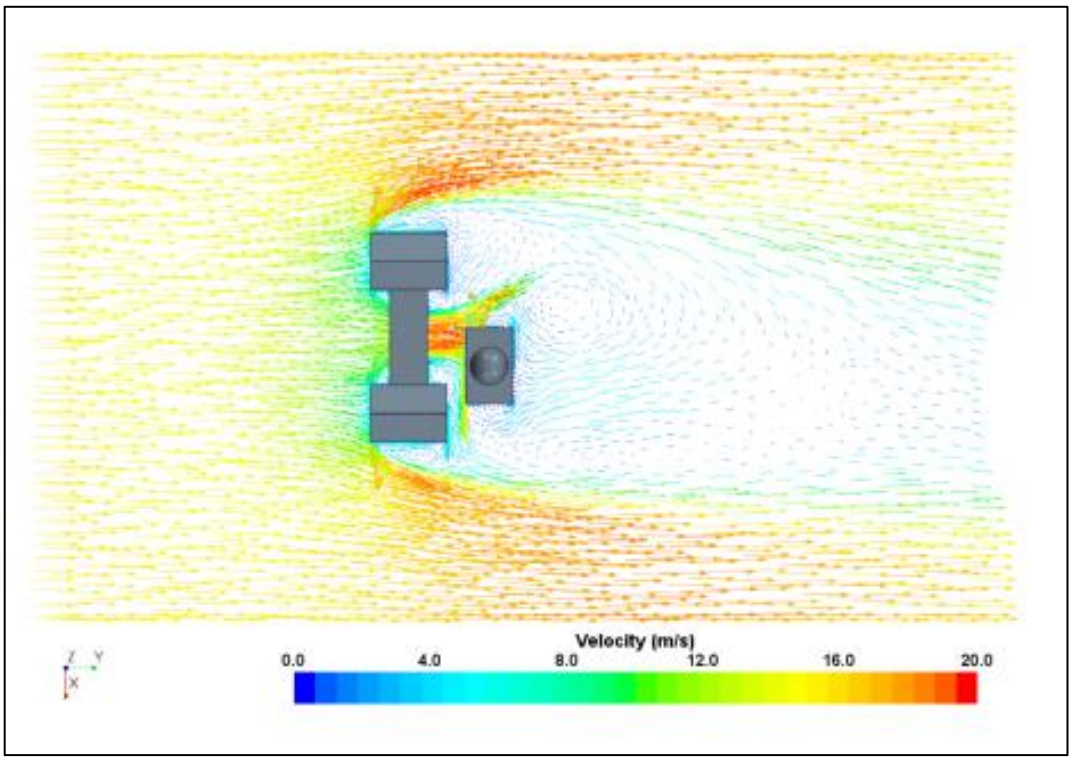


Рисунок 5 – Моделювання вітрового обтікання за допомогою FORTRAN90 та платформи SALOME

Основні переваги мови FORTRAN:

- висока продуктивність;
- має великий набір інструментів, доступних на безкоштовній основі;
- оптимізація для чисельних методів та наукових розрахунків;
- широке застосування у наукових та інженерних галузях;
- мова використовує статичну та сильну типізацію, що дозволяє компілятору відловлювати багато помилок програмування на ранніх стадіях; це також дозволяє компілятор генерувати ефективний двійковий код;

– простий у вивченні та використанні: виразити більшість математичних та арифметичних операцій над великими масивами так само просто, як написати їх у вигляді рівнянь на дошці;
– мова дозволяє писати програми, використовуючи підхід, який найкраще підходить для вирішення вашого завдання: імперативний, процедурний, орієнтований на масиви, об’єктно-орієнтований чи функціональний;
– вбудована підтримка паралельних обчислень з інтуїтивно зрозумілим синтаксисом, подібним до синтаксису масивів, для обміну даними між процесорами;
– мова стандартизована для різних платформ, а її нові версії відрізняються сумісністю з попередніми варіантами;
– має набір засобів (трансляторів) для перетворення на машинні системи різних комп’ютерів.

Недоліки FORTRAN:

– застарілий синтаксис;
– жорсткі вимоги до формату коду: наприклад, версія Fortran77 має обмеження по довжині рядка, а на початку потрібно було робити відступ;
– обмежені можливості для сучасних завдань обробки даних;
– невеликий набір команд управління програмною структурою;
– слабкий набір засобів для опису даних.

1.4 Бібліотеки для наукових обчислень

У роботі розглядається програмне забезпечення з відкритим вихідним кодом з урахуванням високої доступності для дослідників.

Ряд можливостей існуючих бібліотек наукових обчислень, корисних під час вирішення завдань математичного моделювання, наведено у таблиці 1.

Таблиця 1 складена з урахуванням трьох МВР – Octave, Python та Julia. Синтаксис та функції Octave майже повністю сумісні з MATLAB, що дозволяє легко переносити код між двома платформами. Відповідно до Таблиці 1, бібліотеки наукових обчислень надають схожі можливості розробки наукового програмного забезпечення із застосуванням кожної з перелічених мов програмування. Проте слід зазначити деякі відмінності.

Зокрема, для Octave існує лише один пакет (OCL) для виконання обчислень на спеціалізованих процесорах. Реалізація цього типу обчислень також можлива із застосуванням викликів мовою C, проте це є трудомістким для нефахівців у галузі програмування. Крім того, Octave не орієнтований на застосування у галузі машинного навчання. Зокрема, у Таблиці 1 наводиться назва репозиторію з прикладами, проте прикладні бібліотеки відсутні.

В Octave доступні дві функції, які можна використовувати для розв’язування диференціальних рівнянь. Реалізація вбудована в основні функції Octave. Використовувані алгоритми базуються на розв’язниках звичайних диференціальних рівнянь, написаних мовою Fortran. Доступні також функції символьного обчислення, включаючи стандартні інструменти системи комп’ютерної алгебри, такі як алгебраїчні операції, обчислення, розв’язування рівнянь, перетворення Фур’є та Лапласа, арифметика зі змінною точністю та інші функції. Передбачена сумісність з іншими символічними наборами інструментів.

Таблиця 1 – Порівняння мов програмування для інженерних та наукових розрахунків

Мова/можливості	**Octave**	**Python**	**Julia**
Лінійна алгебра	Вбудовано	NumPy/SciPy	Вбудовано
Чисельне рішення диференціальних рівняннь	Вбудовано	SciPy	DifferentialEquations.jl

Продовження табл. 1

Мова/можливості	**Octave**	**Python**	**Julia**
Оптимізація	optim	SciPy	BlackBoxOptim.jl
Датафрейми	dataframes	pandas	Dataframes.jl
JSON	pkg-json	Вбудовано	JSON.jl
Символьна алгебра	Symbolic (SymPy)	SymPy	Symbolics.jl
Статистика	statistics	SciPy	StatsBase.jl
Робота з сигналами	signal	SciPy	SignalAnalysys.jl
Графи	Не відомо	NetworkX	Graphs.jl
Візуалізація	Вбудовано	matplotlib, plotly	Plots.jl
GPGPU	ocl	PyOpenCL, PyCUDA, arrayfire и др.	JuliaGPU
Нечітка логика	fuzzy-logic-toolkit	scikit-fuzzy	FuzzyLogic.jl
Машинне навчання	machine-learning-octave (репозиторій з прикладами)	scikit-learn, pytorch та інші	MLJ.jl, Flux.jl

Large Time/Frequency Analysis Toolbox (LTFAT) – це набір інструментів Matlab/Octave для роботи з частотно-часовим аналізом і синтезом. Він призначений і як навчальний, і як обчислювальний інструмент. Набір інструментів надає велику кількість лінійних перетворень.

1.5 Висновки з розділу 1

Наукові та інженерні розрахунки вимагають виконання складних математичних операцій, обробки великих обсягів даних та моделювання комплексних фізіко-математичних систем.

Для вирішення цих задач є широкий спектр різних апаратних засобів, починаючи від персональних комп'ютерів та закінчуючи обчислювальними кластерами та суперкомп'ютерами.

Існуючі мови програмування мають свої сильні та слабкі сторони. Python, який популярний завдяки своїй простоті та широкому набору бібліотек для наукових обчислень, таких як NumPy та SciPy, має вузьке місце завдяки його інтерпретованості. З іншого боку, мови, такі як C/C++ та Fortran, забезпечують високу продуктивність, але потребують більшого часу на розробку та тестування.

Продуктивність та недостатня точність – головні проблеми використання мов загального призначення при виконанні наукових та інженерних розрахунків.

Julia – сучасна мова програмування, розроблена спеціально для високопродуктивних чисельних обчислень. Вона поєднує зручність використання інтерпретованих мов, таких як Python, з продуктивністю компілюваних мов, таких як C і Fortran.

Julia – найбільш оптимальна мова програмування для для проведення великого об'єму обчислень із максимально можливою продуктивністю. З цієї причини у Julia передбачена підтримка хмар та паралельних обчислень. Також включає розвинені відкриті бібліотеки для лінійної алгебри, генерації випадкових чисел, обробки сигналів і рядкових змінних.

2 ДОСЛІДЖЕННЯ ЗАСОБІВ ЧИСЕЛЬНОГО МОДЕЛЮВАННЯ ДЛЯ ІНЖЕНЕРНО-МАТЕМАТИЧНИХ ТА НАУКОВИХ РОЗРАХУНКІВ МОВОЮ JULIA

2.1 Мова програмування Julia для рішення інженерно-математичних та наукових задач

Мова Julia – це крос-платформна компілювана мова програмування, що вільно розповсюджується, під ліцензією MIT з динамічною типізацією, яка має ряд переваг. Компіляция Just-in-Time (JIT) дозволяє забезпечити одночасно переваги сучасних інтерпретуємих мов і продуктивність таких мов, як C та FORTRAN. Компілятор JIT виконує компіляцію під час першого запуску програмного коду, витягуючи з тексту інформацію, яка не вказано програмістом явно, та використовуючи її для оптимізації машиного коду.

Однією із причин вибору Julia є послідовна реалізація принципу однієї мови. Справа в тому, що у своєму розвитку мови програмування для наукових та інженерних завдань розійшлися за двома напрямками: мови для створення високопродуктивних програм (FORTRAN, C, C++) та мови для швидкого прототипування та використання непрограмістами (Matlab, R, Python).

Базова бібліотека мови в основному написана на мові Julia, але також включає в себе розвинені, кращі у своєму класі відкриті бібліотеки для лінійної алгебри, генерації випадкових чисел, обробки сигналів і рядкових змінних, написані на C і Fortran.

Основні можливості мови Julia:

– множинна диспетчеризація (мультиметод): забезпечення можливості визначати поведінку функції за різних комбінацій типів аргументів;
– динамічна типізація;

– висока продуктивність, що наближається до продуктивності статичних мов;
– вбудований менеджер пакетів;
– макроси та інші об’єкти метапрограмування;
– функції виклику Python: PyCall;
– функції прямої обробки викликів C без надбудов;
– потужні можливості оболонки для керування іншими процесами;
– можливість забезпечення паралелізму та розподілених обчислень;
– ефективна підтримка кодування Unicode;
– ліцензія MIT: поширюється безкоштовно разом із вихідними кодами.

2.1.1 Спеціалізовані бібліотеки мови julia

Екосистема мови Julia містить понад 10 000 різноманітних бібліотек, які зареєстровані у влоасному репозіторії, однак ми розглянемо лише ключові бібліотеки, які частіше за всі інші використовуються на практиці.

Мова програмування Julia має у своєму складі низку бібліотек для роботи з методами кінцевих єлементів – JuliaFEM, Gridap, Ferrite, Trixi та інші, що значно полегшує інженеру чи науковцю створення власних програмних кодів для обчисленння задач FE-методами. Розглянемо дві найбільш доступні для сприйняття бібліотеки JulaiFEM та Ferrite.

Бібліотека програмного забезпечення JuliaFEM [28] – це структура, яка дозволяє розподілено обробляти великі моделі кінцевих елементів у кластерах комп’ютерів за допомогою простих моделей програмування. Його розроблено для масштабування від одного сервера до тисяч машин, кожна з яких пропонує локальні обчислення та зберігання. Усі фізичні моделі є нелінійними, з яких лінеаризація проводиться як окремі випадки. Функції форми, також відомі як

базисні функції, інтерполяційні поліноми тощо. Зазвичай невідома польова змінна інтерполюється в JuliaFEM з вузлових значень елемента за допомогою безперервних функцій (2.1).

$$u(\xi, t) = \sum_i N(\xi, t) u_i[t] \tag{2.1}$$

У бібліотеці реалізовано стандартні функції форми Лагранжа.

Доступні також лінійні функції форми:

– Seg2 (2-node segment);
– Tri3 (3-node triangle);
– Quad4 (4-node quadrangle);
– Tet4 (4-node tetrahedron);
– Pyr5 (5-node pyramid);
– Wedge6 (6-node wedge);
– Hex8 (8-node hexahedra).

Квадратичні та біквадратичні функції форми:

– Seg2 (2-node segment);
– Tri3 (3-node triangle);
– Quad4 (4-node quadrangle);
– Tet4 (4-node tetrahedron);
– Pyr5 (5-node pyramid);
– Wedge6 (6-node wedge);
– Hex8 (8-node hexahedra).

У промислових обчисленнях більший робочий процес зазвичай містить кілька різних аналізів. Порівняно з деякими комерційними програмами FEM, наприклад ABAQUS, аналіз можна розглядати як крок (step), але в більш загальному сенсі. Оскільки Julia є справжньою мовою програмування, можна створити, наприклад, цикл оптимізації, поєднуючи кілька різних аналізів для оптимізації моделі щодо певної мети проєктування.

В JuliaFEM елементи можна розглядати як «контейнери», що поєднують поля та базові функції, описані вище. Серед цього елемент містить інформацію про топологію (зв'язність) і правило чисельного інтегрування. Нові елементи створюються шляхом передачі базового типу (наприклад, Seg2, Quad4, Tet10, ...) елементу та списку ідентифікаційних номерів, де елемент топологічно з'єднаний. Налаштування полів для елемента виконується за допомогою команди update!, яка або створює нове поле, якщо поле з такою назвою ще не існує, або оновлює старе.

Доступ до точок інтегрування елемента здійснюється за допомогою функції get_integration_points.

Комбінуючи інтерполяцію та інтегрування, можна вже розрахувати локальні матриці одного елемента або, наприклад, площу та енергію деформації, рисунок 6.

```
update!(el, "lambda", 96.0)
update!(el, "mu", 48.0)

A = 0.0
W = 0.0
for ip in get_integration_points(el)
    detJ = el(ip, time, Val{:detJ})
    A += ip.weight * detJ
    ∇u = el("displacement", ip, time, Val{:Grad})
    E = 1/2*(∇u + ∇u')
    λ = el("lambda", ip, time)
    μ = el("mu", ip, time)
    W += ip.weight * ( λ/2*trace(E*E') + μ*trace(E)^2) * detJ
end

println("Area: $A")
println("Strain energy: $W")
```

Рисунок 6 – Приклад використання функції get_integration_point

Обчислення локальної матриці жорсткості для задачі Пуассона наведено на рисунку 7.

```
K = zeros(4,4)
update!(el, "coefficient", 36.0)
for ip in get_integration_points(el)
    dN = el(ip, time, Val{:Grad})
    detJ = el(ip, time, Val{:detJ})
    c = el("coefficient", ip, time)
    K += ip.weight * c*dN'*dN * detJ
end
K
```

Рисунок 7 – Обчислення локальної матриці для задачи Пуассона

Для більш ефективного використання пам’яті коду доцільно використовувати функції BasisInfo та element_info! який виділяє робочу пам’ять для обчислення всіх «базових речей» для однієї точки інтегрування, як-от якобіан, визначник якобіана, базис і його часткові похідні відносно еталонної конфігурації X, рисунок 8.

```
bi = BasisInfo(Quad4)
fill!(K, 0.0)
for ip in get_integration_points(el)
    J, detJ, N, dN = element_info!(bi, el, ip, time)
    c = el("coefficient", ip, time)
    K += ip.weight * c*dN'*dN * detJ
end
K
```

Рисунок 8 – Виділення пам’яті під обчислення

Після того, як глобальна збірка елементів та матриць для кожної проблеми готова, вони повинні якимось чином зібратися разом, зазвичай треба побудувати лінійну систему Ax=b, розв’язати систему рівнянь і, можливо, оновити рішення. Це завдання зазвичай (але не завжди) передбачає вирішення великих систем рівнянь.

Наприклад, типовий нелінійний квазістатичний аналіз має віигляд:

– ініціалізація задачі (якщо потрібно);

– збірка задачі;

– об’єднання вузлів для побудови лінійної системи рівнянь Ax=b;
– розв’язування лінійної системи рівнянь якомога ефективніше;
– оновлення рішення;
– постообробка та результати рішення;
– запис результатів до файла;
– перевірка сходимості, якщо не сходиться, повертаємость до збірки задачі.

Коли розрахункові моделі стають достатньо великими, крок 4 стає домінуючим у процесі вирішення. З цієї причини в JuliaFEM є абстрактний тип AbstractLinearSystemSolver, який може бути підкласом для створення власної стратегії вирішення. Ця стратегія може полягати, наприклад, у використанні вбудованих розв’язувачів Julia, MUMPS, ітераційних розв’язувачів тощо. І щоб зробити це якомога стандартнішим, є LinearSystem, що містить усі відповідні матриці, тобто

$$\begin{bmatrix} \boldsymbol{K} & \boldsymbol{C}_1 \\ \boldsymbol{C}_2 & \boldsymbol{D} \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{bmatrix} \tag{2.2}$$

При першої збірки лінійної системи, беруться матриці, які були введені в контексті задачі, рисунок 9.

```
ls = LinearSystem(5)
ls.K = sparse([  4.0  -1.0  -2.0  -1.0   0.0
                -1.0   7.0  -4.0  -2.0   0.0
                -2.0  -4.0  10.0  -1.0  -3.0
                -1.0  -2.0  -1.0   4.0   0.0
                 0.0   0.0  -3.0   0.0   3.0])
ls.C1 = sparse([ 1.0  0.0  0.0  0.0  0.0
                 0.0  0.0  0.0  0.0  0.0
                 0.0  0.0  0.0  0.0  0.0
                 0.0  0.0  0.0  1.0  0.0
                 0.0  0.0  0.0  0.0  0.0])
ls.C2 = ls.C1
ls.f = sparsevec([33.0, 33.0, 165.0, 33.0, 132.0])
nothing # hide
```

Рисунок 9 – Завдання матриць лінійної системи рівнянь

Як було сказано вище, Analysis – це тип, який містить усі проблеми та виконує з ними певну операцію. Як правило, це може бути вирішення проблеми статичної або динамічної рівноваги, аналізу власних значень, аналізу лінійних збурень або виконання певної постобробки для вже виконаного аналізу, як-от аналіз втоми, обчислення критичних напружень тощо.

У промислових обчисленнях більший робочий процес зазвичай містить кілька різних аналізів. Порівняно з деякими комерційними програмами FEM, наприклад, ABAQUS, Analysis можна розглядати як STEP, але в більш загальному сенсі. Оскільки Julia є справжньою мовою програмування, можна створити, наприклад, цикл оптимізації, поєднуючи кілька різних аналізів для оптимізації моделі щодо певної мети моделювання.

Існує не так багато правил щодо визначення нового аналізу. Він має бути підтипом AbstractAnalysis і реалізовувати функцію run! Як мінімальний робочий приклад, реалізація для статичного аналізу має такий вигляд, як на рисунку 10.

Зібравши все разом, типовий процес вирішення варіаційної задачі, що не залежить від часу, виглядає так, як на рисунку 11.

Бібліотека кінцево-елементного моделювання Ferrite [29] – це набір спеціалізованих інструментів, який надає функції для реалізації алгоритмів розрахунків методом кінцевих елементів у Julia. Це досить нова біліотека у Julia, яка зберігає математичні абстракції одночасно узагальнюючі математичні розрахунки для більшої гнучкості та точності.

Бібліотека Ferrite реалізує диференціальні рівняння у приватних похідних (Partial Differential Equations, PDE) і метод скінченних елементів (FEM). Мабуть, найпростішим з усіх PDE є стаціонарне, лінійне рівняння теплопровідності, також відоме як рівняння Пуассона. Розглянемо це рівняння як наочний приклад методу та продемонструємо, як відбувається перехід від сильної форми рівняння до слабкої форми, а потім, нарешті, до дискретної задачі реалізації методу кінцевих елементів у бібліотеці Ferrite.jl.

```
type Static <: AbstractAnalysis
    time :: Float64
end

function Static()
    return Static(0.0)
end

function run!(analysis::Analysis{Static})

    time = analysis.properties.time
    problems = get_problems(analysis)

    # assemble matrices for all problems
    N = 0 # size of resulting matrix
    for problem in problems
        assemble!(problem, time)
        N = max(N, size(problem.assembly.K, 2))
    end

    # create new LinearSystem and add assemblies to that
    ls = LinearSystem(N)

    for problem in problems
        ls.K += sparse(problem.assembly.K, N, N)
        ls.f += sparse(problem.assembly.f, N, 1)
        ls.C1 += sparse(problem.assembly.C1, N, N)
        ls.C2 += sparse(problem.assembly.C2, N, N)
        ls.D += sparse(problem.assembly.D, N, N)
        ls.g += sparse(problem.assembly.g, N, 1)
    end

    # solve linear system using e.g. LUSolver
    solve!(ls, LUSolver())

    # update solution back to problems
    u = Dict(i=>j for (i,j) in enumerate(ls.u))
    for problem in problems
        is_field_problem(problem) || continue
        field_name = get_unknown_field_name(problem)
        elements = get_elements(problem)
        update!(elements, field_name, time => u)
    end
end
```

Рисунок 10 – Реалізація статичного аналізу в JuliaFEM

```
X = Dict(1 => [0.0, 0.0],
         2 => [1.0, 0.0],
         3 => [1.0, 1.0],
         4 => [0.0, 1.0],
         5 => [2.0, 1.0])

el1 = Element(Quad4, [1, 2, 3, 4])
el2 = Element(Tri3, [3, 2, 5])
el3 = Element(Seg2, [3, 5])
elements = [el1, el2, el3]
update!(elements, "geometry", X)
update!(elements, "thermal conductivity", 6.0)
update!(el3, "heat flux", 264.0)
update!(el1, "heat source", 132.0)
problem = Problem(Heat, "test problem", 1)
add_elements!(problem, elements)

bel1 = Element(Seg2, [1, 4])
update!(bel1, "geometry", X)
update!(bel1, "temperature 1", 0.0)
bc = Problem(DirichletBC, "fixed", 1, "temperature")
add_elements!(bc, [bel1])

analysis = Analysis(Static, "static analysis of test problem")
add_problems!(analysis, [problem, bc])
run!(analysis)
```

Рисунок 11 – Типовий процес вирішення варіаційної задачі

Сильну форму рівняння теплопровідності можна записати так:

$$-\nabla \cdot q(u) = f \ \forall x \ \in \Omega, \tag{2.3}$$

де u – невідоме температурне поле, q – тепловой потік, f – внутрішне джерело тепла (теплової енергії), а Ω – це область, на якої визначено рівняння (розрахунковий домен). Щоб задача була гарно обумовлена, потрібно вказати, що відбувається на межі розрахункової області (домена), тобто вказати граничні умови Г. Існують різні типи граиничних умов, серед яких найпоширенішими є умови Діріхле (Dirichlet, [30]), що означає наявність рішення для u на деякій частині межі розрахункової області а також умови Неймана (Neumann, [31]), яке характеризує градієнт рішення ∇u. Для нашого прикладу формально це можно записати наступним чином:

$$u = \ u^P \ \forall x \in \Gamma_D, \tag{2.4}$$

$$q \cdot n = \ q^P \ \forall x \in \Gamma_N, \tag{2.5}$$

тобто температура визначається відомою функцією u^P на граничних умовах Діріхле Γ_D, тепловим потоком q^P на граничних умовах Неймана Γ_N та нормаллю n назовні розрахункової області (домена). Крівм цього необхідне ще одне рівняння, яке пов’язує температурне поле u та тепловий потік через розрахункову область. Таким рівнянням є закон теплопроводності Фурь’є [32]:

$$q(u) = \ -k\nabla u, \tag{2.6}$$

де k – теплопроводність матеріалу, але взагалі цей параметр може змінюватися у межах розрахункової області як фукнція координат, тобто k = k(x).

Розв’язок наведеного вище рівняння зазвичай обчислюється з відповідної слабкої форми. Шляхом множення рівняння на довільну контрольну функцію δu та інтегрування по розрахункової області та використовуючи часткове інтегрування, отримуємо слабку форму. Тепер нашу задачу можна сформулювати наступним чином:

$$\int_{\Omega} \nabla \delta u \cdot (k \nabla \mathrm{u}) \mathrm{d}\Omega = \int_{\Gamma_N} \delta u q^P d\Gamma + \int_{\Omega} \delta u f d\Omega \ \forall \ \delta u \in \mathbb{T}. \qquad (2.7)$$

Використання методу скінченних елементів для вирішення диференціальних рівнянь у приватних похідних зазвичай передує побудові дискретизації області Ω в кінцевий набір елементів (elements) або осередків (cells). Ця геометрічна дискретизація розрахункової області має назву сітки кінцевих елементів (mesh або grid).

Далі ми вводимо наближення скінченних елементів $u_h \approx \mathrm{u}$, як сумму з N вузлових функцій форми (nodal shape functions), де ми позначаємо кожну з цих функцій через ϕ_i і відповідні вузлові значення через u_i. Слід зауважити, що функції форми іноді називають базовими функціями або пробними функціями (test functions) і їх позначають як N_i. У цьому прикладі ми будемо апроксимувати тестову функцію таким же чином. Цей підхід відомий як метод кінцевих елементів Гальоркіна (Galerkin finite element method, [33]).

З урахуванням вищевикладеного, ми можемо написати аппроксимаційну функцію у точці x розрахункового домена Ω:

$$u_h(x) = \sum_{i=1}^{N} \phi_i(x) u_i, \ \delta u_h(x) = \sum_{i=1}^{N} \phi_i(x) \delta u_i \ . \qquad (2.8)$$

Оскільки тестові та пробні функції зазвичай вибираються таким чином, що вони будують основу деякого простору функцій (як базис векторного простору), іноді їх також називають базисними функціями (basis functions). У наступному

аргументі x опускається, щоб зберегти компактність позначення. Тепер ми можемо вставити наближення (2.8) в слабку форму, що призводить до наступної форми:

$$\sum_i^N \delta u_i \left(\sum_j^N \int_{\Omega_h} \nabla \phi_i \cdot \left(k \nabla \phi_j \right) d\Omega u_j \right) = \sum_i^N \delta u_i \left(\int_{\Gamma_N} \phi_i q^P d\Gamma + \int_{\Omega_h} \phi_i f d\Omega \right). \quad (2.9)$$

Оскільки це рівняння має виконуватися для довільних δu_h, однак особливо має виконуватися для конкретного вибору лише одного з вузлових значень δu_i, яке фіксується на 1, а всі інші коефіцієнти фіксуються на 0. Повторюючі цей аргумент для всіх і від 1 до T, отримаємо N лінійних рівняннь. Таким чином дискретну задачу можна записати у вигляді системи лінійних рівнянь:

$$K^T u = f, \quad (2.10)$$

де K^T – тангенційна матриця жорсткості (tangent stiffness matrix), u – вектор вузлових значеннь, а f – вектор нагрузок (force vector). Спеціальна назва зумовлена історичними причинами, оскільки метод скінченних елементів бере свій початок у механіці. Нарешті, нам також потрібно подбати про граничні умови Діріхле. Тепер розв’язання цієї системи рівнянь дає вузлові значення і, отже, наближення до справжнього (теоретичного) розв’язку.

2.2 Висновки з розділу 2

Мова програмування Julia має у своєму складі низку спеціалізованих бібліотек для розробки власних програмних застосунків з виконання розрахунків на основі методів кінцевих елементів.

Бібліотека JuliaFEM дозволяє розподілено обробляти великі моделі кінцевих елементів в кластерах комп'ютерів за допомогою простих моделей програмування. Її розроблено для масштабування від одного сервера до тисяч машин, кожна з яких пропонує локальні обчислення та зберігання. Основний принцип дизайну: все нелінійно. Усі фізичні моделі є нелінійними, з яких лінеаризація проводиться як окремі випадки. У бібліотеці реалізовано стандартні функції форми Лагранжа що значно полегшує подальшу розробку обчислювальних алгоритмів.

Бібліотека Ferrite – це набір інструментів кінцевих елементів, який надає функції для реалізації аналізу кінцевих елементів у Julia. Має більш просту реалізацію ніж JuliaFEM, тому більш легкий для розуміння та використання. У Ferrite використовуються еталонні форми для визначення ячійок розрахункової сітки, інтерполяції функцій (тобто функцій форми) і квадратурних правил.

Для аналізу та візуалізації результатів обчислень FE-інструменти Julia можна використовувати такі застосунки з відкритим вихідним кодом, як SALOME [34] та ParaView [35] з власними потужними інструментами обробки та графічної візуалізації. Це є дуже великим аргументом для інженерів та науковців, які тільки починають роботу над власними проєктами та не можуть собі дозволити «тяжке» комерційне програмне забезпечення на кшталт ANSYS або ABAQUS.

3 РОЗРОБКА СИСТЕМИ АВТОМАТИЗАЦІЇ ІНЖЕНЕРНИХ РОЗРАХУНКІВ ЗАСОБАМИ МОВИ JULIA

Розділ присвячений розробці функцуіоналу власного програмного коду на основі бібліотеки Ferrite.jl для розв’язання задач механіки суцільного середовища за допомогою методу кінцевих елементів (FEM). Результати розрахунків у вигляді відповідних полів напруг та деформацій візуалізуються в застосунках з відкритим вихідним кодом SALOME та ParaView.

3.1 Еталонні форми кінцевих елементів у бібліотеці Ferrite.jl

Еталонні форми у Ferrite використовуються для визначення кінцевих елементів сітки, інтерполяції функцій (тобто функцій форми) і правил квадратури. Наразі визначено наступні еталонні форми:

– елемент RefLine (балковий 2-вузловий або 3-вузловий);
– елемент RefTriangle (трикутний 3-вузловий або 6-вузловий);
– елемент RefQuadrilateral (чотирьохкутний 4-вузловий або 8-вузловий);
– елемент RefTetrahedron (тетраедричний 4-вузловий або 10-вузловий);
– елемент RefHexahedron (гексаедр 8-вузловий або 20-вузловий);
– елемент RefPrism (призматичний 6-вузловий або 12-вузловий);
– елемент RefPyramid (пиармидальний 5-вузловий або 13-вузловий).

Ferrite позначає сутності еталонної форми наступним чином:

– Vertex – 0 – вимірна сутнисть в еталонній формі;
– Edge – 1 – вимірна сутність в еталонній формі;
– Face – 2 – вимірна сутність в еталонній формі;
– Volume – 3 – сутність в еталонній формі.

Зауважимо, що вузол в Ferrite не може бути типу vertex. Вертекси задаються на кінцях ребер (edges), тоді як вузли можуть бути розташовані й на серединах ребер (наприклад в елементі другого порядку QuadraticLine). Для того, щоб отримати код незалежний від розмірності задачі, Ferrite позначає сутності за допомогою їхньої корозмірності, визначеної відносно застосованої розмірності опорної форми. Зокрема, Ferrite має сутності Cell – нуль-вимірна сутність подібно опорним формам та Facet – 1-корозмірна сутність, що визначає межі Cell.

Кожна еталонна форма має нумерацію своїх вершин, ребер і граней, де сутності ребер і граней визначаються черз номери вершин (див. рис. 12).

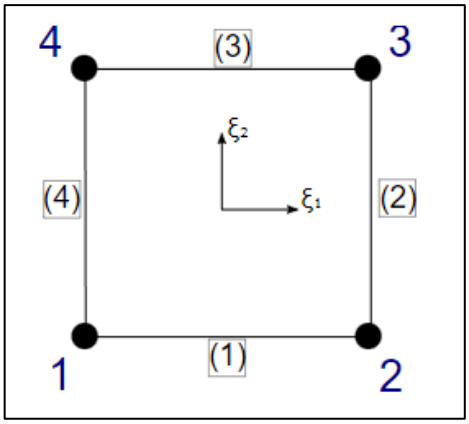


Рисунок 12 – Нумерація вершин, ребер та граней у Ferrite
на прикладі елемента RefQuadrilateral

Вершини (вертекси) елементу можно визначити наступним чином (див. рис. 13).

```
Ferrite.reference_vertices(RefQuadrilateral)
```

```
(1, 2, 3, 4)
```

Рисунок 13 – Визначення вершин елемента RefQuadrilateral

Ребра елемента визначаються, як показано на рисунку 14.

```
Ferrite.reference_edges(RefQuadrilateral)
```

```
((1, 2), (2, 3), (3, 4), (4, 1))
```

Рисунок 14 – Визначення ребер елемента RefQuadrilateral

Грані елемента відповідають наступній функції, як на рисунку 15.

```
Ferrite.reference_faces(RefQuadrilateral)
```

```
((1, 2, 3, 4),)
```

Рисунок 15 – Визначення граней елемента RefQuadrilateral

3.2 Типи об’єктів CellValues та FacetValues

Геометричне відображення процессу накладення (mapping) кінцевого елемента з опорних координат на дійсні координати показано на рисунку 16.

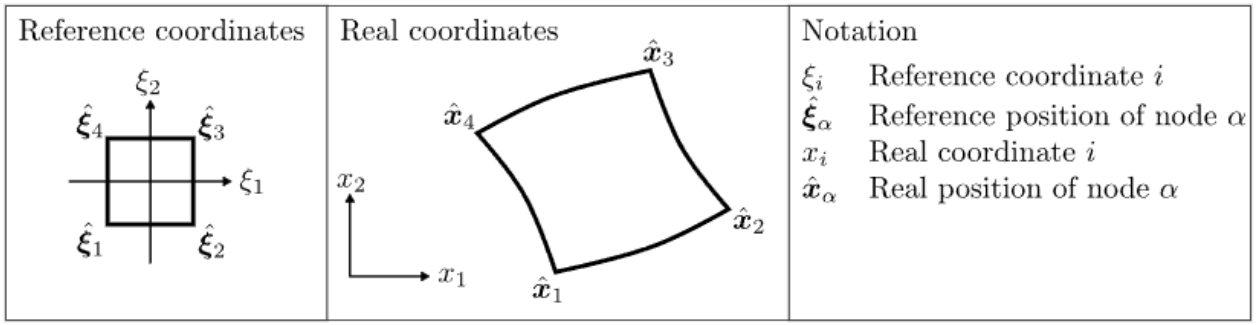


Рисунок 16 – Маппинг кінцевого елемента

Ключовим типом об’єктів у Ferrite є так звані FEValues, де найпоширенішими є CellValues і FacetValues. Ці об’єкти використовуються в підпрограмах елементів для запиту вагових параметров інтеграції, значень функцій форми та градієнтів і багато чого іншого. Щоб ці значення були

правильними, необхідно ініціалізувати їх для поточного кінцевого елементу за допомогою функції reinit!. Ця функція передає значення еталонної форми у фактичну форму кінцевого елементу. Об’єкт CellValues полегшує процес оцінки значень функцій форми, градієнтів функцій форми, значень вузлових функцій, градієнтів і розбіжностей вузлових функцій тощо в кінцевих елементах.

3.3 Ступені свободи (Degrees of Freedom, DOF)

Розподіл і нумерація ступенів свободи (dofs) обробляються спеціальною функцією DofHandler. DofHandler використовуватиметься для запиту інформації про ступені свободи (dofs).

Функція DofHandler базується ра розрахунковій сітці. Наприклад, створимо просту сітку з трикутників, а потім створимо DofHandler для цієї сітки (рис. 17).

```
grid = generate_grid(Triangle, (20, 20))
dh = DofHandler(grid)
```

```
DofHandler{2, Grid{2, Triangle, Float64}}
  Fields:
  Not closed!
```

Рисунок 17 – Створення функції DofHandler для трикутної сітки

Перш ніж розподіляти DOF, потрібно вказати поля (Fields). Поле – це просто невідома функція (функції), яку ми розв’язуємо. Щоб додати поле, нам потрібна назва (символ) і інтерполяція, що описує функції форми його поля. На рисунку 18 ми додаємо скалярне поле :p, інтерпольоване за допомогою лінійних (1 порядок) функцій форми на трикутнику, та векторне поле :u, також інтерпольоване лінійними функціями форми на трикутнику, але зведене до другого порядку, щоб вказати, що це векторне поле з двома компонентами (для 2D-задачі).

```
add!(dh, :p, Lagrange{RefTriangle, 1}())
add!(dh, :u, Lagrange{RefTriangle, 1}()^2)
```

```
DofHandler{2, Grid{2, Triangle, Float64}}
  Fields:
    :p, Lagrange{RefTriangle, 1}()
    :u, Lagrange{RefTriangle, 1}()^2
  Not closed!  Not closed!
```

Рисунок 18 – Створення полів для функції DofHandler

Нарешті, коли ми додали всі поля, нам потрібно закрити функцію DofHandler. Коли DofHandler закривається, він обробляє сітку та розподіляє всі DOF для полів, які ми додали (рис. 19).

```
close!(dh)
```

```
DofHandler{2, Grid{2, Triangle, Float64}}
  Fields:
    :p, Lagrange{RefTriangle, 1}()
    :u, Lagrange{RefTriangle, 1}()^2
  Dofs per cell: 9
  Total dofs: 1323
```

Рисунок 19 – Закриття DofHandler та розподіл dofs по сітці

3.4 Шаблон розрідженості та розріджені матриці

Важливою властивістю методу скінченних елементів є те, що він дає розріджені матриці для розв’язуваних лінійних систем.

У Ferrite розріджена структура лінійної системи залежить від багатьох факторів, таких як, наприклад слабка форма, дискретизація та вибір інтерполяції(й). Зрештою, це зводиться до того, як ступені свободи (DOF) поєднуються один з одним. Найпоширенішою причиною того, що два DOF з’єднуються, є те, що вони належать одному елементу. Однак це не гарантує, що

це призведе до зв’язку, оскільки це залежить від конкретної слабкої форми, яка дискретизується. Граничні умови та обмеження також можуть призвести до додаткових зв’язків DOF.

Шаблон розрідженості також служить «конструктором матриці». Коли всі записи вставляються в шаблон розрідженості, динамічна структура даних зазвичай перетворюється або «стискається» в формат розрідженої матриці, як, наприклад, формат стиснутого розрідженого рядка (CSR) або формат стиснутого розрідженого стовпця (CSC), де останній є типом розрідженої матриці за замовчуванням, реалізований у стандартній бібліотеці SparseArrays.jl. Ці матричні формати дозволяють виконувати швидкі операції лінійної алгебри, такі як розкладання на множники та множення матриці на вектор, які необхідні під час розв’язання лінійної системи.

У багатьох випадках явно не потрібно працювати з шаблоном розрідженості. Для базового використання (наприклад, коли потрібна лише одна матриця, коли не потрібно налаштовувати шаблон тощо) існують зручні методи allocate_matrix, які повертають матрицю безпосередньо. Більшість прикладів у цій роботі не стосуються явно шаблону розрідженості, оскільки достатньо базового методу, тобто можна використовувати методи allocate_matrix, які приймають DofHandler як вхідні дані, що показано наприклад, на рисуноку 20.

```
K = allocate_matrix(dh, ch)
```

Рисунок 20 – Метод allocate_matrix

3.5 Збірка локальних на глобальних матриць

Коли локальна матриця жорсткості та вектор сили були розраховані, вони повинні бути зібрані в глобальну матрицю жорсткості та глобальний вектор сили.

Це лише питання додавання локальної матриці та вектора до глобальної, у правильному місці. Розглянемо наприклад, об'єднання локальної матриці жорсткості ke та локального вектора сили fe в глобальні K і f відповідно. Їх слід зібрати в рядок/стовпець, який відповідає ступеням свободи кінцевого елементу, (рис. 21).

```
K[celldofs, celldofs] += ke
f[celldofs]           += fe
```

Рисунок 21 – Формування глобальних матриць системи

Тут celldofs – це вектор, що містить ступені свободи клітини (dofs). Наведений вище метод є дуже неефективним – особливо дорого індексувати розріджену матрицю K безпосередньо. Тому замість цього ми будемо використовувати асемблер, який допоможе зі складанням як глобальної матриці жорсткості, так і глобального вектора сили.

Для ефективного збирання в розріджену матрицю потрібен додатковий робочий простір. Ця робоча область виділена в спеціальному асемблері. Функція start_assemble використовується для створення саме асемблера (рис. 22).

```
A = start_assemble(K)
A = start_assemble(K, f)
```

Рисунок 22 – Створення збиральника (ассемблера) матриць

Тут K – глобальна матриця жорсткості, а f – глобальний вектор сили. Необов'язково передавати вектор сили в асемблер – іноді немає потреби збирати глобальний вектор сили.

Функція assemble! використовується для збирання вкладень елементів у асемблер. Наприклад, щоб зібрати дотичну жорсткість елемента ke та вектор сили елемента fe до асемблера A, можна використати такий код (рис. 23).

```
assemble!(A, celldofs, ke)
assemble!(A, celldofs, ke, fe)
```

Рисунок 23 – Використання функції assemble!

Зазвичай достатньо побудувати глобальну матрицю K один раз. Нижче наведено (див. рис. 24) код, як це зробити для задачі, що залежить від часу (transient solution).

```
K = allocate_matrix(dh)
f = zeros(ndofs(dh))

for t in 1:timesteps
    A = start_assemble(K, f) # start_assemble zeroes K and f
    for cell in CellIterator(dh)
        ke, fe = element_routine(...)
        assemble!(A, celldofs(cell), ke, fe)
    end
    # Apply boundary conditions and solve for u(t)
    # ...
end
```

Рисунок 24 – Побудова глобальної матриці K для нестаціонарної задачі

3.6 Граничні та початкові умови (Boundary Conditions)

Кожне рівняння з частинними похідними (PDE) супроводжується граничними умовами. Існують різні типи граничних умов, і з ними потрібно працювати по-різному. Нижче ми обговоримо, як обробляти найпоширеніші з них, граничні умови Діріхле та Неймана, і як це зробити за допомогою бібліотеки Ferrite.

При використанні умов Діріхле невідомому полю приписують задане значення. Для дискретного FE-розв’язку це означає, що є деякі ступені свободи, які є фіксованими. Для обробки граничних умов Діріхле у Ferrite використовується функція ConstraintHandler (рис. 25).

```
ch = ConstraintHandler(dh)
```

Рисунок 25 – Створення обробника обмежень

Щоб створити обмеження Діріхле, нам потрібно вказати назву поля, частину межі та функцію для обчислення заданого значення (рис. 26).

```
dbc1 = Dirichlet(
    :u,                         # Name of the field
    getfacetset(grid, "left"), # Part of the boundary
    x -> 1.0,                   # Function mapping coordinate to a prescribed value
)
```

Рисунок 26 – Створення умови Діріхле

За замовчуванням обмеження додається до всіх компонентів заданого поля. Щоб додати обмеження до вибраних компонентів, конструктору слід передати четвертий аргумент із компонентами. Ось приклад (рис. 27), коли обмеження додається до компонентів 1 і 3 векторного поля u.

```
dbc2 = Dirichlet(
    :u,                         # Name of the field
    getfacetset(grid, "left"), # Part of the boundary
    x -> [0.0, 0.0],            # Function mapping coordinate to prescribed values
    [1, 3],                     # Components
)
```

Рисунок 27 – Додавання обмежень Діріхле на обраний компонент

Додавання обмежень до обробника обмежень виконується за допомогою функціїї add! (рис. 28).

```
add!(ch, dbc1)
add!(ch, dbc2)
```

Рисунок 28 – Додавання обмежень до обробника обмежень

Наприкінці, як і для обробника ступенів свободи dof, потрібно використати функцію close! щоб завершити обробник обмежень. Внутрішньо, у коді, ця функція обчислить ступені свободи, які відповідають доданим раніше обмеженням.

Якщо одне або кілька обмежень залежать від часу, тобто вони визначені як f (x, t), задані значення можна перерахувати на кожному новому прирості часу Δt, викликавши функцію update! з відповідним значенням часу (рис. 29).

```
for t in 0.0:0.1:1.0
    update!(ch, t) # Compute prescribed values for this t
    # Solve for time t...
end
```

Рисунок 29 – Створення обмежень у нестаціонарному рішенні

На частині границі Неймана ми знаємо дещо про градієнт розв'язку. Нижче описано два різні методи їх застосування.

Граничні умови Неймана можна додати шляхом ітерації по відповідному фасету за допомогою спеціальної функції FacetIterator. Для скалярного поля це можна зробити наступним чином (рис. 30).

```
grid = generate_grid(Quadrilateral, (3,3))
dh = DofHandler(grid); push!(dh, :u, 1); close!(dh)
fv = FacetValues(QuadratureRule{RefQuadrilateral}(2), Lagrange{RefQuadrilateral, 1}())
f = zeros(ndofs(dh))
fe = zeros(ndofs_per_cell(dh))
qn = 1.0    # Normal flux
for fc in FacetIterator(dh, getfacetset(grid, "right"))
    reinit!(fv, fc)
    fill!(fe, 0)
    for q_point in 1:getnquadpoints(fv)
        dΓ = getdetJdV(fv, q_point)
        for i in 1:getnbasefunctions(fv)
            δu = shape_value(fv, q_point, i)
            fe[i] += δu * qn * dΓ
        end
    end
    assemble!(f, celldofs(fc), fe)
end
```

Рисунок 30 – Застосування умов Неймана для скалярного поля

Інакше такий фрагмент коду можна включити в процедуру елемента, щоб оцінити граничний інтеграл (рис. 31).

```
for facet in 1:nfacets(cell)
    if (cellid(cell), facet) ∈ getfacetset(grid, "Neumann Boundary")
        reinit!(facetvalues, cell, facet)
        for q_point in 1:getnquadpoints(facetvalues)
            dΓ = getdetJdV(facetvalues, q_point)
            for i in 1:getnbasefunctions(facetvalues)
                δu = shape_value(facetvalues, q_point, i)
                fe[i] += δu * qn * dΓ
            end
        end
    end
end
```

Рисунок 31 – Застосування умов Неймана через гранічний інтеграл

У прикладі вище ми починаємо з проходження циклу по всіх фасетах, а потім перевіряємо, чи цей конкретний фасет розташований на цікавому для нас фасеті під назвою «Neumann Boundary». Якщо ми визначили, що поточний фасет справді знаходиться на межі та в нашому наборі фасетів, ми повторно ініціалізуємо FacetValues для цього фасета за допомогою функції reinit! Під час повторного запуску FacetValues також потрібно вказати номер фасета на додаток до сіткової ячейки. Далі алгоритм просто проходитиме в циклі всі квадратурні точки граней кінчевого елементу, а потім проходитиме в циклі всі тестові функції та складатиме внесок в вектор сили.

У випадку періодичних граничних умов, у Ferrite цей тип граничних умов Діріхле можна додати до ConstraintHandler шляхом створення екземпляра PeriodicDirichlet. Зазвичай це робиться в два кроки. Спочатку ми обчислюємо відображення між періодичними фасетами допомогою функції collect_periodic_facets. Далі, ми створюємо обмеження за допомогою конструктора PeriodicDirichlet. При додаванні обмеження до ConstraintHandler обчислюється результуюче відображення dof.

На прикладі нижче (рис. 32) періодичність застосовується для компонентів 1 і 2 поля :u між дзеркальним набором кордонів «left» і набором кордонів зображення «right».

```
# Create a constraint handler from the dof handler
ch = ConstraintHandler(dofhandler)

# Compute the facet mapping
φ(x) = x - Vec{2}((1.0, 0.0))
face_mapping = collect_periodic_facets(grid, "left", "right", φ)

# Construct the periodic constraint for field :u
pdbc = PeriodicDirichlet(:u, face_mapping, [1, 2])

# Add the constraint to the constraint handler
add!(ch, pdbc)

# If no more constraints should be added we can close
close!(ch)
```

Рисунок 32 – Створення періодичних гарничних умов (лінійна періодіка)

При розв’язуванні задач, що залежать від часу, можуть знадобитися початкові умови, відмінні від нуля.

Для кінцево-елементної інтерпертації рівніння у частинних похідних, тобто $u'(t) = f(u(t), t)$, де $u(t)$ – є ступенями свободи, початкові умови можна вказати за допомогою функції apply_analytical! (див. рис. 33).

```
ρ = 1000; g = 9.81    # density [kg/m³] and gravity [N/kg]
grid = generate_grid(Quadrilateral, (10,10))
dh = DofHandler(grid); add!(dh, :u, 2); add!(dh, :p, 1); close!(dh)
u = zeros(ndofs(dh))
apply_analytical!(u, dh, :p, x -> ρ * g * x[2])
```

Рисунок 33 – Приклад задання початкових умов

3.7 Обмеження (Constraints)

Рівняння у частинних похідних можуть мати ряд обмежень:

$$qI(\underline{a}) = 0,\ I = 1\ to\ n_c, \tag{3.1}$$

де q – нелінійне рівняння обмежень, $\underline{a}$ – вектор ступенів свободи, n_c – число обмежень (constraints).

Є багато способів забезпечити дотримання цих обмежень, наприклад методи штрафів і методи множників Лагранжа [36].

Афінні або лінійні обмеження можна обробляти безпосередньо у Ferrite. Такі обмеження зазвичай можна виразити наступними виразами (3.2):

$$\begin{aligned} a_1 &= 5a_2 + 3a_3 + 1, \\ a_4 &= 2a_3 + 6a_5, \end{aligned} \tag{3.2}$$

де a_1, a_2, … є ступенями свободи системи. У Ferrite ми можемо врахувати таке обмеження за допомогою спеціальної функції ConstraintHandler (рис. 34).

```
ch = ConstraintHandler(dh)
lc1 = AffineConstraint(1, [2 => 5.0, 3 => 3.0], 1)
lc2 = AffineConstraint(4, [3 => 2.0, 5 => 6.0], 0)
add!(ch, lc1)
add!(ch, lc2)
```

Рисунок 34 – Створення обмежень за допомогою функції ConstraintHandler

Афінні обмеження також пливатимуть на шаблон розрідженості матриці жорсткості, тому важливо також включити ConstraintHandler як аргумент під час створення шаблону розрідженості (рис. 35).

```
K = allocate_matrix(dh, ch)
```

Рисунок 35 – Включення ConstraintHandler до шаблону розрідженості

3.8 Рішення лінійних та нелінійних систем у Ferrite.jl

Для вирішення системи виду $Ka = f$ необхідно враховувати афінні обмеження так само, як і граничні умови Діріхле; відбувається перший виклик функції apply! (K, f, ch). Це стисне K і f на місці (тобто нова матриця не буде створена). Однак зауважимо, що ми також повинні визвати функцію apply! на векторі рішення (solution vector) після розв'язання системи для забезпечення афінних обмежень (рис. 36).

```
# ...
# Assemble K and f...

apply!(K, f, ch)
a = K\f
apply!(a, ch) # enforces affine constraints
```

Рисунок 36 – Рішення лінійних систем рівняннь

При рішенні нелінійних систем рівняннь важливо перевірити нев'язку (residuals) після застосування граничних умов з афінними обмеженнями. Функція apply_zero!(K, r, ch) змінює залишкові записи для DOF, які беруть участь у обмеженнях, щоб врахувати сили обмеження. На рисунку 37 показан типовий шаблон для вирішення нелінійної задачі методом Ньютона.

```
a = initial_guess(...)  # Make any initial guess for a here, e.g. `a=zeros(ndofs(dh))`
apply!(a, ch)           # Make the guess fulfill all constraints in `ch`
for iter in 1:maxiter
    doassemble!(K, r, ...)  # Assemble the residual, r, and stiffness, K=∂r/∂a.
    apply_zero!(K, r, ch)   # Modify `K` and `r` to account for the constraints.
    check_convergence(r, ...) && break # Only check convergence after `apply_zero!(K, r, ch)`
    Δa = K \ r              # Calculate the (negative) update
    apply_zero!(Δa, ch)     # Change the constrained values in `Δa` such that `a-Δa`
                            # fulfills constraints if `a` did.
    a .-= Δa
end
```

Рисунок 37 – Розв'язання нелінійних систем рівняннь методом Ньютона

3.9 Розрахункова сітка у Ferrite.jl

Розрахункову сіьтку у Ferrite створити за допомогою функції generate_grid. Більш складні сітки можна імпортувати за допомогою бібліотеки FerriteMeshParser.jl [37] або навіть створити та імпортувати власноруч за допомогою пакетів Gmsh.jl [38] і FerriteGmsh.jl [39] відповідно.

FerriteGmsh.jl підтримує всі визначені типи кінцевих елементів, які були визначені у Ferrite.jl. Сітка створюється на льоту за допомогою gmsh API, або сітка у форматі *.msh чи *.geo може бути прочитана та імпортована за допомогою функції FerriteGmsh.togrid (рис. 38).

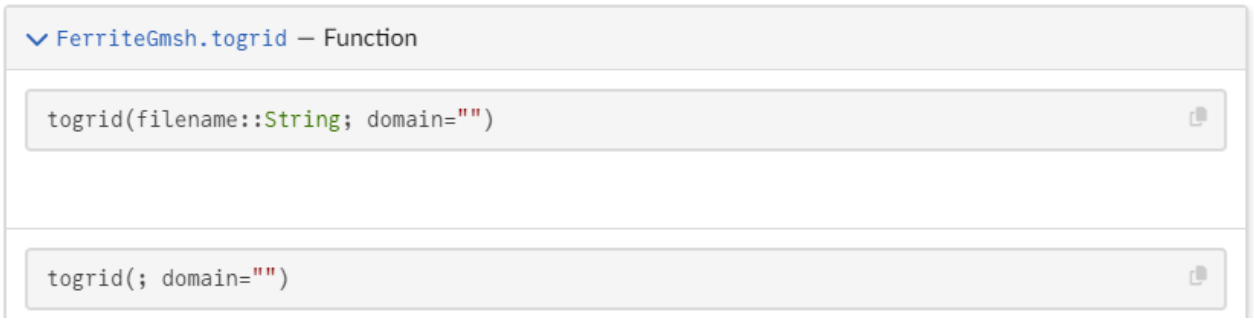


Рисунок 38 – Створення сітки за допомогою функції FerriteGmsh.togrid

Бібліотека FerriteMeshParser.jl перетворює сітку з вхідного файлу Abaqus (.inp) у Ferrite.Grid за допомогою функції get_ferrite_grid. Імпортуються всі стандартні 2d і 3d елементи Abaqus (рис. 39).



Рисунок 39 – Імпорт готової сітки за допомогою бібліотеки FerriteMeshParser.jl

У Ferrite сітка – це сукупність вузлів і ячійок (cells), яка параметризована фізичною розмірністю та типом комірки. Вузли є точками у фізичному просторі та можуть бути ініціалізовані N-кортежем, де N відповідає розмірам, наприклад n1 = Node((0.0, 0.0)).

Комірки сітки визначаються на основі ідентифікаторів вузлів. Отже, вони збирають ідентифікатори в N-кортежі. Розглянемо таку двовимірну сітку (рис. 40).

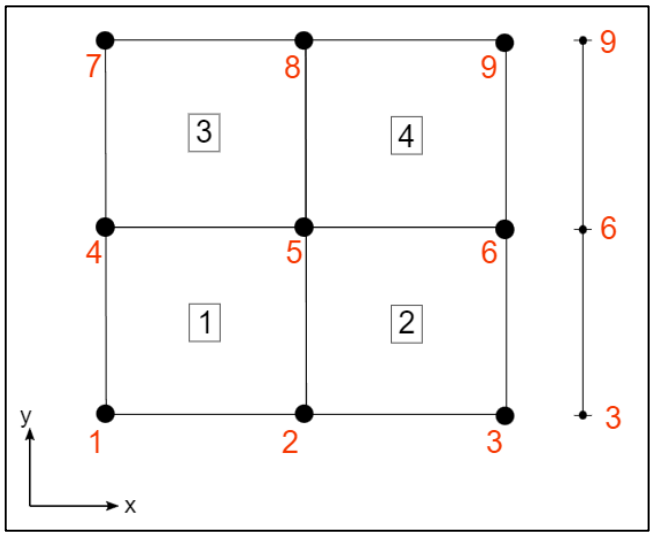


Рисунок 40 – Проста двувимірна сітка у Ferrite

Комірку сітки можна описати наступним чином (див. рис. 41).

```
cells = [Quadrilateral((1, 2, 5, 4)),
         Quadrilateral((2, 3, 6, 5)),
         Quadrilateral((4, 5, 8, 7)),
         Quadrilateral((5, 6, 9, 8))]
```

Рисунок 41 – Створення комірки сітки

Кожен чотирикутник AbstractCell визначається кортежем ідентифікаторів вузлів. Крім того, структура даних Grid містить набори вузлів, клітинок, фасетів і вершин. Кожен із цих наборів визначається через словник Dict{String,

OrderedSet}. Набори вузлів і ячійок представлені в OrderedSet{Int}, що дає набір ідентифікаторів вузла або комірки. Виділені грані, тобто два ребра від ідентифікатора вузла від 3 до 6 і від 6 до 9, на правій стороні нашої тестової сітки тепер можна описати наступним чином (рис. 42).

```
boundary_facets = [(3, 6), (6, 9)]
```

Рисунок 42 – Задання граничних фасетів

Тоді для повної ідентифікації комірки 2 отримаємо (рис. 43).

```
julia> Ferrite.facets(Quadrilateral((2, 3, 6, 5)))
((2, 3), (3, 6), (6, 5), (5, 2))
```

Рисунок 43 – Задання комірки 2

Унікальне представлення ячійок дає відсортована версія цього кортежу. Хоча ми могли б використати цю інформацію для створення набору фасетів, бібліотека Ferrite може створити цей набір шляхом фільтрації на основі координат за допомогою функції addfacetset!, яка додає фасети до сітки з назвою ключа. Набір фасетів відображає ключ String на OrderedSet кортежів, що відповідає (global_cell_id, local_facet_id). Фасети можна використовувати для ініціалізації граничних умов Діріхле для ConstraintHandler.

3.10 Експорт результатів розрахунків

Після того, як задачу було розв’язано і вектор рішення u відомий, ми зазвичай хочемо її візуалізувати. Найпростіший спосіб зробити це – записати

розв’язок у VTK-файл, який можна переглянути, наприклад за допомогою ParaView. Для запису VTK-файлів бібліотека Ferrite постачається з інтерфейсом експорту з серверною частиною WriteVTK.jl [40] для спрощення експорту результатів розрахунків.

Наступну структуру коду можна використовувати для запису різних виводів результатів у vtk-файл (рис. 44).

```
VTKGridFile("my_solution", grid) do vtk
    write_solution(vtk, dh, u)
end;
```

```
VTKGridFile for the closed file "my_solution.vtu".
```

Рисунок 44 – Запис результатів розрахунків в файл VTK

Write_solution() є лише одним із наступних функцій, які можна використовувати:

– write_solution(vtk::VTKGridFile, dh::AbstractDofHandler, u::Vector, suffix="") – зберігає значення у вузлах вектора u до vtk. Кожне поле в dh буде збережено окремо, а суфікс можна використовувати для додавання до імені поля;

– write_node_data(vtk::VTKGridFile, nodedata::Vector{Real}, name) – записує вузлові дані (nodedata), впорядковані вузлами в сітці, у vtk;

– write_projection(vtk::VTKGridFile, proj::L2Projector, vals::Vector, name::AbstractString) – виконує маппинг значень у вузли сітки за допомогою і зберігає результат у vtk;

– Ferrite.write_cellset(vtk, grid::AbstractGrid) – записує усі набори ячійок сітку з назвами відповідно до їхніх ключів 1, якщо клітинка є в наборі, і 0 в іншому випадку;

– Ferrite.write_nodeset(vtk::VTKGridFile, grid::AbstractGrid, nodeset::String) – записує вузлові значення результатів;

– Ferrite.write_constraints(vtk::VTKGridFile, ch::ConstraintHandler) – зберігає граничні умови Діріхле у файл vtk.

Для моделювання з кількома кроками (multistep analysis) зазвичай записується один файл VTK (.vtu) для кожного кроку. Щоб зв'язати фактичний крок з кожним із цих файлів, можна використовувати фукнцію paraview_collection() із WriteVTK.jl. Це створить один файл даних paraview (*.pvd) і один VTKGridFile (*.vtu) для кожного кроку (рис. 45).

```
using WriteVTK
pvd = paraview_collection("my_results")
for (step, t) in enumerate(range(0, 1, 5))
    # Do calculations to update u
    VTKGridFile("my_results_$step", dh) do vtk
        write_solution(vtk, dh, u)
        pvd[t] = vtk
    end
end
vtk_save(pvd);
```

```
6-element Vector{String}:
 "my_results.pvd"
 "my_results_1.vtu"
 "my_results_2.vtu"
 "my_results_3.vtu"
 "my_results_4.vtu"
 "my_results_5.vtu"
```

Рисунок 45 – Збереження результатів для multistep analysis

3.11 Реалізація функціоналу програмних кодів для виконання інженерних розрахунків методом кінцевих елементів засобами Julia

На основі бібліотеки Ferrite.jl було реалізовано декілька задач для рішення методом кінцевих елементів у Julia.

Задача 1. Лінійно-пружне деформування.

Це класична задача метода кінцевих елементів механіки твердого тіла з находження невідомого вектора переміщень u(x). Схема навантаження та сама модель показана на рисунку 46.

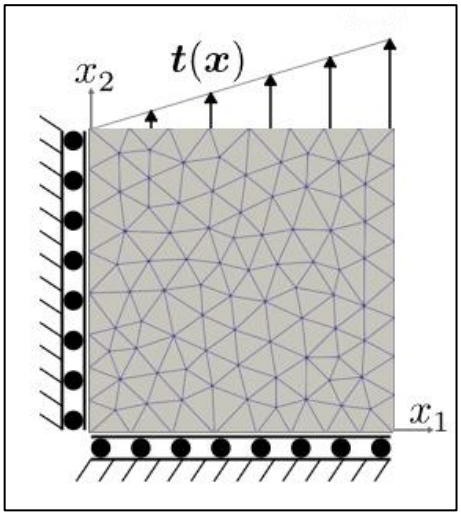


Рисунок 46 – Схема навантаження та розрахункова модель.

Для цієї задачи баланс моментів для квазі-статичного навантаження відповідає наступним співвідношенням (3.3):

$$\begin{aligned} &div(\sigma) + b = 0 \quad && x \in \Omega, \\ &u = u_D && x \in \Gamma_D, \\ &n \cdot \sigma = t_N && x \in \Gamma_N, \end{aligned} \tag{3.3}$$

де σ – тензор напружень Коші [41], b – вектор навантаження (body forces), розрахунковий домен Ω містить граничні умови Діріхле Γ_D та Нейманна Γ_N вектор напрямку n для котрих направлений назовні розрахункової області, u_D являє собою задані переміщення на границі Γ_D, а t_N – відомі усилля на границі Γ_N. Для нашої задаячи ми будемо обчислювати наступну лінійно-пружну форму деформування:

$$\sigma = \mathrm{C} : \varepsilon, \varepsilon = [grad(u)]^{sym}. \tag{3.4}$$

Як можна побачити в (3.4), C є тензором пружної жорсткості 4-го порядку, а ε – тензор малих деформацій.

Реалізація на Julia. Безпосередньо розробка коду відбувалась в IDE Visual Studio Code із завантаженими доповнень для мови Julia. Код цієї задачи (файл linear_elasticity.jl) з коментарями викладено на GitHub [43]. Розмірність розрахункової сітки дорівнює 178 елементам та 106 вузлам.

Результати обчислень. Після закінчення процедури обчислень, ми отримаємо результати у вигляді файлу linear_elasticity.vtu який можна подивитися у застосунку з відкритим кодом Paraview (див. рис. 47 – 52).

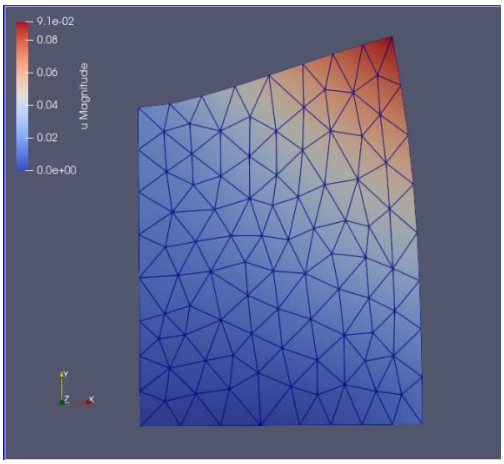


Рисунок 47 – Сумарні переміщення Usum, mm

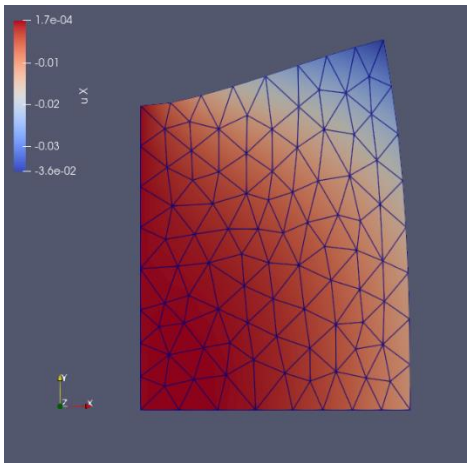


Рисунок 48 – Горизонтальні переміщення Ux, mm

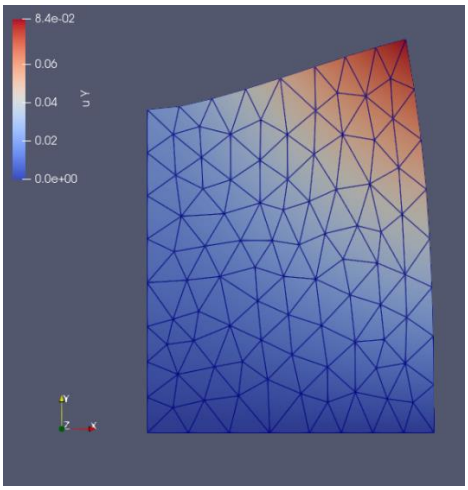


Рисунок 49 – Вертикальні переміщення Uy, mm

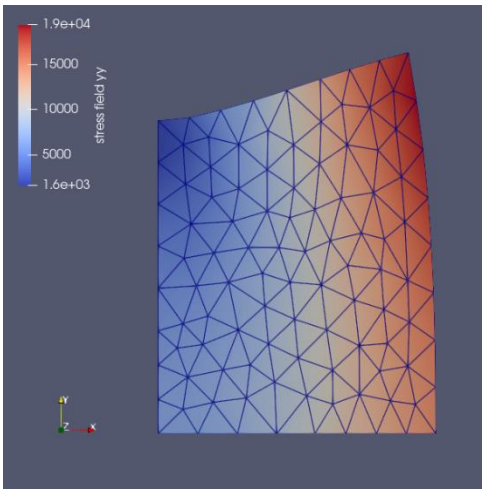


Рисунок 50 – Вертикальні нормальні напруження σ_{yy}, МПа

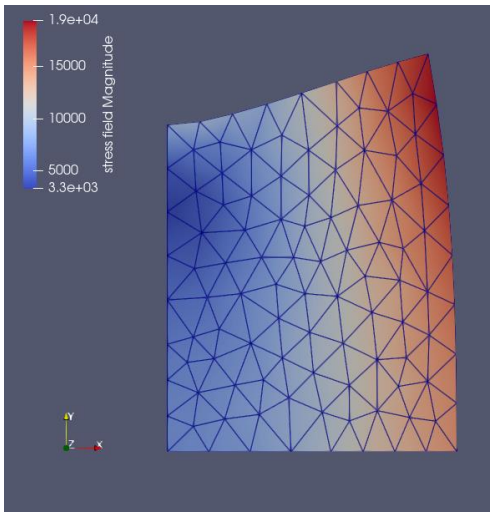


Рисунок 51 – Єквивалентні (сумарні) напруження σ_{eqv}, МПа

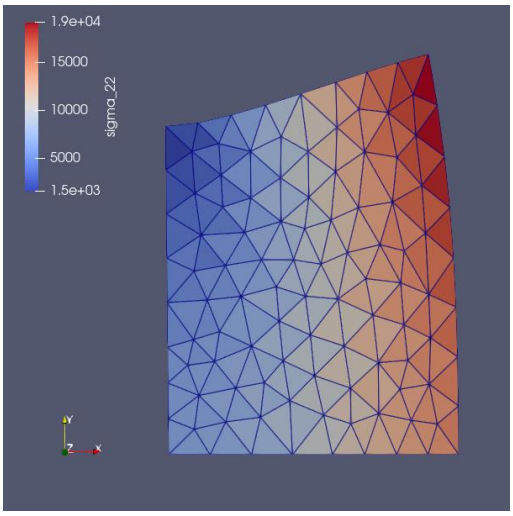


Рисунок 52 – Поле напружень в межах кожного елемента, МПа

Порівняння результатів обчислень в Julia з рішеннім в Ansys. Створимо ідентичну розрахункову модель в програмному комплексі Ansys з аналогічними умовами заріплення та навантаження (див. рис. 46). Далі, порівняємо результати розрахунків в Julia та Ansys при збільшенні розмірності моделі. Розрахункова модель Ansys показана на рисунку 53.

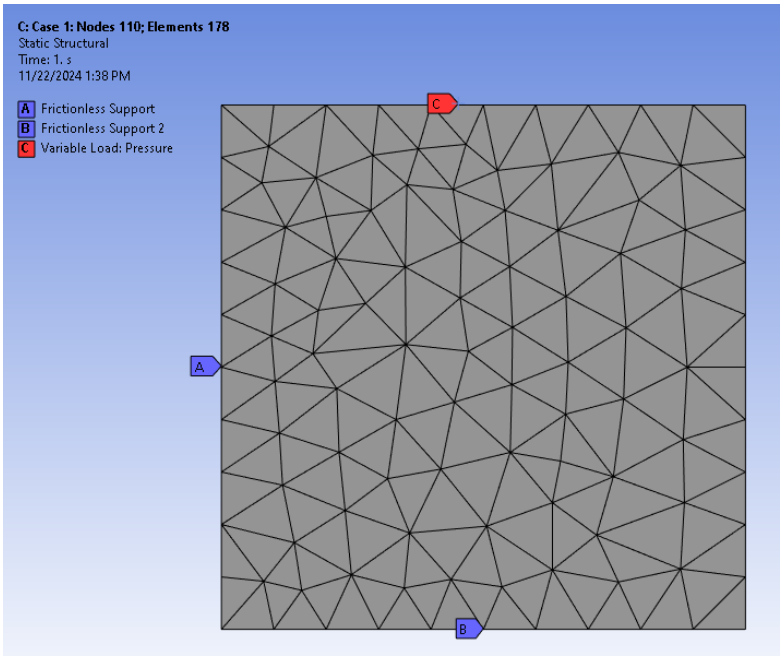


Рисунок 53 – Розрахункова модель Ansys

На рисунках 54 – 57 показані порівняні результати розрахунків у Julia та Ansys.

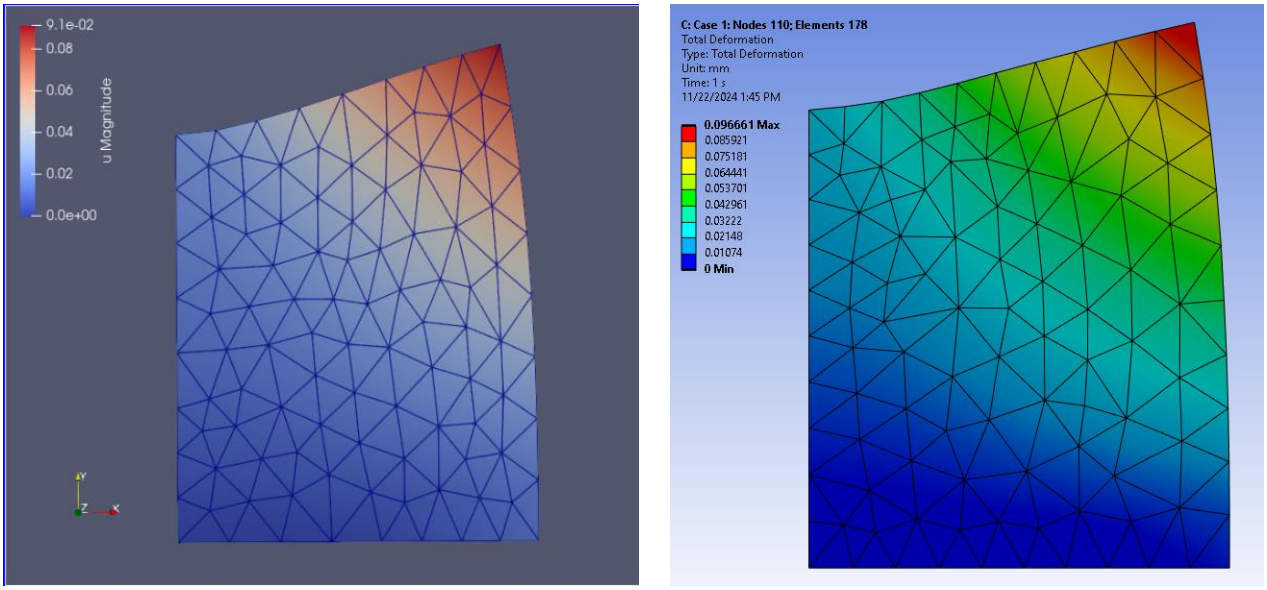


Рисунок 54 – Сумарні переміщення Usum
в Julia (ліворуч) та Ansys (праворуч), mm

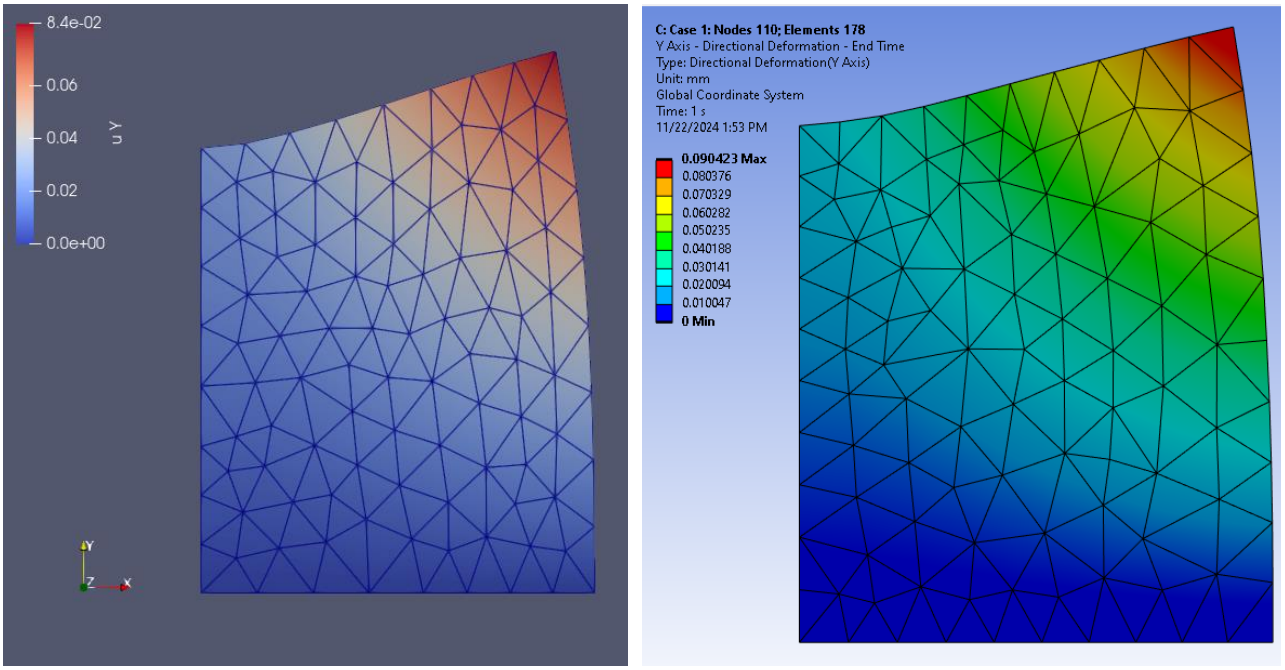


Рисунок 55 – Вертикальні переміщення Uy
в Julia (ліворуч) та Ansys (праворуч), mm

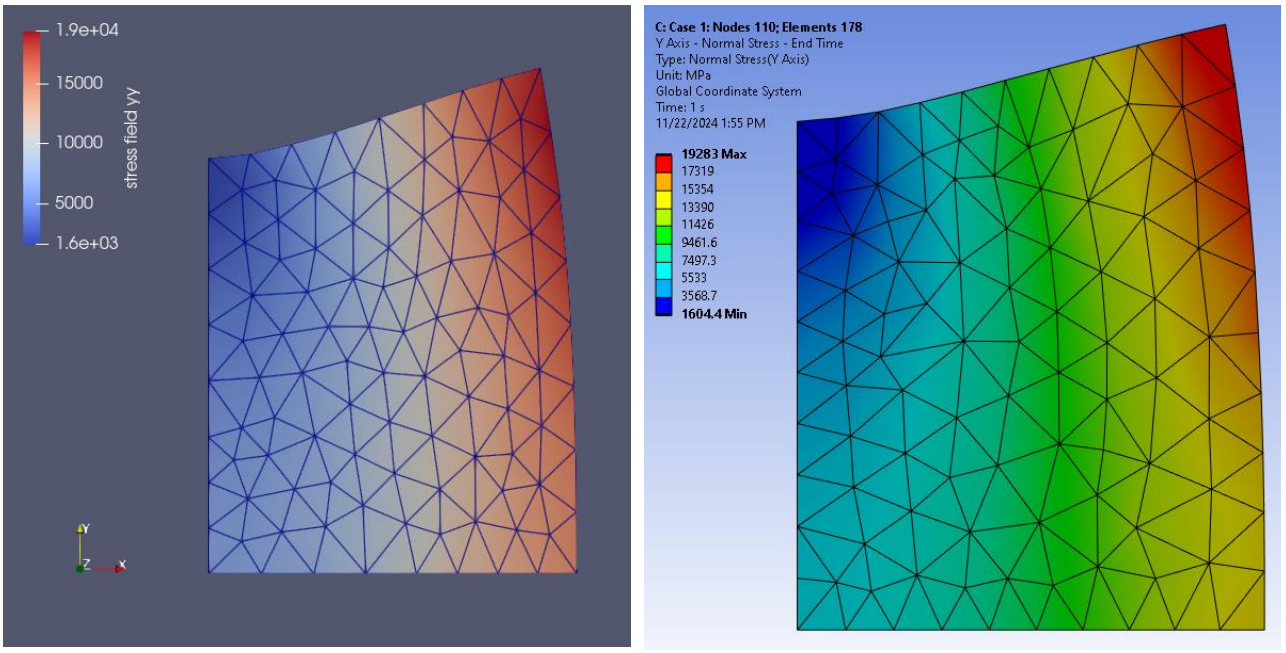


Рисунок 56 – Вертикальні нормальні напруження σ_{yy}
в Julia (ліворуч) та Ansys (праворуч), МПа

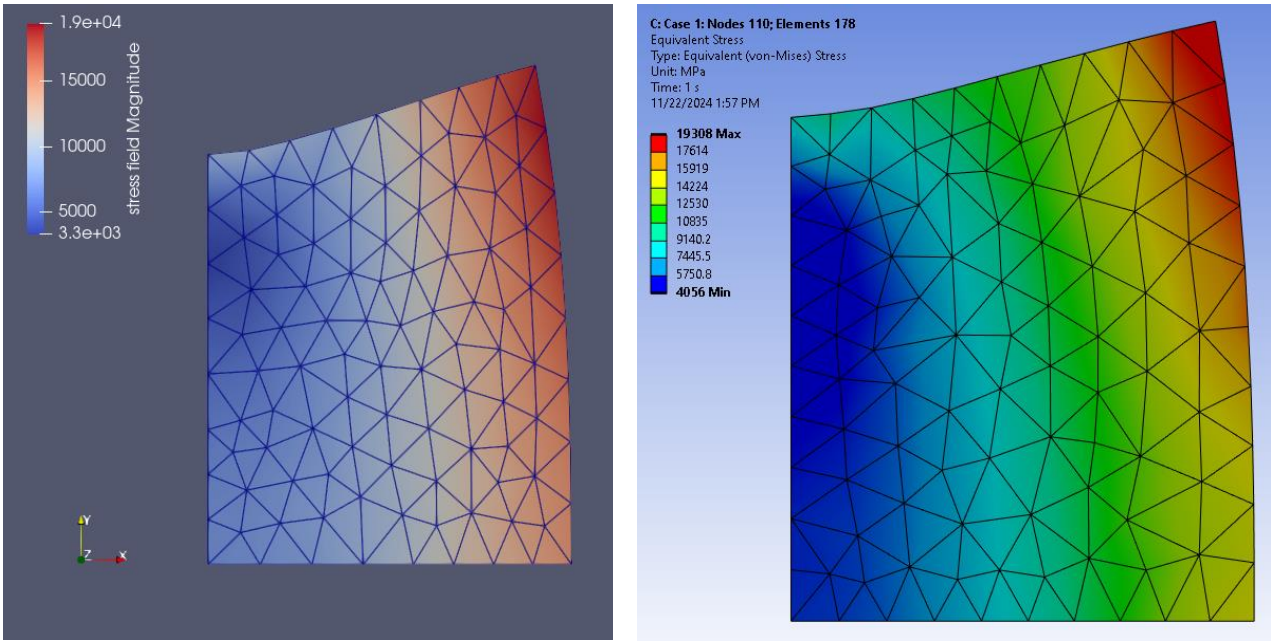


Рисунок 57 – Єквивалентні (сумарні) напруження σ_{eqv}
в Julia (ліворуч) та Ansys (праворуч), МПа

В таблиці 2 наведені результати розрахунків на сітках різних розмірностей в порівнянні з відповідним рішенням у Julia.

Таблиця 2 – Результати розрахунків в Ansys на сітках різних розмірностей в порівнянні з рішенням у Julia

N	Розмірність сітки, вузли	Результати моделювання							
		Julia				Ansys			
		Usum, mm	Uy, mm	σ_{yy}, МПа	σ_{eqv}, МПа	Usum, mm	Uy, mm	σ_{yy}, МПа	σ_{eqv}, МПа
0	**110**	**0.0910**	**0,0840**	**19000**	**19000**	**0,0966**	**0,0904**	**19283**	**19308**
1	500	0.0975	0.0910	19250	19280	0.0979	0.0916	19777	19821
2	1000	0.0978	0.0915	19445	19550	0.0981	0.0918	19869	19896
3	2500	0,0982	0,0916	19650	19680	0,0983	0,0919	19875	19879
4	5000	0,0982	0,0917	19755	19780	0,0983	0,0919	19983	19989
5	10000	0,0982	0,0917	19855	19912	0,0983	0,0919	20037	20049
6	15000	0,0982	0,0918	19988	20010	0,0983	0,0919	20001	20002
7	20000	0,0982	0,0918	20015	20025	0,0983	0,0919	20030	20033

На рисунках 58 – 61 наведено графіки досліджень на сіткову незалежність.

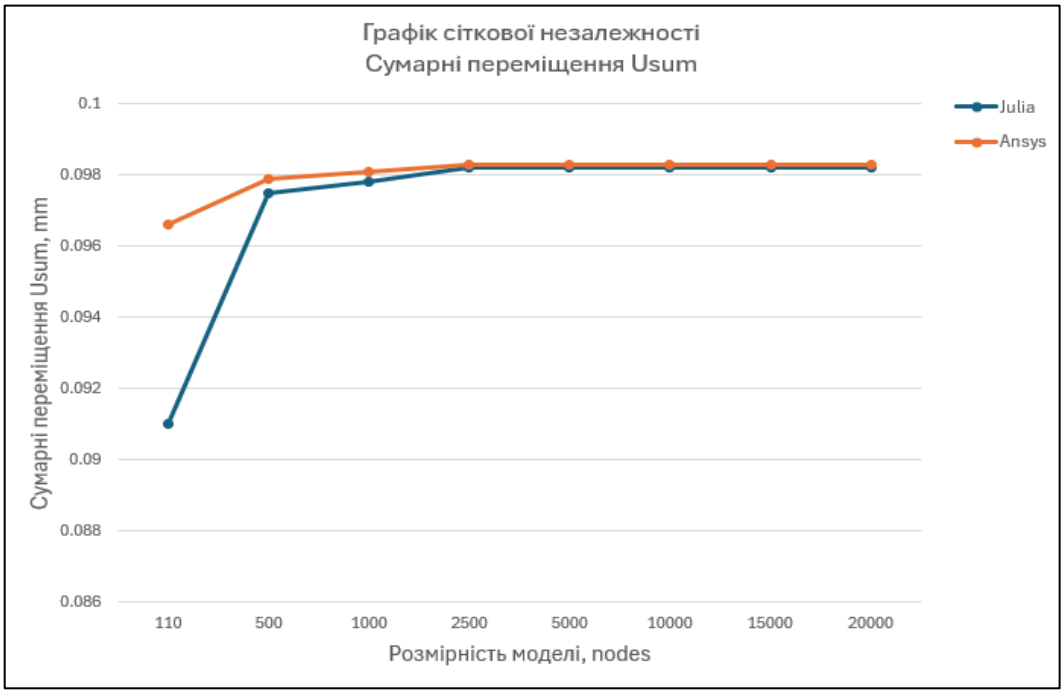


Рисунок 58 – Графік порівняння сумарних переміщень Usum

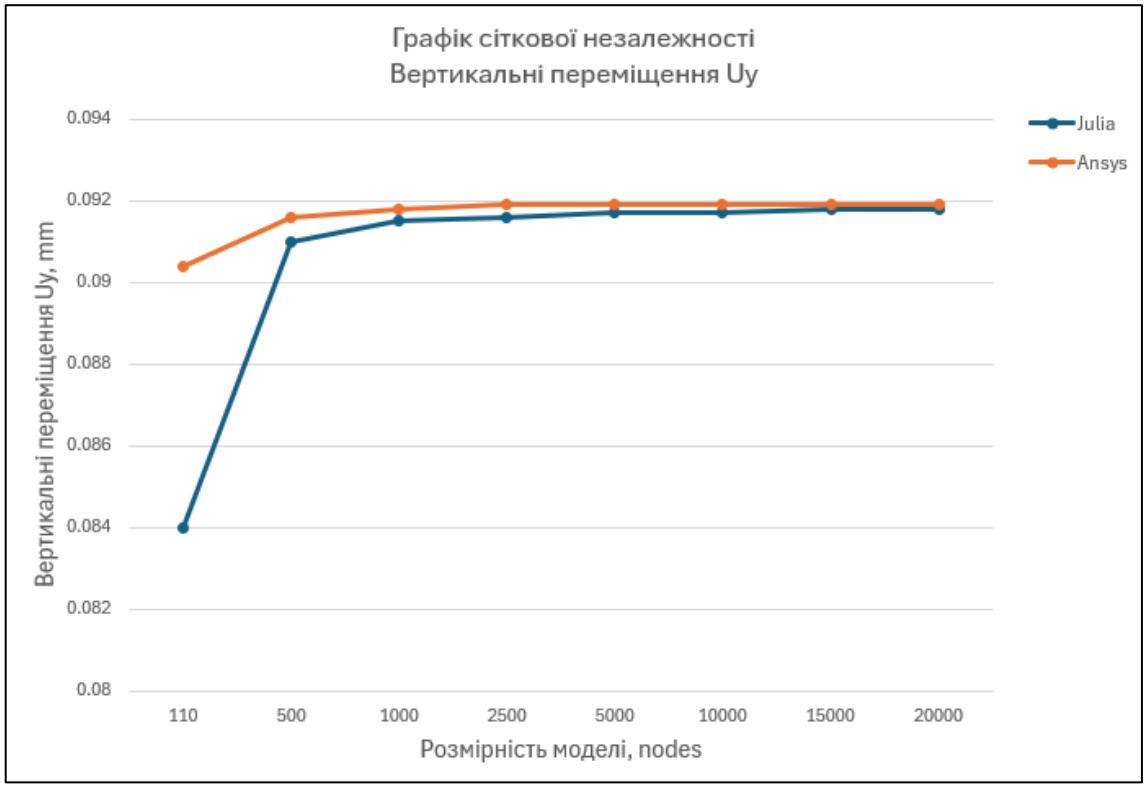


Рисунок 59 – Графік порівняння вертикальних перміщень Uy

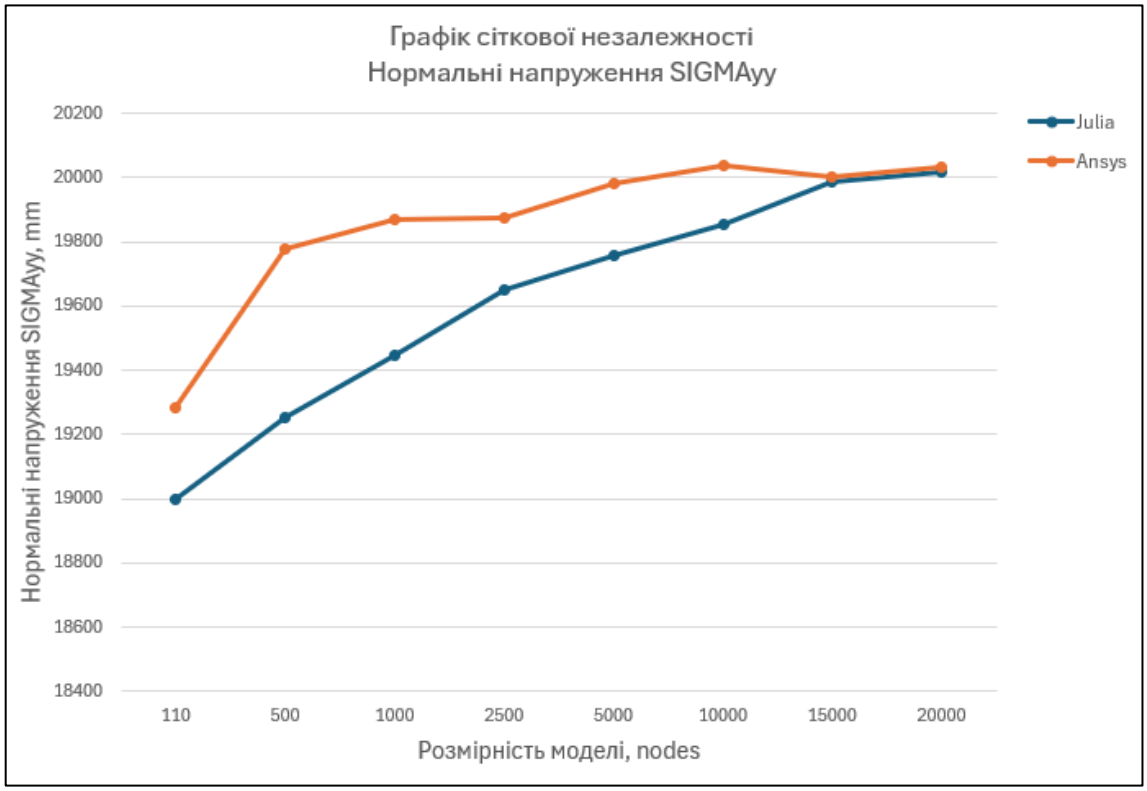


Рисунок 60 – Графік порівняння нормальних напружень σ_{yy}

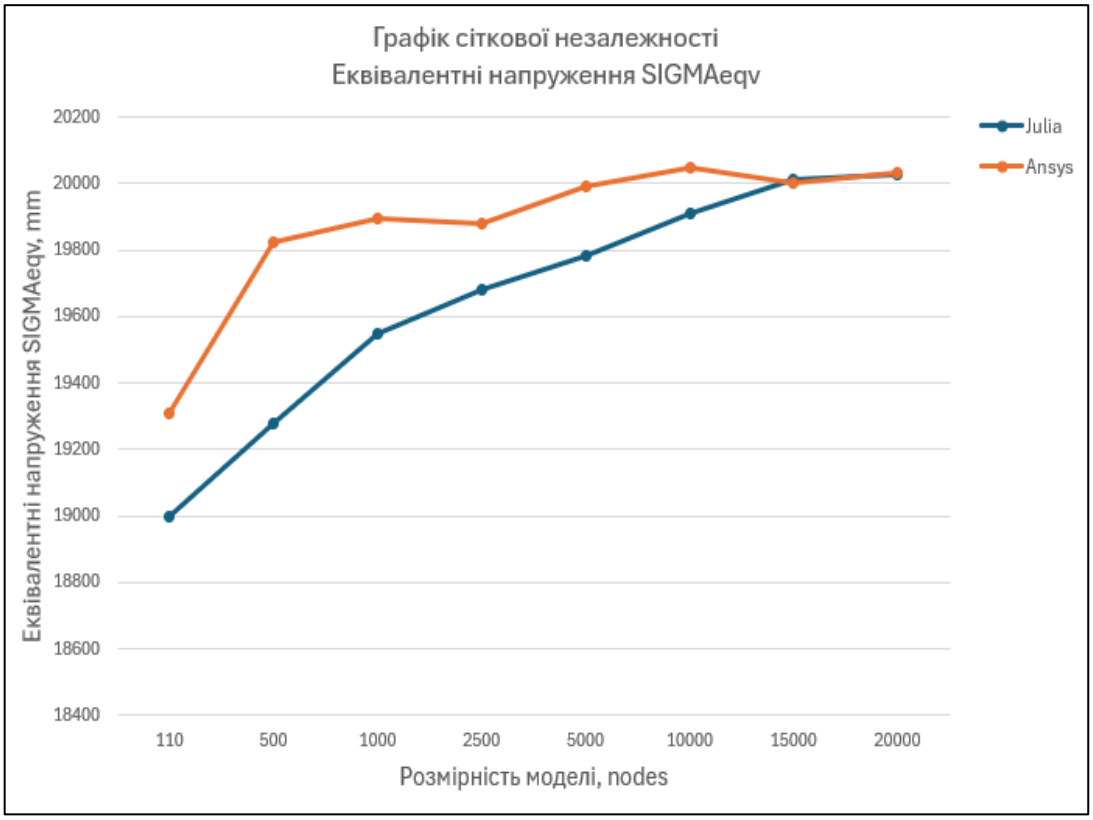


Рисунок 61 – Графік порівняння еквівалентних напружень σ_{eqv}

Як ми можемо бачити з Таблиці 5 та рисунків 54 – 61, результати, які були отримані в Julia певним чином узгоджуються з відповідними результатами моделювання в Ansys, при цьому точність розрахунків зростає по мірі збільшення розмірності задачі та рішення стає майже постійним та незалежним від розмірності розрахункової моделі. Це може свідчити про те, що математика біліотеки Ferrite може бути використана для робудови більш складних розрахункових алгоритмів метода кінцевих елементів і точність отриманих результатів буде достатньою для подальшого інженерного аналізу.

Задача 2. Пластичне деформування по Мізесу.

Задача описує нелінійну модель матеріала та обчислює напруження Мізеса при навантаженні консольної балки. Більш детально теорія пластичності Мізеса наведена в [42] і тут не наводиться через великий обсяг теоретичного матеріалу. Але базову математику та принципи обчислення крітерія пластичності по Мізесу

можна описати наступним чином. У механіці безперервної середовища критерій максимальної енергії формозмінення (також відомий, як критерій текучості фон Мізеса) стверджує, що текучість пластичного матеріалу починається, коли другий інваріант девіаторної напружень J_2 досягає критичних значень. Математично крітерій текучости Мізеса відповідає співвідношенню (3.5):

$$J_2 = k^2, \tag{3.5}$$

де k – це напруження текучості матеріала (yield stress) при чистому зсуві (pure shear). Як буде показано далі, на початку текучості матеріала, величина межі текучості при чистому зсуві в $\sqrt{3}$ рази нижча, ніж межа текучості при розтягуванні у випадку простого розтягу. Таким чисном отримаємо співвідношення (3.6):

$$k = \frac{\sigma_y}{\sqrt{3}}, \tag{3.6}$$

де σ_y – межа текучості матеріалу при розтягуванні (tensile yield strength).

Якщо напруженння Мізеса стануть рівними межі текучості матеріалу, то критерій текучості фон Мізеса буде характерізуватися наступним співвідношенням:

$$\sigma_v = \sigma_y = \sqrt{3J_2}. \tag{3.7}$$

Підставляючи вираз для J_2 у тензор напружень Коші, отримаємо основне співвідношення для крітерія текучості Мізеса:

$$\sigma_v^2 = \frac{1}{2}[(\sigma_{11} - \sigma_{22})^2 + (\sigma_{22} - \sigma_{33})^2 + (\sigma_{33} - \sigma_{11})^2 + 6(\sigma_{23}^2 + \sigma_{31}^2 + \sigma_{12}^2)]. \tag{3.8}$$

Це рівняння визначає поверхню текучості як циліндричну поверхню (див. рис. 62). Як видно з рисунку нижче, умова текучості не залежить від гідростатичних напружень.

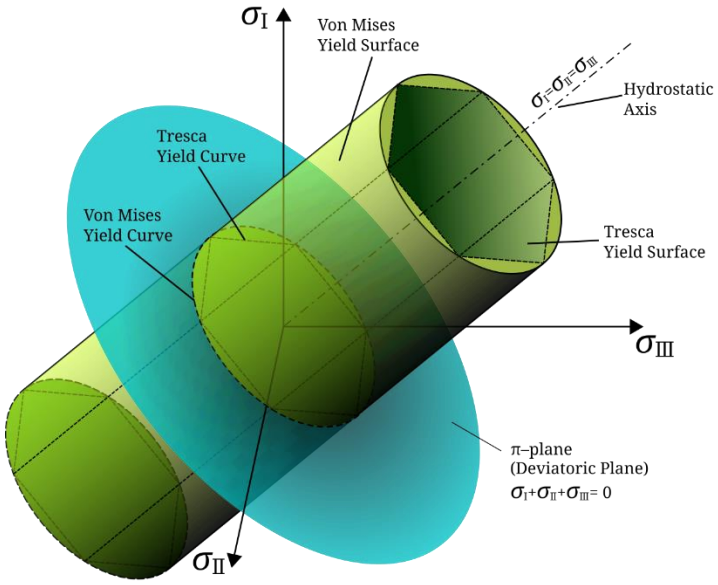


Рисунок 62 – Поверхня текучості за Мізесом

Реалізація на Julia. Тепер спробуємо реалізувати теоретичні відомості про критерій текучості Мізеса на практиці за допомогою бібліотеки Ferrite.jl. Для цього напишемо відповідний програмний код, який буде обчислювати пластичну поведінку консольно закріпленої балки, яка з другого кінця навантажена зусиллям F (рис. 63).

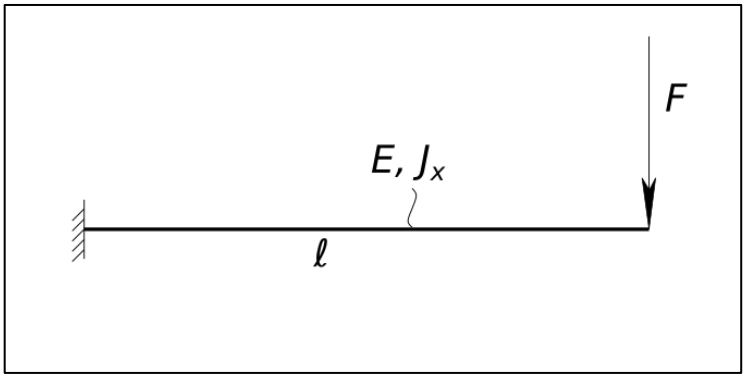


Рисунок 63 – Розрахункова схема

Безпосередньо розробка коду відбувалась в IDE Visual Studio Code із завантаженими доповнень для мови Julia. Код цієї задачи (файл Mises_plasticity.jl) з коментарями викладено на GitHub [43].

Результати обчислень. Після завершення розрахункового циклу, програма видає звіт по ітераціям (див. рис. 64) вирішувача та файл результатів *.vtu для подальшої візуалізації у Paraview (провідний у світі механізм візуалізації постобробки з відкритим кодом. Він інтегрується з наявними інструментами та робочими процесами, дозволяючи створювати візуалізації для швидкого аналізу даних).

```
 Starting Netwon-Raphson Iterations:

 Time Step @time = 1:
Iteration: 0    residual: 1435838.41167605
Iteration: 1    residual: 118655.22429549
Iteration: 2    residual: 59.50456058
Iteration: 3    residual: 0.00002640

 Time Step @time = 2:
Iteration: 0    residual: 159537.60129737
Iteration: 1    residual: 1694313.86970367
Iteration: 2    residual: 61777.44063541
Iteration: 3    residual: 14.34471354
Iteration: 4    residual: 0.00001300

 Time Step @time = 3:
Iteration: 0    residual: 159537.60129729
Iteration: 1    residual: 2983079.21464719
Iteration: 2    residual: 184748.94467316
Iteration: 3    residual: 194.33615860
Iteration: 4    residual: 0.00024519

 Time Step @time = 4:
Iteration: 0    residual: 159537.60129756
Iteration: 1    residual: 3791724.91970473
Iteration: 2    residual: 103717.95331262
Iteration: 3    residual: 47.03142228
Iteration: 4    residual: 0.00002289

 Time Step @time = 5:
Iteration: 0    residual: 159537.60129661
Iteration: 1    residual: 5186027.57387942
Iteration: 2    residual: 858572.34134710
Iteration: 3    residual: 3227.93307728
Iteration: 4    residual: 0.06673495

 Time Step @time = 6:
Iteration: 0    residual: 159537.60129740
Iteration: 1    residual: 6165102.90240116
Iteration: 2    residual: 1424967.82954959
Iteration: 3    residual: 13237.36029615
Iteration: 4    residual: 1.42544587
Iteration: 5    residual: 0.00001981
```

```
 Time Step @time = 7:
Iteration: 0    residual: 159537.60129760
Iteration: 1    residual: 7104345.12277559
Iteration: 2    residual: 2244851.12009734
Iteration: 3    residual: 30125.16204767
Iteration: 4    residual: 6.53428720
Iteration: 5    residual: 0.00002458

 Time Step @time = 8:
Iteration: 0    residual: 159537.60129742
Iteration: 1    residual: 8811547.99068300
Iteration: 2    residual: 2125693.06329002
Iteration: 3    residual: 19051.80404251
Iteration: 4    residual: 2.99707053
Iteration: 5    residual: 0.00003397

 Time Step @time = 9:
Iteration: 0    residual: 159537.60129841
Iteration: 1    residual: 9387830.47201345
Iteration: 2    residual: 2082416.95822988
Iteration: 3    residual: 14918.19061243
Iteration: 4    residual: 1.45155031
Iteration: 5    residual: 0.00003574

 Time Step @time = 10:
Iteration: 0    residual: 159537.60129800
Iteration: 1    residual: 9545475.05532265
Iteration: 2    residual: 1946989.46728202
Iteration: 3    residual: 34428.26867748
Iteration: 4    residual: 4.33797283
Iteration: 5    residual: 0.00004122

julia>
```

Рисунок 64 – Ітерації Ньютона-Рафсона при обчисленні критерію пластичності за Мізесом

На рисунках 65 – 68 наведено результати обчисленнь у вигляді розподілу відповідного параметра. Треба зауважити, що метод ітерування Ньютона-Рафсона, це простий та ефективний алгоритм для наближеного знаходження коренів дійснозначних функцій, тобто розв’язання рівнянь виду: f (x) = 0. Метод Ньютона дуже важливий у обчислювальної математики: здебільшого саме він використовується для знаходження чисельних розв’язків систем лыныйних алгебраїчних рівнянь.

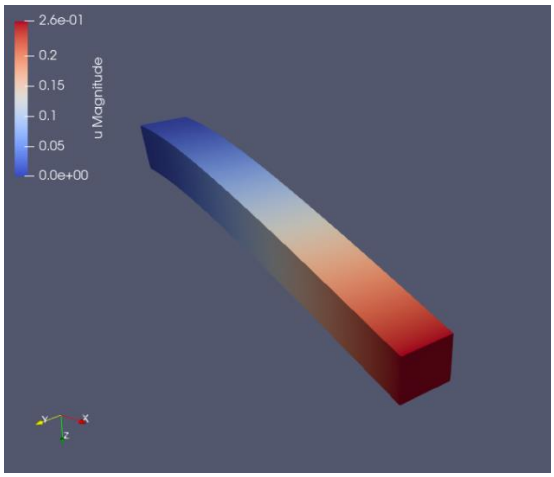


Рисунок 65 – Розподіл переміщень по балці U, m

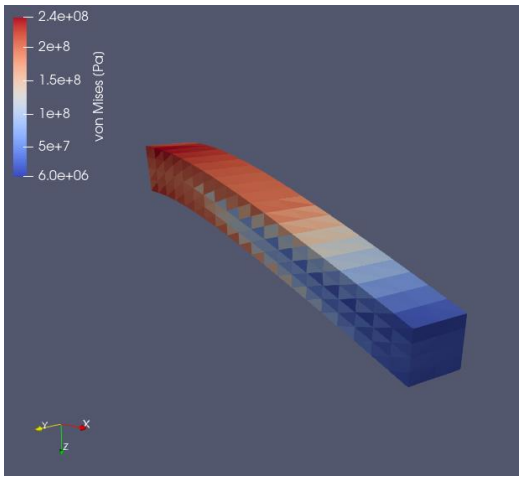


Рисунок 66 – Розподіл напруженнь за Мізесом, Па

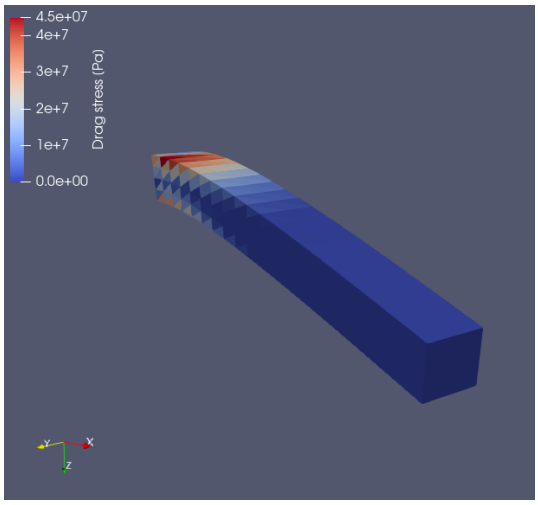


Рисунок 67 – Розподіл напруженнь зсуву, Па

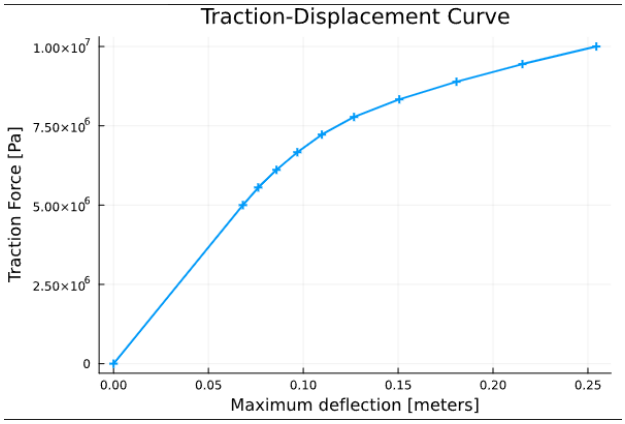


Рисунок 68 – Розрахункова крива «навантаження-пеерміщення», m

Як можна побачити з рисунку 68, маємо пластичну поведінку матеріала відповідно до критерію пластичності Мізеса, який було обчислено вище.

Порівняння результатів обчислень в Julia з рішеннім в Ansys. Порівняємо результати розрахунків з Julia на однаковій розрахунковій сітці, а потім будемо поступово збільшувати розмірність моделі в Ansys та Julia для отримання більш точних результатів. Розрахункова модель Ansys показана на рисунку 69.

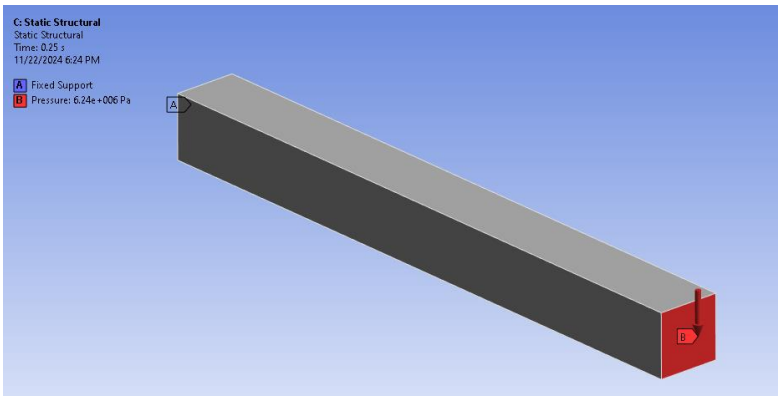


Рисунок 69 – Розрахункова модель балки в Ansys

На рисунках 70 та 71 наведено порівняні результати розрахунків у Julia та Ansys.

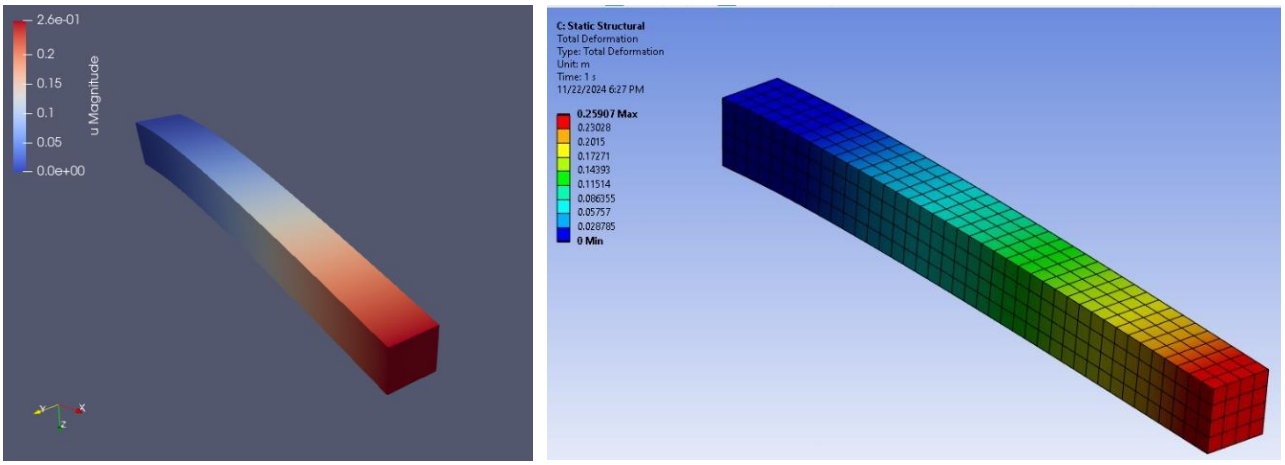


Рисунок 70 – Розподіл переміщень по балці U
в Julia (ліворуч) та Ansys (праворуч), m

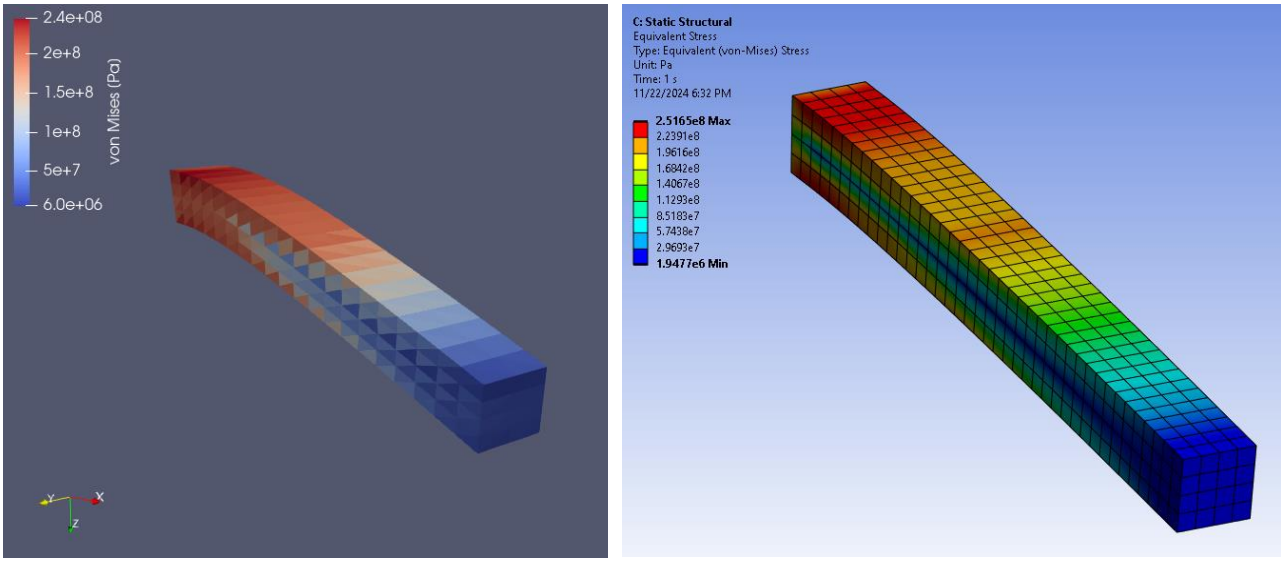


Рисунок 71 – Розподіл напруженнь за Мізесом σ_{eqv}
в Julia (ліворуч) та Ansys (праворуч), Па

Як можна бачити з результатів порівняння двох чисельних розрахунків (розмірність моделей в Julia та ANSYS приблизно однакова та становить близько 1000 вузлів та 600 елементів), максимальні переміщення в Julia становлять 0,26м, в Ansys – 0.25907м, а еквівалентні напруження за Мізесом – в Julia 2.4E+08 Па, в Ansys – 2.5165E+08 Па, тобто маємо дуже близькі результати. Це свідчить про достатню точність розрахункових алгоритмів бібліотеки Ferrite.jl та загалом мови програмування Julia.

В таблиці 3 наведено результати розрахунків на сітках різних розмірностей в порівнянні з відповідним рішенням у Julia.

Як ми можемо бачити з таблиці 3 та рисунків 72 і 73, результати, які були отримані в Julia певним чином узгоджуються з відповідними результатами моделювання в Ansys, при цьому точність розрахунків зростає по мірі збільшення розмірності задачі та стає майже незмінною та незалежною від розмірності розрахункової моделі. Це може свідчити про те, що математика біліотеки Ferrite дає достатню точність для проведення інженерних розрахунків.

Таблиця 3 – Результати розрахунків консольної балки в Ansys на сітках різних розмірностей в порівнянні з рішенням у Julia

N	Розмірність сітки, вузли	Результати моделювання			
		Julia		Ansys	
		Umax, mm	σ_{eqv}, МПа	Umax, mm	σ_{eqv}, МПа
0	**1000**	**0.26000**	**2.4000E+08**	**0.25907**	**2.5165E+08**
1	2000	0.25298	2.4970E+08	0.25401	2.5170E+08
2	2500	0.25325	2.5334E+08	0.25787	2.5532E+08
3	5000	0.25456	2.5412E+08	0.25656	2.5401E+08
4	10000	0.25722	2.5598E+08	0.25731	2.5685E+08
5	15000	0.25789	2.6089E+08	0.25786	2.6185E+08
6	20000	0.25763	2.6425E+08	0.25760	2.6405E+08
7	25000	0.25799	2.6589E+08	0.25810	2.6679E+08
8	50000	0.25815	2.7565E+08	0.25825	2.7599E+08
9	75000	0.25825	2.8158E+08	0.25836	2.8049E+08
10	100000	0.25832	2.8415E+08	0.25848	2.8494E+08

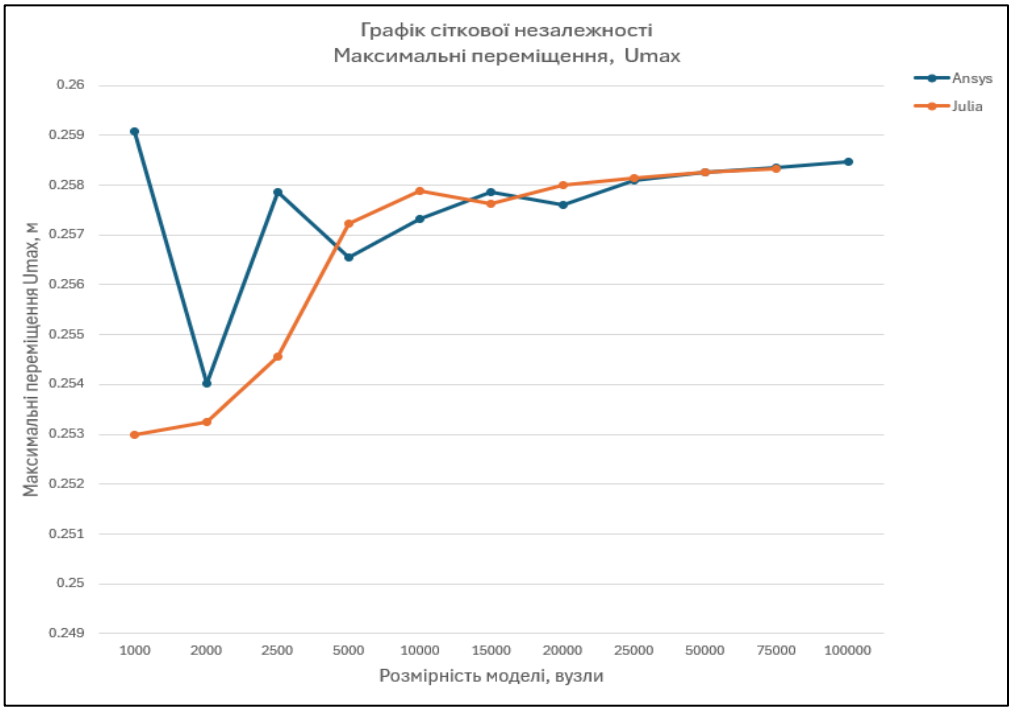


Рисунок 72 – Графік порівняння максимальних переміщень Umax

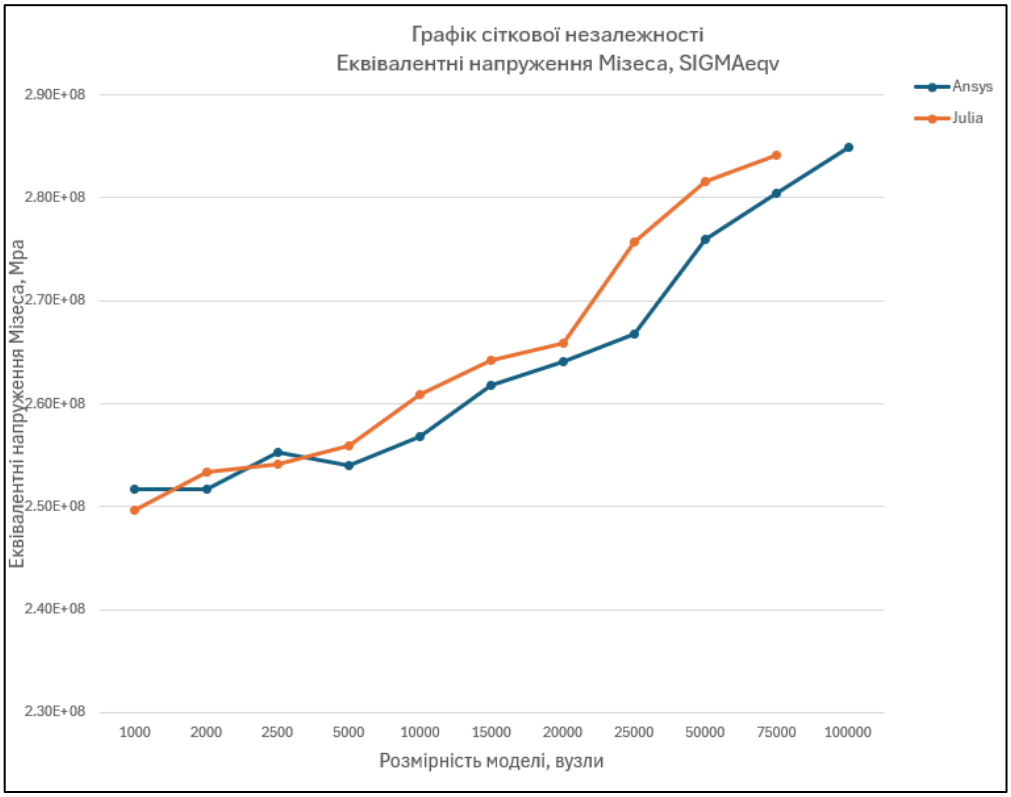


Рисунок 73 – Графік порівняння еквівалентних напружень σ_{eqv}

3.12 Висновки з розділу 3

Бібліотека кінцево-елементного аналізу Ferrite.jl маєу своєму складі багатий функціонал для розробки та виконання різноманітних інженерних задач, як з області прикладної механіки, так із низки інших фізичних дисциплін (лінійний та нелінійний теплообмін, обчислювальна гидродинаміка тощо).

За допомогою Ferrite.jl було розроблено декілька програмних кодів для обчислення різноманітних фізичних задач механіки деформування твердого тіла та зроблено порівняння з аналогічними розрахунками в Ansys.

Реалізована класична задача лінійно-пружного деформування моделі за допомогою додаткових бібліотек Tensors.jl, FerriteGmsh.jl, SparseArrays.jl.

Реалізована модель гіперпружної поведінки матеріалу Нео-Хукина (Neo-

Hookian) за допомогою додаткового пакета IterativeSolvers.jl та ітеративного методу Ньютона-Рафсона (Newton-Raphson).

Реалізована та обчислена модель пластичної поведінки матеріала на базі крітерія пластичності Мізеса (von-Mises Yeild Criterion), також сумісно з ітеративним алгоритмом Ньютона-Рафсона.

Бібліотека Ferrite – це досіть потужний набір інструментів кінцевих елементів на основі якого можлива реалізація більш складних проєктів реалізації методів кінцевих елементів для будь-якої фізічної проблем.

Візуалізація результатів розрахунків может бути виконана як власними інструментарієм Julia (наприклад, бібліотека Plots.jl), так й за допомогою більш продвинутих засобів візуалізації результатів обрахунків кінцево-елементних моделей SALOME та ParaView.

ВИСНОВКИ

Досліджена проблема використання сучасних інструментів програмування для розробки та виконання інженерних обчислень, оскільки наукові та інженерні розрахунки вимагають виконання складних математичних операцій, обробки великих обсягів даних та моделювання комплексних фізіко-математичних систем.

Досліджені засоби для рішення поставленої проблеми: існуючі мови програмування мають свої сильні та слабкі сторони. Наприклад, Python, який популярний завдяки своїй простоті та широкому набору бібліотек для наукових обчислень, таких як NumPy та SciPy, має вузьке місце завдяки його інтерпретованості. З іншого боку, мови, такі як C/C++ та Fortran, забезпечують високу продуктивність, але потребують більшого часу на розробку та тестування.

Проведено дослідження мови програмування Julia на предмет її використання для розробки алгоритмів та програмних реалізацій методів кінцевих елементів широкого кола інженерних фізичних задач.

Розроблені готові до застосування програмні коди, які реалізують метод кінцевих елементів у розрахунках лінійно-пружніх задач деформування твердих тіл, пластичну поведінку матеріала, моделі гіперпружності, крітерії пластичности Мізеса та нестаціонарні ітеративні розрахунки.

Завдяки багатому вибору математичних та спеціалізованих бібліотек, мова програмування Julia є найбільш оптимальним засобом для проведення великого об’єму інженерних обчислень із максимально можливою продуктивністю та точністю, що стане у нагоді в кожного інженера-винахідника та спеціалістів з програмування фізичних задач, у тому числі, штучного інтелекту для передбачень можливих режимів роботи механічних систем.

ПЕРЕЛІК ПОСИЛАНЬ

1. Ayer V. M., Miguez S., Toby B. H. Why scientists should learn to program in Python. *Powder Diffr*. 2014. Vo1. 29, no. s2. P. S48–S64. URL: https://doi.org/10.1017/S0885715614000931 (дата звернення: 10.05.2024).

2. Lubin M., Dunning I. Computing in operations research using Julia. *INFORMS J. Comput.* 2015. Vo1. 27, no. 2. P. 193–430. URL: https://doi.org/10.1287/ijoc.2014.0623 (дата звернення: 10.05.2024).

3. Bezanson J., Karpinsky S., Shah V. B. et a1. Julia: a fast dynamic language for technical computing. *Arxiv.org*. URL: https://arxiv.org/abs/1209.5145 (дата звернення: 10.05.2024).

4. Bezanson J., Edelman A., Karpinsky S. et a1. Julia: a fresh approach to numerical computing. *SIAM Rev*. 2017. Vol. 59, no. 1. P. 65–98. URL: https://doi.org/10.1137/141000671 (дата звернення: 10.05.2024).

5. Lattner C., Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. *Proc. CGO Int. Symp*. 2004. P. 75–86. URL: https://doi.org/10.1109/CGO.2004.1281665 (дата звернення: 10.05.2024).

6. Mogensen P.K., Riseth A.N. Optim: a mathematical optimization package for Julia. *J. Open Source So/w.* 2018. Vo1. 3, no. 24. URL: https://doi.org/10.21105/joss.00615 (дата звернення: 14.05.2024).

7. Dunning I., Huchette J., Lubin M. JuMP: a modeling language for mathematical optimization. *SIAM Rev*. 2017. Vol. 59, no. 2. P. 295–320. URL: https://doi.org/10.1137/15M1020575 (дата звернення: 14.05.2024).

8. Udell M., Mohan K., Zeng D., et al. Convex optimization in Julia. *Proc. 1st Workshop for High Performance Technical Computing in Dynamic Languages.* 2014. P. 18–28. URL: https://doi.org/10.1109/HPTCDL.2014.5 (дата звернення: 16.05.2024).

9. Grant M., Boyd S., Ye У. Disciplined convex programming. In: Global optimization. Springer, 2006. P. 155–210.

10. Biel M., Johansson M. Efficient stochastic programming in Julia. URL: https://arxiv.org/abs/1909.10451 (дата звернення: 16.05.2024).

11. Rackauckas C., Nie Q. Differential Equations.jl – a performant and feature rich ecosystem for solving differential equations in Julia. *J. Open Res. Softw.* 2017. Vol. 5, no. 1, art. l5. URL: https://doi.org/10.5334/jors.151 (дата звернення: 16.05.2024).

12. Garstka M., Cannon M., Goulart P. COSMO: a conic operator splitting method for convex conic problems. URL: https://arxiv.org/abs/1901.10887 (дата звернення: 17.05.2024).

13. Castelani E.V., Lopes R., Wesley YI. et a1. RAFF.jl: robust algebraic fitting function in Julia. *J. Open Res. Softw.* 2019. Vol. 4, no. 39. URL: https://doi.org/10.21105/joss.01385 (дата звернення: 17.05.2024).

14. Otter M., Elmqvist H., Zimmer D., et al. Thermodynamic property and fluid modeling with modern programming language construct. *Proc. 13th Int. Modelica Conf.* Regensburg, Germany. 2019. P. 589–598. URL: https://doi.org/10.3384/ecp19157589 (дата звернення: 17.05.2024).

15. Pawar S., San O. CFD Julia: a learning module structuring an introductory course on computational fluid dynamics. *Fluids.* 2019. Vol. 4, no. 3, art. 159. URL: https://doi.org/10.3390/fluids4030l59 (дата звернення: 17.05.2024).

16. Frondelius T., Aho J. JuliaFEM – open-source solver for both industrial and academia usage. *Rakenteiden Mekaniikka.* 2017. Vo1. 50, no. 3. P. 229–233. URL: https://doi.org/10.23998/rm.64224 (дата звернення: 17.05.2024).

17. Revels J., Lubin M., Papamarkou T. Forward-mode automatic differentiation in Julia. URL: https://arxiv.org/abs/1607.07892 (дата звернення: 17.05.2024).

18. Rapo M., Aho J., Frondelius T. Natural frequency calculations with JuliaFEM. *Rakenteiden Mekaniikka.* 2017. Vol. 50, no. 3. P. 300–303. URL: https://doi.org/10.23998/rm.65040 (дата звернення: 20.05.2024).

19. Fairbrother J., Nemeth C., Rischard M., et a1. GaussianProcesses.j1: A Nonparametric Bayes package for the Julia language. URL: https://arxiv.org/abs/1812.09064 (дата звернення: 20.05.2024).

20. MATLAB. Math. Graphics.Programming. URL: https://www. mathworks. com/products/matlab.html (дата звернення: 22.05.2024).

21. Python 3.12.3. Documentation. URL: https://docs.python.org/3/ (дата звернення: 22.05.2024).

22. R Programming Language. Documentation. URL: https://www.r-project.org (дата звернення: 23.05.2024).

23. Julia Programming Language. Documentation v1.10. URL: https://docs.julialang.org/en/v1/ (дата звернення: 23.05.2024).

24. FORTRAN Programming Language. Documentation. URL: https://fortran-lang.org/index (дата звернення: 24.05.2024).

25. FORTRAN Packages. URL: https://fortran-lang.org/packages/ (дата звернення: 27.05.2024).

26. What are IMSL. URL: https://www.imsl.com (дата звернення: 27.05.2024).

27. ANSI. X3.9-1966 – FORTRAN. 1966. URL: https://wg5-fortran.org/ARCHIVE/Fortran66.pdf (дата звернення: 27.05.2024).

28. JuliaFEM.jl. Julia Documentation. URL: https://juliapackages.com /p/ juliafem (дата звернення: 30.05.2024).

29. Ferrite.jl. Julia Documentation. URL: https://ferrite-fem.github.io/Ferrite.jl/stable/ (дата звернення: 30.08.2024).

30. Dirichlet boundary conditions. URL: https://www.sciencedirect.com/topics /engineering/dirichlet-boundary-condition (дата звернення: 05.11.2024).

31. Neumann boundary conditions. URL: https://www.sciencedirect.com/topics/engineering/neumann-boundary-condition (дата звернення: 05.11.2024).

32. Furier Law. URL: https://en.wikipedia.org/wiki/Thermal_conduction (дата звернення: 05.11.2024).

33. Galerkin finite element method. URL: https://www.sciencedirect.com/ science/article/pii/0307904X82900051 (дата звернення: 05.11.2024).

34. SALOME CAE Opensource Platform. URL: https://www.salome-platform.org (дата звернення: 02.06.2024).

35. ParaView. Opensource post-processing visualization engine. URL: https://www.paraview.org (дата звернення: 02.06.2024).

36. On the contact domain method: A comparison of penalty and Lagrange multiplier implementations. URL: https://www.sciencedirect.com/ science/ article/abs/ pii/ S0045782511000120 (дата звернення: 11.11.2024).

37. Бібліотека FerriteMeshParser.jl. URL: https://juliapackages.com/p /ferritemeshparser (дата звернення: 12.11.2024).

38. Бібліотека Gmsh.jl. URL: https://juliapackages.com/p/gmsh (дата звернення: 12.11.2024).

39. Бібліотека FerriteGmsh.jl. URL: https://juliapackages.com/p/ferritegmsh (дата звернення: 12.11.2024).

40. Бібліотека WriteVTK.jl. URL: https://juliapackages.com/p/writevtk (дата звернення: 12.11.2024).

41. Stress Tensor Cauchy. URL: https://www.sciencedirect.com/topics/ engineering/cauchy-stress-tensor (дата звернення: 14.11.2024).

42. Von Mises yield criterion. URL: https://www.sciencedirect.com/topics /engineering/von-mises-criteria (дата звернення: 15.11.2024).

43. FEA Repository. URL: https://github.com/YuriyK2022/ferrite-fem/tree/master (дата звернення: 15.11.2024).