МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Кваліфікаційна наукова праця на правах рукопису

ГНЄЗДОВСЬКИЙ ОЛЕКСІЙ ВАЛЕНТИНОВИЧ

УДК 519.688:519.6:514.752

ДИСЕРТАЦІЯ

ВІДКРИТА ОБ’ЄКТНО-ОРІЄНТОВАНА АРХІТЕКТУРА СИСТЕМ СКІНЧЕННО-ЕЛЕМЕНТНОГО АНАЛІЗУ

Спеціальність: 122 Комп’ютерні науки
Галузь знань: 12 Інформаційні технології

Подається на здобуття наукового ступеня **доктора філософії**

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ О. В. Гнєздовський

Науковий керівник: **Кудін Олексій Володимирович**, кандидат фізико-математичних наук, доцент

Запоріжжя – 2023

АНОТАЦІЯ

Гнєздовський О. В. Відкрита об’єктно-орієнтована архітектура систем скінченно-елементного аналізу. – Кваліфікаційна наукова праця на правах рукопису. Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 122 “Комп’ютерні науки”. – Запорізький національний університет, Запоріжжя, 2023.

Дисертаційна робота присвячена розробці відкритої об’єктно-орієнтованої архітектури, яка може бути застосована при розробці як універсальних, так і спеціалізованих систем скінченно-елементного аналізу широких класів крайових задач.

В наш час велика кількість актуальних проблем науки і техніки призводять до необхідності розв’язання різних типів крайових задач. На жаль, для більшості з них на сьогодні неможливо отримати точні аналітичні розв’язки, що потребує створення відповідного математичного та програмного забезпечення для наближеного чисельного розв’язання диференційних рівнянь (звичайних та у частинних похідних) або їх систем.

Найбільш поширеним на сьогодні способом наближеного розв’язання крайових задач є метод скінченних елементів (МСЕ). Для його використання створено значну кількість як універсального, так і спеціалізованого програмного забезпечення (ПЗ). Серед найбільш відомих програм скінченно-елементного аналізу (пропрієтарних та з відкритим кодом) можна відзначити Abaqus, Ansys, FreeFEM, Mathematica, Nastran, OpenCAD та багато інших. Тут слід зазначити, що на сьогодні такого ПЗ розроблено (і постійно продовжує розроблятися) велика кількість, оскільки воно призначено як для розв’язання різних типів задач, так і різних форм застосування (комерційного, навчального, наукового, open source тощо). Крім того, постійно змінюються можливості обчислювальної техніки (широкого розповсюдження останнім часом отримали багатопроцесорні системи) і

вимоги практики (наприклад, необхідність враховувати застосування нових типів матеріалів, таких як композити або матеріали з пам’яттю).

Переважну більшість з наявних програм скінченно-елементного аналізу реалізовано мовами програмування C та C++. Проте, внесення змін в таке ПЗ для його адаптації до вирішення нових класів задач через складність C/C++ є нетривіальною процедурою. Тому на практиці виникає завдання створення таких програм для скінченно-елементного аналізу, які б, з одного боку, мали відкриту й просту архітектуру, а, з іншого боку, підтримували можливість легкого внесення в них змін для розширення функціональності. Іншими словами, мали б у порівнянні з ПЗ, яке написано на мовах C/C++, високу швидкість розробки та супроводу.

Останнім часом однією з найбільш популярних при розробці наукового ПЗ є мова програмування Python, яка початково створювалася для підвищення ефективності роботи програміста. Вона має легкий для розуміння синтаксис і зараз для неї розроблено велику кількість бібліотек, що реалізують різний математичний функціонал (наприклад, NumPy та SciPy, що використовуються для наукових розрахунків). Слід зазначити, що до сьогодні систем скінченно-елементного аналізу на Python розроблено відносно мало. Серед них можна виділити, наприклад, fempy, FEniCS, PolyFEM, SfePy, втім частина з них має лише python-інтерфейс, але написана на C++. Це можна пояснити тим фактом, що прийнято вважати швидкість роботи python-програм нижчою, ніж у аналогів, написаних на C++. Але, динамічна типізація Python і його можливості продуктивної розробки програм роблять його перспективним засобом створення нового ефективного ПЗ для чисельного вирішення широких класів крайових задач з використанням МСЕ.

Таким чином, створення нового ПЗ скінченно-елементного аналізу з відкритою архітектурою і програмним кодом на мові Python, яке б давало змогу користувачам розв’язувати широкі класи крайових задач на сучасних комп’ютерних платформах є актуальною й важливою задачею.

На логічному рівні будь-яке сучасне ПЗ для скінченно-елементного аналізу за своєю архітектурою може бути поділено на три окремі складові (підсистеми):

1) препроцесор – підсистема автоматизації підготовки вихідних даних для чисельного розрахунку (у більшості випадків це генератор скінченно-елементної моделі вихідної геометричної області, для якої виконується розрахунок);

2) процесор – підсистема, що безпосередньо реалізує скінченно-елементний розрахунок задачі (центральний елемент будь-якого ПЗ чисельного аналізу);

3) постпроцесор – підсистема, що автоматизує аналіз великих масивів числової інформації, отриманих як результат роботи процесору (частіше за все у вигляді різноманітних зображень полів досліджуваних функцій).

Генерація скінченно-елементної моделі, яку покладено на препроцесор, в загальному вигляді може бути поділена два окремих етапи:

1) побудова формального опису геометричної області, для якої виконується розрахунок, у певному форматі, який є придатним для подальшої автоматичної обробки;

2) генерація скінченно-елементної сітки на основі раніше отриманого опису геометричної моделі.

На сьогодні існує багато способів формального опису геометричних моделей областей складної форми (наприклад, твердотільне геометричне моделювання), проте найбільш універсальним серед них є функціональний підхід (FREP – Functional Representation), який базується на побудові деякого математичного співвідношення, що однозначно описує поверхню вихідної геометричної області. Найбільш відомим класом таких співвідношень є R-функції, які були запропоновані академіком В. Л. Рвачовим.

Опис функціональних співвідношень, що визначають границю області розрахунку, в загальному випадку потребує розробки відповідної спеціалізованої формальної мови, а також засобів трансляції її у машинний код. Проте, оскільки реалізований у дисертації постпроцесор написано на мові Python, яка за своєї

природою є інтерпретованою, описувати геометричні моделі дво- та тривимірних областей довільної конфігурації можна безпосередньо на цій мові. В дисертаційній роботі наведено опис запропонованої відкритої об’єктно-орієнтованої архітектурі препроцесора, який є зручним для його подальшої програмної реалізації.

У дисертації також описана запропонована об’єктно-орієнтована архітектура скінчено-елементного процесора, яка була реалізована на мові програмування Python. Була розроблена ієрархічна структура класів, що інкапсулюють об’єкт розрахунку, статичну й динамічну реалізацію МСЕ, скінченні елементи різних типів, дискретну модель вихідного об’єкту тощо. Завдяки простоті реалізації вдалося побудувати набір ефективних й інтуїтивно-зрозумілих класів, які дозволяють, з одного боку, виконувати чисельне розв’язання різних типів крайових задач, а, з іншого боку, надають можливість легко його розширювати для підвищення функціональності процесора.

Крім того, в дисертаційній роботі запропонована архітектура універсального постпроцесора та описана його програмна реалізація.

Таким чином, у дисертаційній роботі було вирішено актуальну задачу створення відкритої об’єктно-орієнтованої архітектури систем скінченно-елементного аналізу.

Ключові слова: метод скінченних елементів, об’єктно-орієнтована модель, Python, відкрита архітектура, препроцесор, процесор, постпроцесор.

ABSTRACT

Gnezdovsky O. V. Open object-oriented architecture of finite element analysis systems. – Qualifying scientific work on the rights of the manuscript. The

dissertation on competition of a scientific degree of the doctor of philosophy on a specialty 122 “Computer Science”. – Zaporizhia National University, Zaporizhia, 2023.

The thesis is devoted to the development of an open object-oriented architecture, which can be used in the development of both universal and specialized finite element analysis systems of wide classes of boundary value problems.

Nowadays, a large number of topical problems of science and technology lead to the need to solve various types of boundary problems. Unfortunately, for most of them today it is not possible to obtain exact analytical solutions, which requires the creation of appropriate mathematical and software for the approximate numerical solution of differential equations (ordinary and partial differential equations) or their systems.

The most common method of approximate solution of boundary value problems today is the finite element method (FEM). A significant amount of both universal and specialized software (software) has been created for its use. Abaqus, Ansys, FreeFEM, Mathematica, Nastran, OpenCAD and many others can be noted among the most well-known finite element analysis programs (proprietary and open source). It should be noted here that a large number of such software has been developed (and is constantly being developed), as it is intended for solving various types of problems and various forms of application (commercial, educational, scientific, open source, etc.). In addition, the capabilities of computing technology (multiprocessor systems have become widespread recently) and the requirements of practice (for example, the need to consider the use of new types of materials, such as composites or materials with memory) are constantly changing.

The vast majority of available finite element analysis programs are implemented in C and C++ programming languages. Unfortunately, making changes to such software to adapt it to solving new classes of problems due to the complexity of C/C++ is a non-trivial procedure. Therefore, in practice, the task of creating such programs for finite element analysis, which would, on the one hand, have an open and simple architecture, and, on the other hand, support the possibility of easy changes to them to expand

functionality, arises in practice. In other words, compared to software written in C/C++, they would have a high development speed.

Recently, one of the most popular in the development of scientific software is the Python programming language, which was originally created to increase the efficiency of the programmer's work. It has an easy-to-understand syntax and now a large number of libraries have been developed for it, implementing various mathematical functionality (for example, NumPy and SciPy, used for scientific calculations). However, to date, relatively few finite element analysis systems have been developed in Python. Among them, you can highlight, for example, fempy, FEniCS, PolyFEM, SfePy, however, some of them have only a python interface, but are written in C++. This can be explained by the fact that it is generally accepted that the speed of python programs is lower than that of their counterparts written in C++. Nevertheless, Python's dynamic typing and its capabilities of productive program development make it a promising means of creating new efficient software for numerically solving broad classes of boundary value problems using MSE. Therefore, the development of an open software architecture for MSE is an urgent task in our time.

Thus, creating a new finite-element analysis software with an open architecture and software code that would enable users to solve wide classes of boundary value problems on modern computer platforms is an urgent and important task.

At the logical level, any modern software for finite element analysis can be divided into three separate components (subsystems) by its architecture:

1) preprocessor – a subsystem for automating the preparation of initial data for numerical calculation (in most cases, it is a generator of a finite-element model of the initial geometric area for which the calculation is performed);

2) processor is a subsystem that directly implements the finite element calculation of the problem (the central element of any numerical analysis software);

3) post processor – a subsystem that automates the analysis of large arrays of numerical information obtained as a result of the processor's work (most often in the form of various images of the fields of the investigated functions).

The generation of the finite-element model, which is assigned to the preprocessor, can generally be divided into two separate stages:

1) construction of a formal description of the geometric area for which the calculation is performed, in a certain format that is suitable for further automatic processing;

2) generation of a finite-element grid based on the previously obtained description of the geometric model.

Today, there are many ways to formally describe geometric models of areas of complex shape (for example, solid geometric modeling), but the most universal among them is the functional approach (FREP - Functional Representation), which is based on the construction of some mathematical relationship that uniquely describes the surface of the original geometric area . The most famous class of such ratios are R-functions, which were proposed by Academician V. L. Rvachev.

The description of the functional relationships that determine the boundary of the calculation area, in general, requires the development of an appropriate specialized formal language, as well as means of translating it into machine code. However, since the post-processor implemented in the thesis is written in the Python language, which is interpreted by its nature, it is possible to describe geometric models of two- and three-dimensional areas of arbitrary configuration directly in this language. The dissertation provides a description of the proposed open object-oriented preprocessor architecture and its software implementation.

The thesis also describes the proposed object-oriented architecture of the finite element processor, which was implemented in the Python programming language. A hierarchical structure of classes encapsulating the calculation object, static and dynamic implementation of MSE, finite elements of various types, discrete model of the original object, etc. was developed. Thanks to the simplicity of the implementation, it was possible to build a set of efficient and intuitively understandable classes, which allow, on the one hand, to perform the numerical solution of various types of boundary value

problems, and, on the other hand, provide an opportunity to easily expand it to increase the functionality of the processor.

In addition, the dissertation proposed the architecture of a universal postprocessor and described its software implementation.

Thus, the actual task of creating an open object-oriented architecture of finite element analysis systems was solved in the dissertation work.

Keywords: finite element method, object-oriented model, Python, open architecture, preprocessor, processor, postprocessor..

СПИСОК ПУБЛІКАЦІЙ ЗДОБУВАЧА ЗА ТЕМОЮ ДИСЕРТАЦІЇ, В ЯКИХ ОПУБЛІКОВАНІ ОСНОВНІ НАУКОВІ РЕЗУЛЬТАТИ ДИСЕРТАЦІЇ:

1) Морозов Д. Н., Гнездовский А. В., Мыльцев А. М., Толок А. В. Когнитивная компьютерная графика в процессе решения оптимизационных задач математического моделирования. *Прикладна геометрія та інженерна графіка*. 2010. Вип. 86. С. 112-117.

2) Морозов Д. М., Юрєчко В. З., Гнєздовський О. В. Скінченно-елементний підхід до визначення напружено-деформованого стану пористого матеріалу засобами Pyton. *Енергоефективність в будівництві та архітектурі: науково-технічний збірник*. 2017. Вип. 9. С. 174-178.

3) Игнатченко М. С., Кудин А. В., Гнездовский А. В. Объектно-ориентированная реализация библиотеки конечно-элементного анализа на языке программирования Python. *Вісник Запорізького національного університету. Фізико-математичні науки*. 2020. № 1. С. 138-147.

4) Богуславська А. М., Гребенюк С. М, Морозов Д. М., Гнєздовський О. В. Розрахунок напружено-деформованого стану металевих паль. *Вісник Запорізького національного університету. Фізико-математичні науки*. 2020. № 2. С. 14-19.

5) Игнатченко М. С., Гнездовский А. В. Объектно-ориентированное моделирование задач механики. *Сучасні проблеми машинобудування*. Конференція молодих вчених та спеціалістів: зб. тез доп. Харків: Інститут проблем машинобудування НАН України, 2016. С. 23-24.

6) Гнєздовський О. В. Застосування мови програмування Python для розробки систем скінченно-елементного аналізу. *Актуальні проблеми математики та інформатики*: Збірка тез доповідей Тринадцятої Всеукраїнської, двадцятої регіональної наукової конференції молодих дослідників (м. Запоріжжя, 17 червня 2022 р.). Запоріжжя: ЗНУ, 2022. С. 8-9.

7) Hniezdovskyi O., Kudin O., Belokon Y., Kruglyak D., Ilin S. Designing an object-oriented architecture for the finite element simulation of structural elements. Eastern-European Journal of Enterprise Technologies, 2022, 6(2-120), pp. 78-84 (SCOPUS).

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ГЕ – граничний елемент

ГМЖ – глобальна матриця жорсткості

ЛМЖ – локальна матриця жорсткості

МСЕ – метод скінченних елементів

ООП – об’єктно-орієнтоване програмування

ПЗ – програмне забезпечення

СЕ – скінченний елемент

СЛАР – система лінійних алгебраїчних рівнянь

BREP – boundary representation

CSG – constructive solid geometry

DSL – domain-specific language

FREP – function representation

FEA – finite element analysis

UML – unified modeling language

ВСТУП

Актуальність теми. В наш час велика кількість актуальних науково-технічних проблем призводить до необхідності розв’язання різних типів крайових задач. На жаль, для більшості з них на сьогодні неможливо отримати точні аналітичні розв’язки, що потребує створення відповідного математичного та програмного забезпечення для наближеного чисельного розв’язання диференційних рівнянь (звичайних та у частинних похідних) або їх систем.

Найбільш поширеним на сьогодні способом наближеного розв’язання широких класів крайових задач є метод скінченних елементів (МСЕ) [1]. Застосування МСЕ без використання комп’ютерної техніки і відповідного математичного та програмного забезпечення фактично неможливе, тому на сьогодні розроблено значну кількість програмного забезпечення (ПЗ), яке призначене для автоматизації проведення чисельних розрахунків із застосуванням МСЕ. Все ПЗ скінченно-елементного аналізу можна умовно поділити на дві великі групи: комерційне (пропрієтарне) та відкрите (open source). Серед найбільш відомих систем скінченно-елементного аналізу можна відзначити такі програмні продукти: Abaqus [2], Ansys [3], COMSOL [4], dial.II [5], DIANA FEA [6], FreeFEM [7], MATLAB [8], MSC Nastran [9], OpenCAD [10], SOLIDWORKS [11] та ін. [12-15]. Кількість та різноманіття такого ПЗ стрімко зростає, оскільки на практиці все частіше використовуються нові типи матеріалів (композити, матеріали з пам’яттю і таке інше), а також нові типи обчислювальної техніки (багатопроцесорні системи зі спільною або розподіленою пам’яттю). Таким чином, для продуктивного використання наявної комп’ютерної техніки та врахування особливостей нових конструктивних матеріалів необхідно створювати нове ПЗ скінченно-елементного аналізу, особливо таке, що належить до класу open source.

Таким чином, можна констатувати той факт, що на сьогодні проблема створення відкритого ПЗ скінченно-елементного аналізу, яке б давало можливість ефективного вирішення актуальних науково-технічних проблем, є актуальною.

Зв’язок роботи з науковими програмами, планами, темами. Одержані в дисертаційній роботі результати повністю відповідають основним напрямам наукових досліджень, що виконуються у Запорізькому національному університеті. Зокрема, робота виконувалася у відповідності до плану науково-технічних робіт Запорізького національного університету при виконанні науково-дослідної теми “Математичне та програмне забезпечення наукових досліджень” (номер державної реєстрації 0121U114694), яка виконувалась у межах основного робочого часу викладачів.

Мета і задачі дослідження. Метою дисертаційного дослідження є розв’язання актуальної задачі створення відкритої об’єктно-орієнтованої архітектури систем скінченно-елементного аналізу, яка може бути використана при розробці як універсального, так і спеціалізованого ПЗ, призначеного для аналізу широких класів крайових задач.

Досягнення поставленої мети передбачає розв’язання наступних задач:

1) проведення критичного аналізу наявних систем скінченно-елементного аналізу та їх програмної архітектури і способів застосування;

2) розробки відкритої об’єктно-орієнтованої архітектури систем скінченно-елементного аналізу, яка б дозволяла легко додавати новий функціонал до такого типу ПЗ;

3) розробки відповідних алгоритмів реалізації препроцесора, процесора і постпроцесора, які базуються на запропонованій архітектурі;

4) виконання розробки відповідного програмного забезпечення;

5) виконання тестових розрахунків, які б дозволили виконати доведення ефективності запропонованої архітектури та алгоритмів.

Об’єктом дослідження є процес розробки скінченно-елементних програмних систем.

Предметом дослідження є об’єктно-орієнтована архітектура відкритого програмного забезпечення скінченно-елементного аналізу.

Методи дослідження базуються на застосуванні апарату обчислювальної математики, прикладного та системного програмування, об’єктно-орієнтованого аналізу та паралельних розрахунків.

Наукова новизна одержаних результатів полягає у розв’язанні актуальної задачі створення ефективної об’єктно-орієнтованої архітектури систем скінченно-елементного аналізу з відкритим програмним кодом.

При виконанні дисертаційної роботи отримано такі наукові результати:

1) *вперше* запропоновано ієрархічну об’єктно-орієнтовану архітектуру систем скінченно-елементного аналізу, яка на відміну від наявних аналогів дозволяє легко додавати нові типи розрахунків (статика, динаміка, фізична нелінійність) та враховувати геометричні особливості конструкцій, що розраховуються;

2) *отримали подальший розвиток* алгоритми розв’язання крайових задач методом скінченних елементів, що базуються на запропонованій об’єктно-орієнтованій архітектурі;

3) *вперше* програмно реалізовано на мові Python система скінченно-елементного аналізу, яка базується на запропонованій архітектурі і може виконувати розрахунок широкого кола задач в статичній та динамічній постановці;

4) *вперше* отримано чисельні розрахунки певних класів крайових задач.

Практичне значення одержаних результатів. Запропонована в дисертації архітектура систем скінченно-елементного аналізу дозволяє істотно підвищити продуктивність програмної реалізації відповідного ПЗ. Розроблений програмний засіб pyfem дає можливість автоматизувати всі етапи практичного використання МСЕ – генерацію дискретних геометричних моделей (препроцесінг); чисельний розрахунок (процесінг) та візуалізацію чисельних результатів (постпроцесінг). Система pyfem реалізована у вигляді бібліотеки класів з відкритим початковим

кодом із застосуванням стандартних бібліотек мови Python NumPy, SciPy та інших.

Апробація результатів дослідження. Результати дисертаційного дослідження були оприлюднені на таких наукових конференціях:

5) Конференції молодих вчених та спеціалістів “Сучасні проблеми машинобудування” (Харків, 2016 р.);

6) Тринадцятій Всеукраїнській, двадцятій регіональній науковій конференції молодих дослідників “Актуальні проблеми математики та інформатики” (Запоріжжя, 2022 р.).

Публікації. Основні положення роботи було опубліковано у 7 наукових працях, із них: 1 стаття у виданні, яке індексується у наукометричній базі Scopus; 3 опубліковані у виданнях, що належать до категорії Б; а також 3 тез доповідей на науково-практичних конференціях.

Структура та обсяг роботи. Дисертація складається зі вступу, чотирьох розділів, які містять 19 підрозділів, висновків, списку використаних джерел та додатків. Загальний обсяг дисертації становить 122 сторінки, у т. ч. основного тексту – 105 сторінок. Список використаних джерел налічує 89 найменувань (8 сторінок). Додатки – 10 сторінок.

1 АНАЛІЗ СТАНУ ПРОБЛЕМИ

1.1 Огляд сучасних систем скінченно-елементного аналізу

Розвиток науки і техніки в сьогоднішніх умовах в основному базується на застосуванні обчислювальної техніки для наближеного розв’язання широкого кола крайових задач. Однією з найбільш поширених на практиці наближених технік розв’язання диференціальних та інтегральних рівнянь або їх систем є застосування методу скінченних елементів [1, 16, 17]. МСЕ ґрунтується на ідеї заміни вихідної суцільної геометричної області певною дискретною моделлю, утвореною кінцевою сукупністю геометричних областей простої форми, що не перетинаються, – скінченних елементів (СЕ). В якості СЕ частіше за все використовують трикутники та чотирикутники у двовимірному випадку та трикутні піраміди (тетраедри) і чотирикутні призми (куби) у тривимірному випадку. Для кожного СЕ може бути легко побудована лінійно-незалежна система базисних функцій (функцій форми), які апроксимують шукану функцію на цьому СЕ. Такий підхід дозволяє відмовитися від необхідності побудови лінійно-незалежної системи базисних функцій, яка задовольняє граничним умовам, для всієї вихідної геометричної області, що є неможливим в разі її складної форми.

Загальний алгоритм використання МСЕ може бути описаний наступним чином:

1) генерація дискретної (скінченно-елементної) моделі вихідної геометричної області;

2) підстановка апроксимації шуканих функцій у вихідне рівняння і формування так званих локальної матриці жорсткості (ЛМЖ) для кожного СЕ;

3) об’єднання ЛМЖ в єдину матрицю – глобальну матрицю жорсткості (ГМЖ);

4) обчислення вузлових значень навантажень;

5) врахування граничних умов;

6) розв’язання отриманої системи лінійних алгебраїчних рівнянь (СЛАР);

7) розрахунок додаткових результатів (наприклад, деформацій та напружень за отриманими переміщеннями).

Кожен з вищенаведених кроків потребує використання комп’ютерної техніки, тому це передбачає необхідність розробки спеціалізованого ПЗ для скінченно-елементного аналізу, яке в англомовних джерелах прийнято називати FEA (FEA – Finite Element Analysis) Software [18, 19].

FEA-системи можна класифікувати різними способами: проприєтарні та вільні; універсальні та спеціалізовані, програми або бібліотеки і так далі. Серед найбільш відомих комерційних FEA-систем можна виділити Abaqus [2], Ansys [3], COMSOL [4], DIANA-FEA [6], MSC Nastran [9], SOLIDWORKS [11] та ін. [12]. Серед ПЗ з відкритим програмним кодом слід відзначити Elmer FEM [20], FENRIS [21], FreeFEM [7], Netgen/NGSolve [22], OpenCAD [10] та ін. [14, 15]. До найбільш відомих універсальних (повнофункціональних) FEA відносяться Ansys [3], SOLIDWORKS [11], а серед спеціалізованих систем можна, наприклад, відзначити вітчизняну розробку МІРЕЛА+ [23], що призначена для аналізу напружено-деформованого стану конструкцій з еластомерів. Серед систем, реалізованих у вигляді бібліотек підпрограм або класів, призначених для скінченноелементного розрахунку, можна відзначити fem [24], QFEM [15], МІРЕЛА+ [23] та ін. [21, 25].

Сучасні повнофункціональні FEA-системи зазвичай складаються з трьох базових підсистем:

1) препроцесора – підсистеми автоматизації підготовки вихідних даних для розрахунку (для FEA це зазвичай генератор скінченно-елементної моделі вихідної геометричної області);

2) процесора – ядра FEA, яке безпосередньо реалізує скінченно-елементний розрахунок задачі (статика, динаміка, термопружність тощо);

3) постпроцесора – підсистеми, яка автоматизує аналіз отриманих числових результатів, наприклад, шляхом їх певної візуалізації. (рис. 1.1).

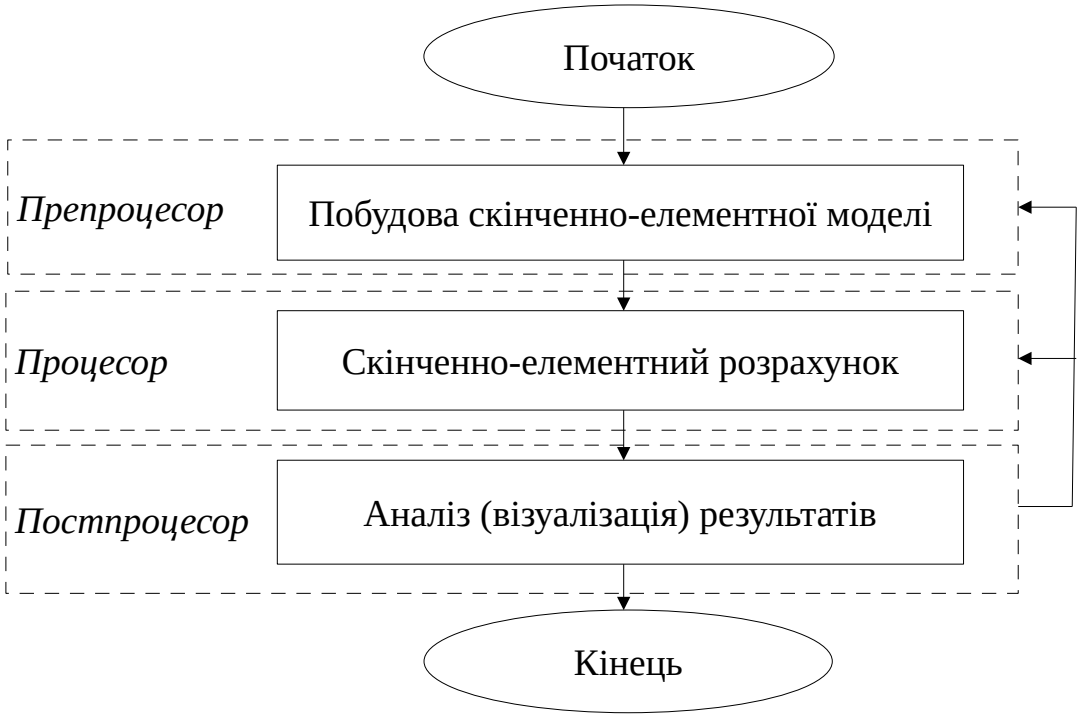

Рис. 1.1 – Типова структура системи FEA

Як вже зазначалося, функцією препроцесора є автоматизація побудови СЕ-моделі вихідної геометричної області, для якої виконується розрахунок. В загальному випадку ця задача поділяється на дві взаємопов’язані складові:

1) безпосередньо геометричне моделювання, тобто побудова деякого формалізованого опису вихідної області, придатного для подальшої автоматизованої обробки;

2) генерація на базі наявної геометричної моделі дискретної моделі, яка складається з елементів певної форми (або деякої комбінації СЕ різної форми, наприклад, трикутників і чотирикутників).

Перша задача є достатньо складною і творчою, особливо для геометричних областей нетипової форми. Вона погано підлягає автоматизації і, зазвичай, сучасні

препроцесори лише надають певний інструментарій користувачу для виконання геометричного моделювання. Ґрунтовний огляд наявних підходів до геометричного моделювання наведено в [26, 27]. Проблема автоматичної генерації дискретних моделей на сьогодні добре досліджена. Існує ціла низка підходів, за допомогою яких можна виконувати автоматичне розбиття вихідної геометричної моделі на заданий тип СЕ. Серед них можна відзначити алгебраїчні, диференційні, варіаційні методи, методи генерації неструктурованих сіток, адаптивні та фронтальні методи тощо [28]. Препроцесори можуть бути як невід'ємною складовою повнофункціональних FEA, наприклад, Ansys [3], так і окремими програмними продуктами. Серед останніх можна виділити ANSA [29], Gmsh [30], GRUMMP [31], Netgen/NGSolve [22], TetGen [32] та ін. [33].

Головною складовою будь-якої FEA-системи є процесор. Він безпосередньо реалізує той чи інший алгоритм застосування МСЕ для розв'язання конкретного класу задач. Як правило, цей алгоритм дозволяє побудувати для кожного СЕ відповідні йому матриці жорсткості, маси та демпфування, які залежать від використаного енергетичного функціоналу (варіаційного принципу). Всі подальші функції процесору (або вирішувача) досить стандартні: додавання локальних матриць СЕ до відповідних їм глобальних; врахування крайових умов; формування СЛАР з урахуванням крайових умов і розв'язання СЛАР. Архітектура процесора в загальному випадку залежить від типу задач, на які він орієнтований (наприклад, статика або динаміка), проте у більшості випадків вона є закритою ((тобто відсутня можливість легкого та швидкого розширення та адаптації до нових умов використання). На сьогодні розроблено велику кількість процесорів, які є як складовими FEA-систем, так і окремими програмними продуктами. Серед останніх можна відзначити GetFEM++ [34], hp-FEM [35], MFEM [36], OFELI [37], OpenSees [38] та ін. [13-15, 33].

Оскільки результатом роботи будь-якого процесора FEA-системи є великий масив числової інформації, то виникає задача підвищення наочності його аналізу. З цією метою розробляються так звані постпроцесори, які, з одного боку,

автоматизують аналіз числової інформації (зазвичай, шляхом її певної візуалізації у вигляді різноманітних графіків або кольорових зображень розподілу значень отриманої функції – фазової змінної по області розрахунку), а, з іншого боку, можуть синтезувати додаткову інформацію (наприклад, розрахунок інтенсивності напружень на основі значень раніше отриманих компонент тензору напружень) [39].

Як і у випадку препроцесора, постпроцесор може бути як невід’ємною складовою повнофункціональних FEA-систем (наприклад, Ansys [3] або SOLIDWORKS [11]), так і окремо реалізованим програмним засобом, наприклад, FEMDS [40], Gmsh [30], Netgen/NGSolve [22], MIRELA+ [41, 42] та ін. [43, 44].

Крім візуалізації розподілу полів фазової змінної по області розрахунку (переміщень, деформацій, напружень) постпроцесор може виконувати велику кількість допоміжних операцій: будувати різноманітні графіки, виконувати анімацію змін результатів нестаціонарних розрахунків, а також, як вже зазначалося, виконувати синтез додаткових результатів (наприклад, розрахунок коефіцієнту інтенсивності напружень за відомими напруженнями у механіці руйнування).

1.2 Постановка мети та задач дослідження

Аналіз великої кількості наявних FEA-систем показав, що у більшості випадків універсальні системи скінченно-елементного аналізу, призначені для розв’язання широких класів крайових задач, є пропрієтарними. Таким чином, у загальному випадку виникає потреба у створенні відкритих універсальних FEA, які були б придатні для аналізу широкого кола задач. Можливим варіантом вирішення цієї проблеми є створення відкритої архітектури FEA-систем, яка б

дозволяла легко розширювати наявний функціонал для розв’язання нових типів задач або використовувати нові методи (алгоритми).

Таким чином, підсумовуючи все вищезазначене, можна зробити наступний висновок: на сьогодні є актуальною задача створення таких універсальних систем скінченно-елементного аналізу з відкритим програмним кодом (open source), які дозволять користувачам розв’язувати широкі класи крайових задач різними методами.

Для досягнення поставленої мети необхідно:

1) виконати критичний аналіз наявних систем скінченно-елементного аналізу, а також їх програмної архітектури;

2) розробити відкриту архітектуру FEA-системи, яка дозволить користувачу розширювати можливості FEA-системи для використання нових типів СЕ або алгоритмів розрахунку (статика, динаміка, фізична нелінійність тощо);

3) на базі запропонованої архітектури розробити відповідну FEA-систему із застосуванням сучасної мови програмування Python, як однієї з найбільш популярних на сьогодні при реалізації наукового ПЗ;

4) виконати низку обчислювальних експериментів для підтвердження ефективності запропонованої архітектури.

Висновки до розділу 1

Отже, враховуючи проведений аналіз відповідної предметної області, можна зробити висновок про те, що задача розробки відкритої архітектури систем скінченно-елементного аналізу, яка б дозволяла розширювати можливості чисельного розв’язання широкого класу крайових задач є актуальною.

Розв’язання цієї задачі передбачає не тільки розробки відповідних структур даних, методів і алгоритмів, а й виконання цілої низки тестових розрахунків для верифікації запропонованого підходу.

Основні науково-практичні результати першого розділу опубліковано в роботах [45-48].

2 ВІДКРИТА АРХІТЕКТУРА СИСТЕМ СКІНЧЕННО-ЕЛЕМЕНТНОГО АНАЛІЗУ КРАЙОВИХ ЗАДАЧ

2.1 Типова схема моделювання із застосуванням методу скінченних елементів

Загальна концепція використання МСЕ (як і більшості інших чисельних методів розв’язання крайових задач) може бути схематично представлена наступним чином (рис. 2.1).

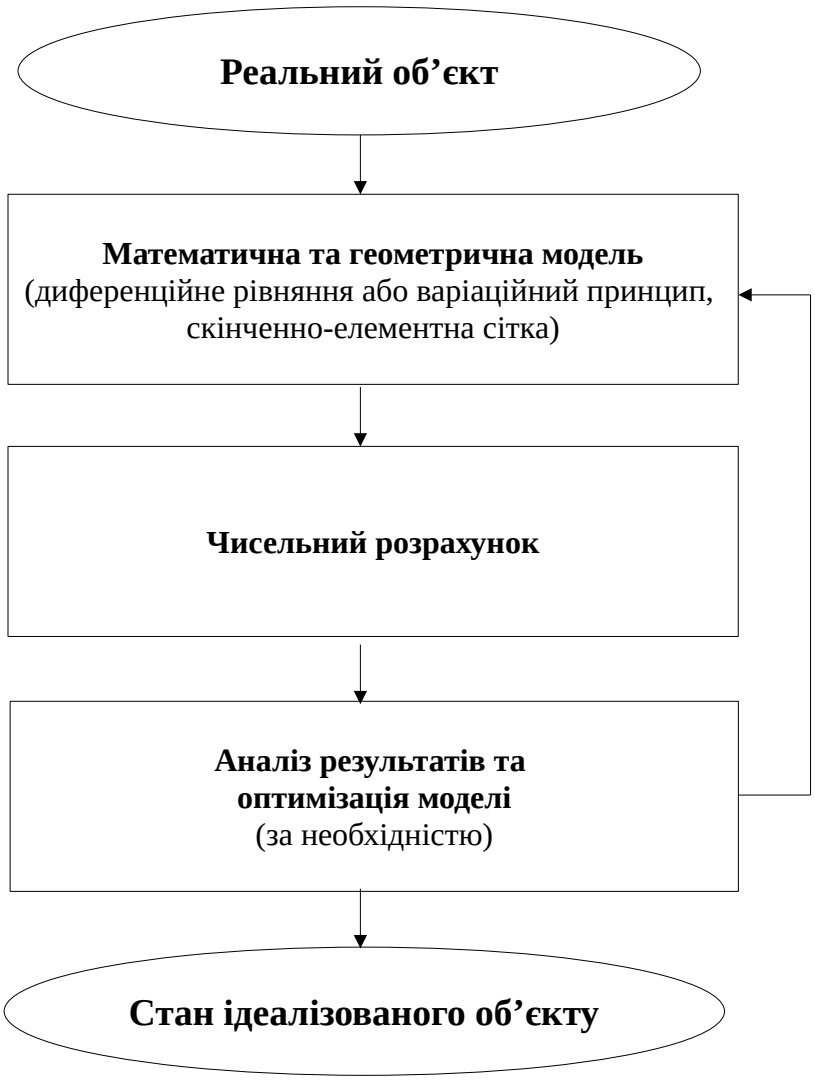

Рис. 2.1 – Типова схема комп'ютерного моделювання із застосуванням МСЕ

З цієї схеми логічно витікає загальна архітектура FEA-системи, яка наведена на рис. 1.1. Так, дійсно, на першому етапі дослідження реального об’єкта відбувається побудова його математичної та геометричної моделі (будемо їх розрізняти при застосуванні МСЕ). Математична модель зазвичай описується у вигляді певних диференційних рівнянь (або їх систем) або варіаційних принципів (функціоналів, що описують закон збереження енергії) [49].

Такій схемі моделювання природньо відповідає структура FEA-системи, яка складається з трьох базових компонентів: препроцесора, процесора і постпроцесора, кожен з яких автоматизує відповідну стадію моделювання.

2.2 Архітектура препроцесора

В якості геометричної моделі використовується певний формалізований опис топології вихідного геометричного об’єкту (придатний до подальшої автоматичної обробки), а також побудоване на його основі дискретне (скінченно-елементне) представлення вихідного об’єкту. Дискретна модель зазвичай утворюється з елементів певного типу (трикутників або чотирикутників у двовимірному випадку, тетраедрів або чотирикутних призм у тривимірному випадку тощо), але може бути і гібридною, тобто утвореною з елементів різних типів.

Після побудови математичної і геометричної моделі на основі певного алгоритму, який залежить від типу задачі (статика, динаміка, фізична нелінійність і т. п.), виконується чисельний розрахунок, причому параметрами цього алгоритму є як дискретна, так і математична модель.

Отримані чисельні результати аналізуються на предмет відповідності фізичному змісту задачі, наявному експерименту, результатам, отриманим іншими

авторами і т. п. При необхідності в параметри розв’язання задачі вносяться необхідні корективи і розрахунок повторюється.

Найбільш важливою з точки зору ефективності чисельного моделювання при застосуванні МСЕ є якість скінченно-елементної моделі. Від її адекватності і відповідності вихідному геометричному об’єкту напряму залежить точність розрахунку. При цьому задача автоматизації побудови скінченно-елементних моделей у загальному випадку є складною і досі актуальною. Розглянемо типову архітектуру препроцесора FEA-системи. Звісно, її структура залежить від вживаного алгоритму дискретизації (фронтальний, Рапперта, Шевчука) [50-53]. Типова схема геометричного моделювання з подальшою дискретизацією може бути представлена таким чином (рис. 2.2).

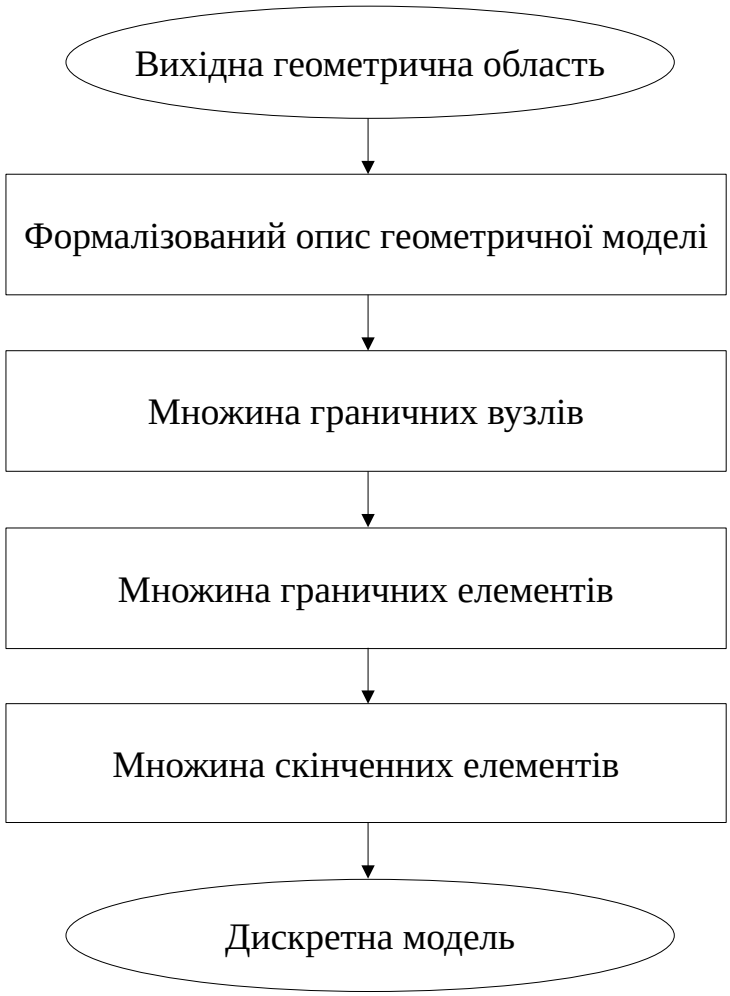

Рис. 2.2 – Типова схема побудови дискретної моделі

Такому узагальненому алгоритму побудови СЕ-розбиття в цілому відповідає наступна типова архітектура препроцесора (рис. 2.3). Центральне місце в такій архітектурі посідають структури даних, що призначені для зберігання інформації про топологію (форму) вихідного геометричного об’єкта і його дискретну модель.

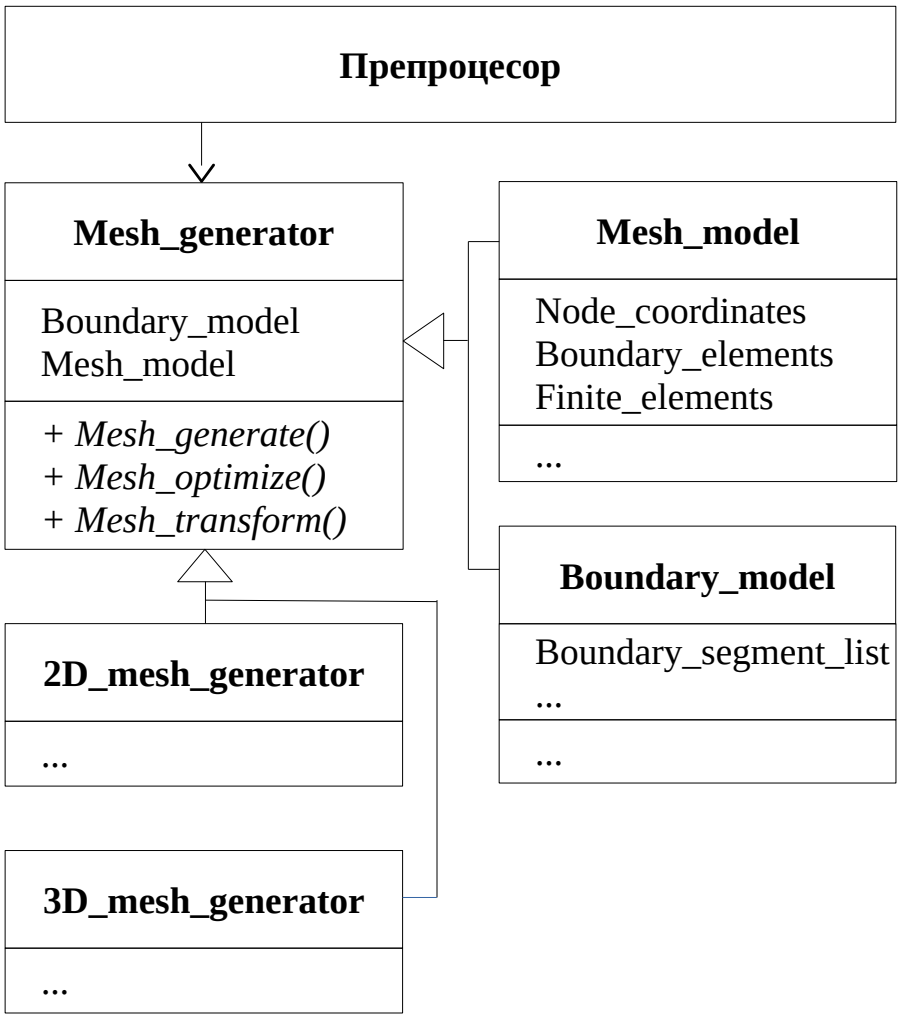

Рис. 2.3 – Типова архітектура препроцесора

Структура даних для збереження інформації про вихідну геометрію прямо залежить від обраного способу опису моделі. Найбільш популярними на сьогодні способами схемами подання інформації про геометрію об’єкту є:

1) конструктивна блокова геометрія (CSG – Constructive Solid Geometry) [54], яка дозволяє об’єднувати в єдиний об’єкт сімейство наявних геометричних

примітивів (твердих тіл) у вигляді результату логічних операцій диз'юнкції, кон'юнкції та інверсії;

2) граничне подання (BREP – Boundary Representation) [55], яке дає можливість представити будь-який геометричний об'єкт у вигляді певної композицію геометричних областей, які обмежують його внутрішню частину (тобто належать його границі);

3) функціональне подання (FREP – Function Representation) – дозволяє представити будь-яку геометричну область у вигляді деякого аналітичного співвідношення [56, 57].

Найбільш відомим способом реалізації FREP є використання функцій В. Л. Рвачова, які отримали назву R-функцій [56]. Згідно з теорією В. Л. Рвачова будь-яка геометрична область $\Omega \in \mathbb{R}^3$ може бути аналітично описана у вигляді деякої аналітичної функції $F(x,y,z)$, для якої виконуються наступні співвідношення: $F(x,y,z) \geq 0$, якщо $(x,y,z) \in \Omega$ ($F(x,y,z)=0$, для будь-яких $(x,y,z) \in \partial\Omega$, де $\partial\Omega$ – границя області Ω) [56].

Як і у випадку застосування CSG, при використанні R-функцій для конструювання геометричних моделей використовуються операції кон'юнкції, диз'юнкції та інверсії. Функція $F_1 \wedge F_2$ називається R-кон'юнкцією, якщо вона визначається співвідношенням $F_1 \wedge F_2 = (F_1 + F_2 - \sqrt{F_1^2 + F_2^2})/2$. Аналогічно R-диз'юнкцією називається функція $F_1 \vee F_2 = \left(F_1 + F_2 + \sqrt{F_1^2 + F_2^2}\right)/2$. Вираз $\overline{F} = -F$ називається R-інверсією.

Наприклад, геометрична фігура "Атом" (рис. 2.4) може бути описана наступною R-функцією: $F(x,y,z) = F_1(x,y,z) \vee F_2(x,y,z)$.

Тут $F_1(x,y,z) = 9 - x^2 - y^2 - z^2$;

$F_2(x,y,z) = F_3(x,y,z) \vee F_4(x,y,z)$;

$F_3(x,y,z) = F_5(x,y,z) \vee F_6(x,y,z)$;

$F_4(x,y,z) = 10\sqrt{(x^2+y^2)} - (x^2+y^2+z^2) - 24$;

$F_5(x,y,z) = 10\sqrt{(x^2+z^2)} - (x^2+y^2+z^2) - 24$;

$$F_6(x,y,z)=10\sqrt{(y^2+z^2)}-(x^2+y^2+z^2)-24.$$

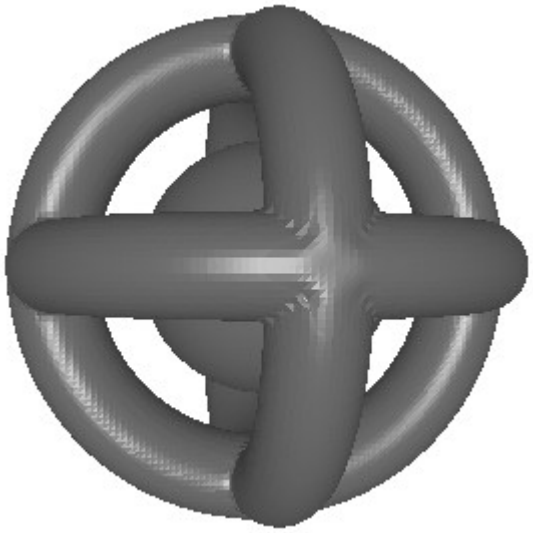

Рис. 2.4 – Геометрична фігура "Атом"

Головною проблемою практичного використання R-функцій є їх неявна природа. Це приводить до певних складностей при пошуку вузлів, які розташовані на границі геометричних областей, аналітично описаних із застосуванням цих функцій. Стандартною стратегією, яка використовується на практиці, є побудова певної апроксимації поверхні вихідної геометричної області із застосуванням деякого множини граничних вузлів, для яких виконується співвідношення $F(x,y,z)\approx 0$). Огляд алгоритмів, які дозволяють здійснювати пошук вузлів на границі вихідного об’єкта наведено в роботі [58]. Головною їх ідеєю є побудова певної так званої “супер-області” навколо вихідного геометричного об’єкта а потім її сканування для виявлення множини вузлів, для яких R-функція приймає нульове або близьке до нього значення (рис. 2.5).

Ґрунтовний огляд алгоритмів побудови та уточнення апроксимації поверхні функціонально заданих об’єктів наведено в роботах [26, 59]. Таким чином, загальна схема побудови дискретної моделі геометричної області у разі її функціонального задання складається з наступних кроків:

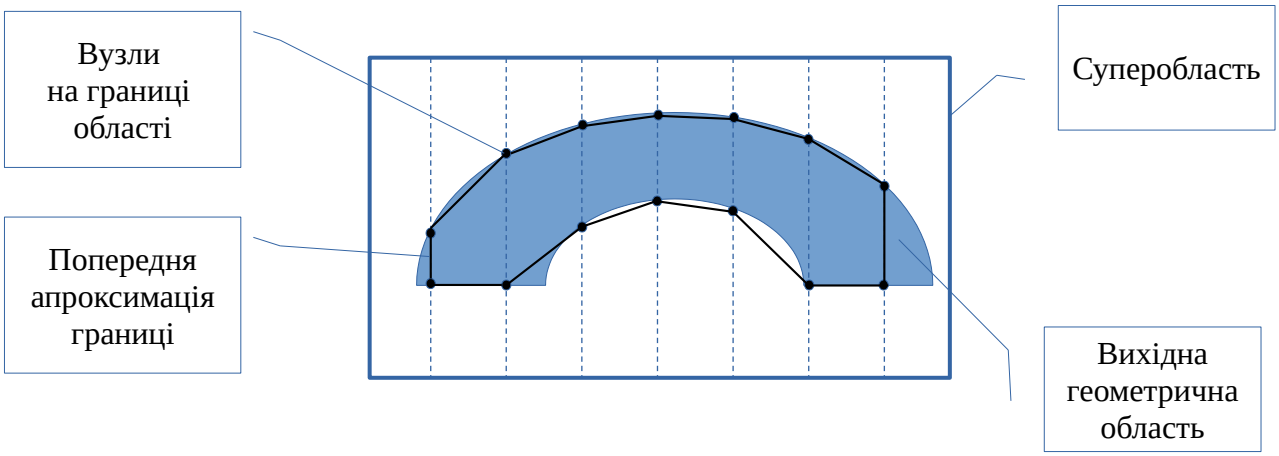

Рис. 2.5 – Апроксимація поверхні неявно заданої геометричної області

1) створення аналітичного опису вихідної геометричної області із застосуванням R-функцій;

2) пошук множини точок на границі області;

3) побудова попередньої апроксимації границі та її оптимізація;

4) побудова попередньої скінченно-елементної моделі та її оптимізація (рис. 2.6).

Такій послідовності дій відповідає наступна архітектура препроцесора FEA, діаграма UML (Unified Modeling Language) [60] якої наведена на рис. 2.7 (в деталізації для двовимірного випадку з використанням трикутних скінченних елементів).

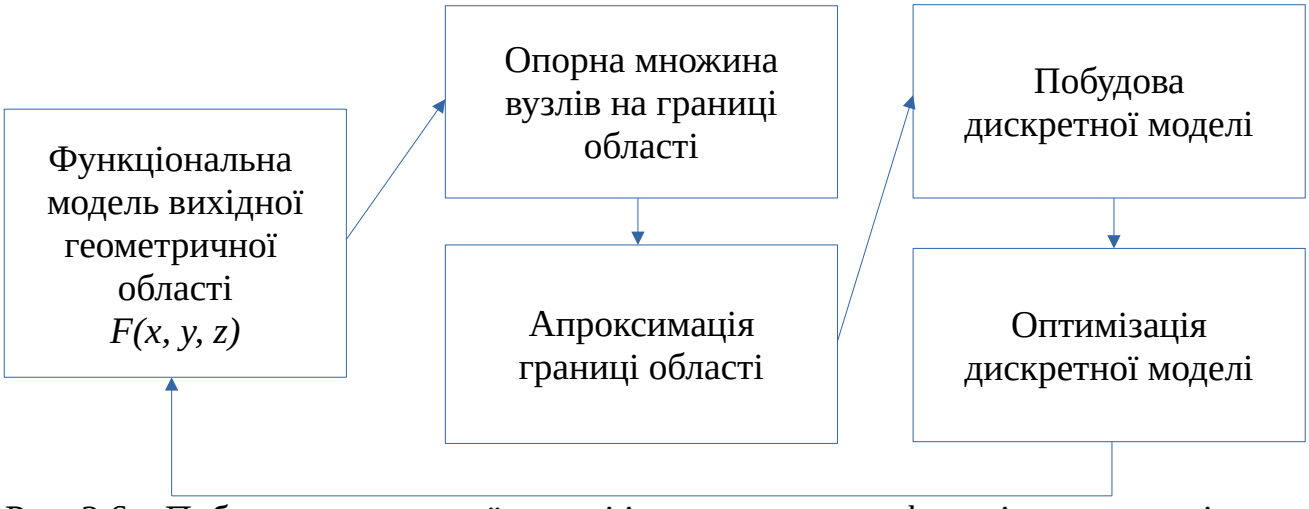

Рис. 2.6 – Побудова дискретної моделі із застосуванням функціонального підходу

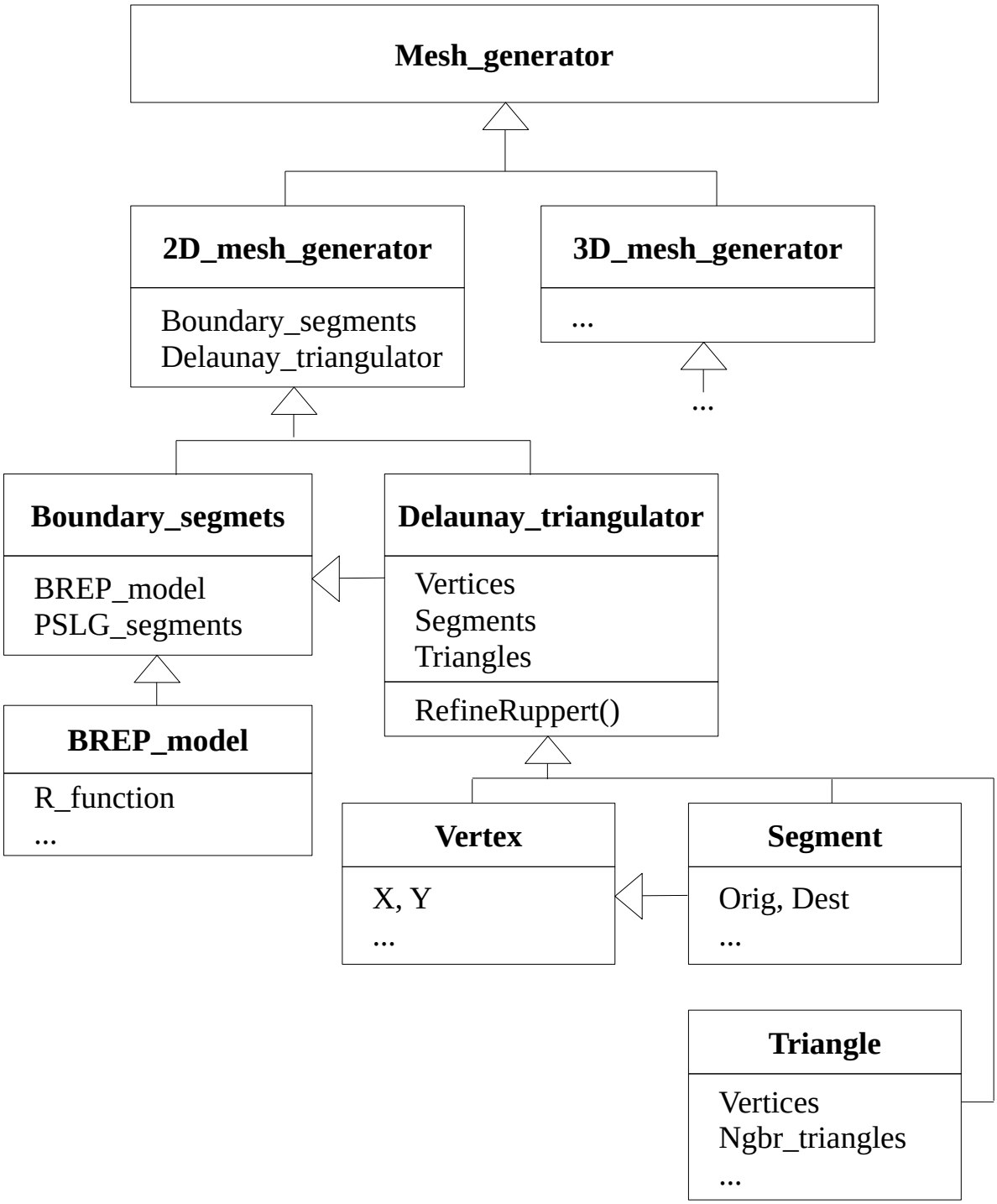

Рис. 2.7 – UML-діаграма препроцесора

Оскільки R-функція є неявною, то найбільш складним етапом побудови дискретної моделі функціонально заданої геометричної області є пошук множини точок, що належать границі області. Ефективним способом попереднього пошуку множини таких точок є застосування алгоритму Marching cubes [61] (Marching squares [62] у двовимірному випадку). За допомогою цього алгоритму можна

швидко знайти попередню апроксимацію границі області у вигляді набору граничних сегментів полігональної форми (відрізків у двовимірному випадку).

Алгоритм пошуку границі двовимірної області, заданої R-функцією, із застосуванням алгоритму Marching squares можна описати за допомогою псевдокоду [63] наступним чином:

Алгоритм *Пошук граничної апроксимації двовимірної геометричної області*

procedure *FindPSLG*(*RFunction*, *SuperSquare*, *N*)

PSLG – шуканий вектор граничних сегментів

$SuperSquare = \left(X_{min}, Y_{min}, X_{max}, Y_{max} \right)$ – координати суперобласті (зони пошуку)

$N = \left(N_x, N_y \right)$ – кількість кроків вздовж осей координат

begin

$H_x = \left(X_{max} - X_{min} \right) / N_x$

$H_y = \left(Y_{max} - Y_{min} \right) / N_y$

while $i \in \left[0, N_x - 1 \right]$ **do**

$x_0 = X_{min} + i \cdot H_x$

$x_1 = X_{min} + \left(i + 1 \right) \cdot H_x$

while $j \in \left[0, N_y - 1 \right]$ **do**

$y_0 = Y_{min} + j \cdot H_y$

$y_1 = Y_{min} + \left(j + 1 \right) \cdot H_y$

$square = \{ \left(x_0, y_0 \right), \left(x_1, y_1 \right) \}$

$\text{check} \left(square \right) \rightarrow PSLG$

end while

end while

end procedure

Тут процедура *check*() реалізує пошук відповідного шаблону (рис. 2.8) для поточного прямокутника, координати лівого нижнього та правого верхнього кута якого визначаються значеннями (x_0, y_0) та (x_1, y_1) (відповідно).

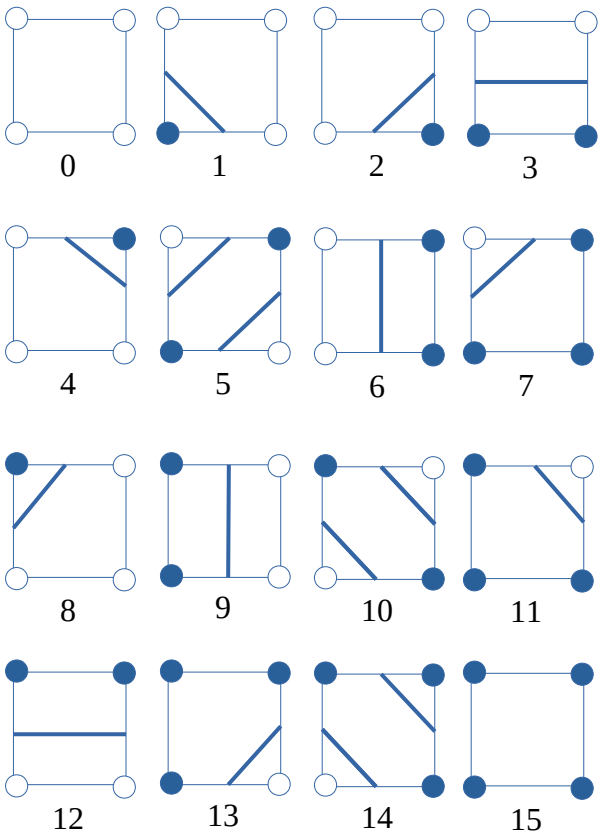

Рис. 2.8 – Шаблони пошуку граничних сегментів

Найбільш складною проблемою після отримання попередньої апроксимації границі області є пошук пропущених особливих точок (рис. 2.9).

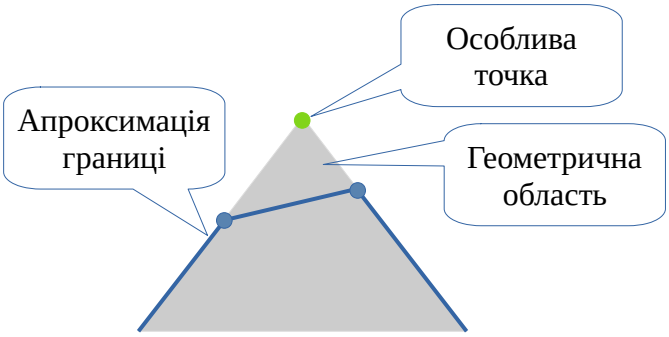

Рис. 2.9 – Пропущена особлива точка границі області

Для уточнення попередньої апроксимації границі пропонується наступний рекурсивний алгоритм (на прикладі двовимірного випадку). Відшукуються граничні сегменти, кут між якими більше певного критерію. Після чого будується нормаль до центру сегменту і знаходиться точка її перетину і границі області (рис. 2.10). Алгоритм повторюється до тих пір, доки кут між сусідніми граничними сегментами перестане змінюватися.

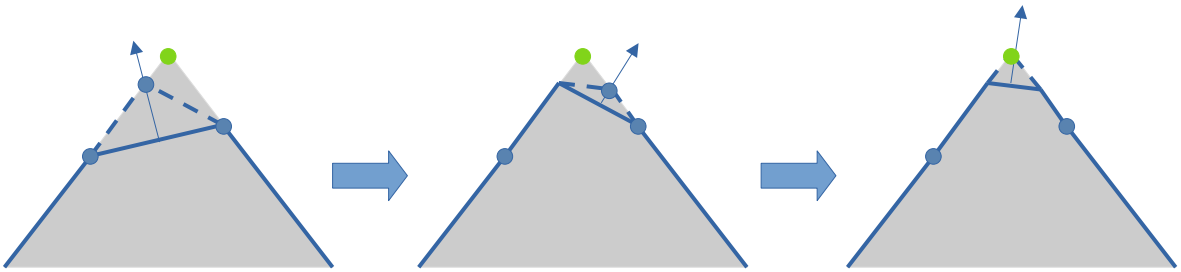

Рис. 2.10 – Рекурсивний алгоритм пошуку особливих точок

У тривимірному випадку пошук особливих точок границі області набагато складніший у реалізації, але концептуально збігається з наведеним алгоритмом.

2.3 Архітектура процесора

Головною задачею будь-якого процесора FEA-системи є безпосереднє виконання всіх необхідних розрахунків згідно з методом скінченних елементів. Безпосередньо алгоритм залежить від типу задачі (статика, динаміка, фізична нелінійність тощо), обраного скінченного елементу (двовимірні, тривимірні елементи, пластини, оболонки і т. п.). В самому загальному вигляді схему скінченно-елементного розрахунку можна представити наступним чином (рис. 2.11).

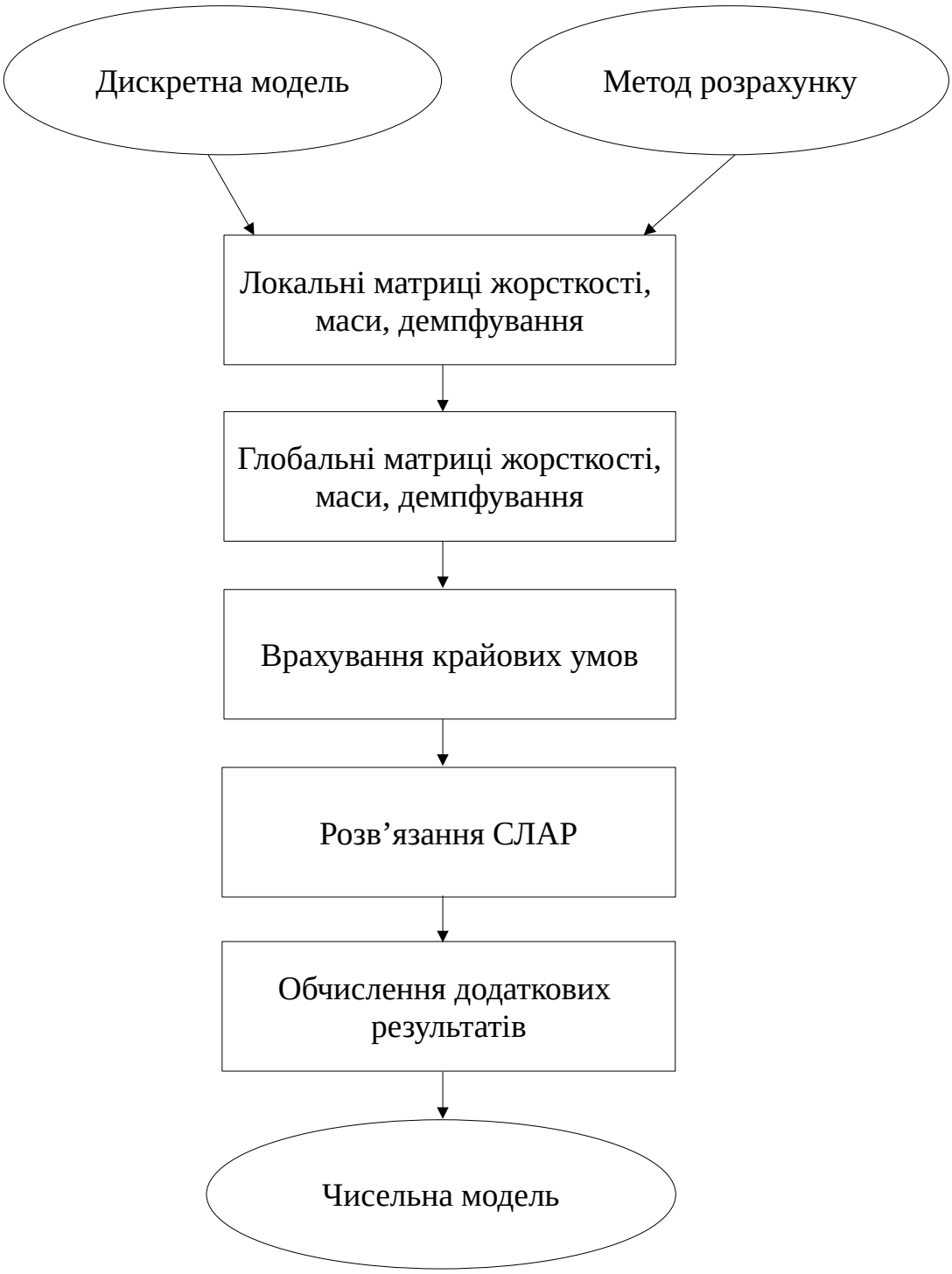

Рис. 2.11 – Типова схема скінченно-елементного розрахунку

Вхідними параметрами розрахунку є дискретна модель (отримана на етапі застосування препроцесора) і в певному сенсі числова схема (метод розрахунку). В залежності від обраної схеми розрахунку реалізується певний алгоритм побудови локальних матриць жорсткості, маси та демпфування, які будуються для

кожного СЕ вихідної дискретної моделі. Всі інші етапи розрахунку більш-менш стандартні.

Алгоритм розрахунку типової задачі статики в пружній постановці можна описати за допомогою псевдокоду наступним чином:

Алгоритм *Скінченно-елементний розрахунок задачі статики*
procedure *CalcStaticProblem*(*Mesh, Algorithm, BoundaryConditions*)
Mesh **–** дискретна модель
Algorithm – обчислювальна схема побудови локальної матриці жорсткості
BoundaryConditions – граничні умови
begin
GSM **–** глобальна матриця жорсткості
GLV **–** глобальний вектор навантажень
Result **–** глобальний вектор результатів
LSM **–** локальна матриця жорсткості скінченного елемента
LLV **–** локальний вектор навантажень
N **–** кількість скінченних елементів
while $i \in [0, N]$ **do**
$Algorithm \rightarrow LSM, LLV$
$GSM \leftarrow LSM$
$GLV \leftarrow LLV$
end while
UseBoundaryCondition(GSM, GLV, BoundaryConditions)
$SolveLinearSystem(GSM, GLV) \rightarrow Result$
end procedure

Тут процедура *UseBoundaryCondition()* реалізує застосування граничних умов до матриці жорсткості та вектора-стовпця правої частини, а *SolveLinearSystem()* – розв’язання відповідної СЛАР.

Аналогічний алгоритм розрахунку пружної задачі в динамічній постановці можна описати за допомогою псевдокоду наступним чином:

Алгоритм *Скінченно-елементний розрахунок задачі динаміки*

procedure *CalcDynamicProblem*(*Mesh*, *Alg1*, *Alg2*, *BoundaryConditions*)

Mesh **–** дискретна модель

Alg1 – обчислювальна схема побудови локальних матриць жорсткості, маси і демпфування

Alg2 – алгоритм дискретизації у часі

BoundaryConditions – крайові умови

begin

GM **–** глобальна матриця коефіцієнтів СЛАР

GSM **–** глобальна матриця жорсткості

GMM **–** глобальна матриця маси

GDM **–** глобальна матриця демпфування

GLV **–** глобальний вектор навантажень

Result **–** глобальний вектор результатів

LSM **–** локальна матриця жорсткості

LMM **–** локальна матриця маси

LDM **–** локальна матриця демпфування

LLV **–** локальний вектор навантажень

N **–** кількість скінченних елементів

t_0, t_n, t_h **–** початковий та кінцевий момент часу, крок

while $i \in [0, N]$ **do**

$Alg1 \rightarrow LSM, LMM, LDM, LLV$

$GSM \leftarrow LSM$

$GMM \leftarrow LMM$

$GDM \leftarrow LDM$

$GLV \leftarrow LLV$

end while

$t = t_0$

while $t \le t_0$ **do**

$Alg2(t, BoundaryConditions) \rightarrow GM, GV$

$SolveLinearSystem(GM, GV) \rightarrow Result$

$t = t + t_h$

end while

end procedure

Аналіз алгоритмів розв’язання крайових задач із застосуванням МСЕ дозволив прийти висновку, що їх програмна реалізація потребує, по-перше, реалізації алгоритму побудови локальних матриць СЕ (жорсткості, маси, демпфування), а, по-друге, при необхідності (в залежності від типу задачі) – безпосереднього алгоритму розв’язання задачі: дискретизації у часі, лінеаризації нелінійної задачі і т.п.

Базовий абстрактний клас FEM, що описує узагальнену реалізацію метода скінченних елементів можна, наприклад, представити таким чином (рис. 2.12). Основними властивостями цього класу є:

– скінченно-елементна сітка (Mesh), що описує область розрахунку;

– параметри розрахунку (Params): модуль Юнга, коефіцієнт Пуассона, крайові умови тощо;

– скінченний елемент (FE), екземпляр класу, що інкапсулює вживаний для розв’язання конкретної задачі елемент (дво- чи тривимірний, оболонковий, серендиповий [64] і т.п.).

Крім властивостей цей клас містить низку методів, головними серед яких є абстрактні Calc_problem() та Calc_results(), які реалізуються в класах-нащадках і функціонал яких залежить від типу задачі, що розв’язується.

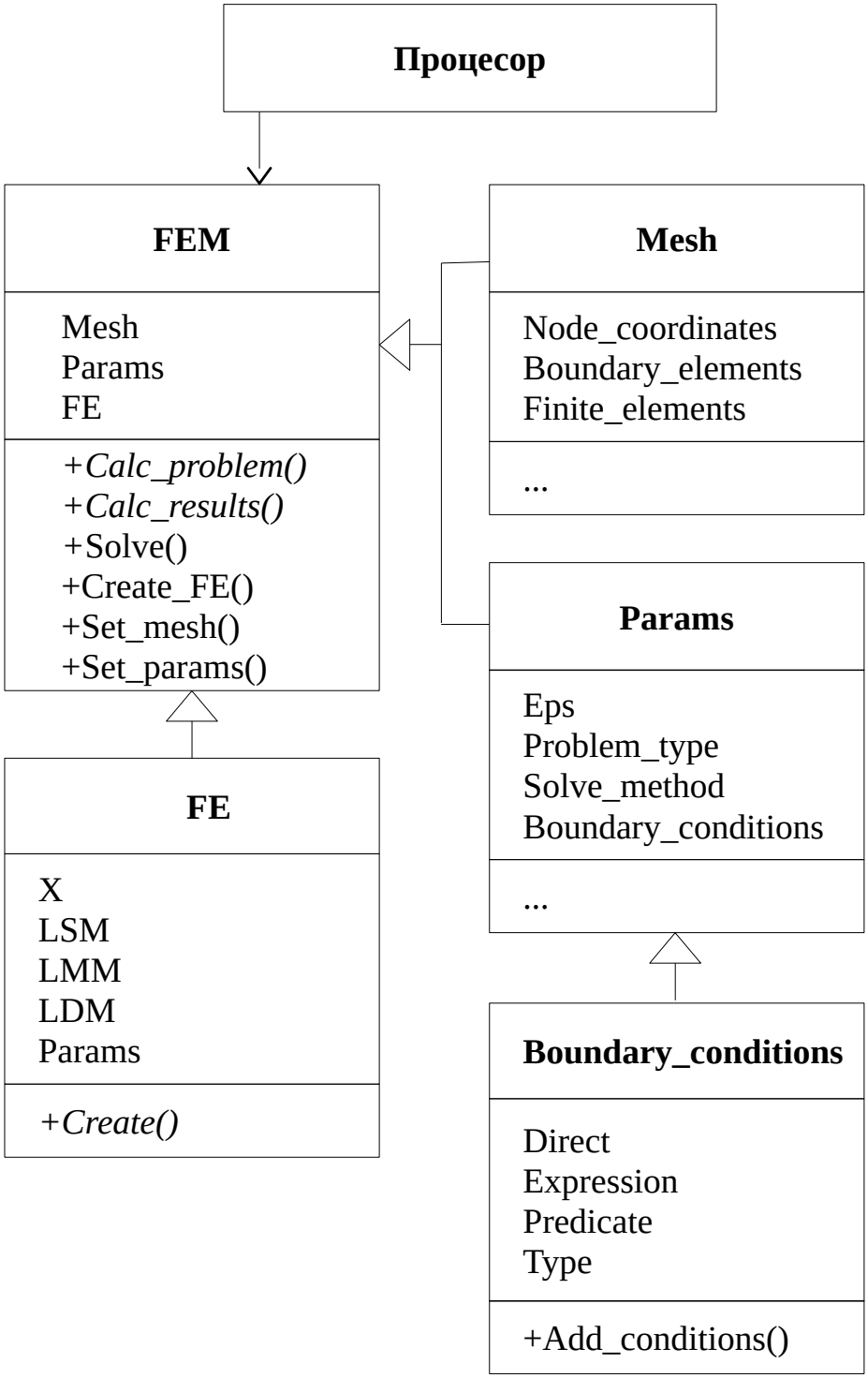

Рис. 2.12 – Базовий абстрактний клас, що описує МСЕ

Ієрархія класів, що базуються на FEM, може мати такий вигляд (рис. 2.13). Ця ієрархія використовується для реалізації кожного специфічного розрахунку (статика, динаміка, фізична або геометрична нелінійність тощо).

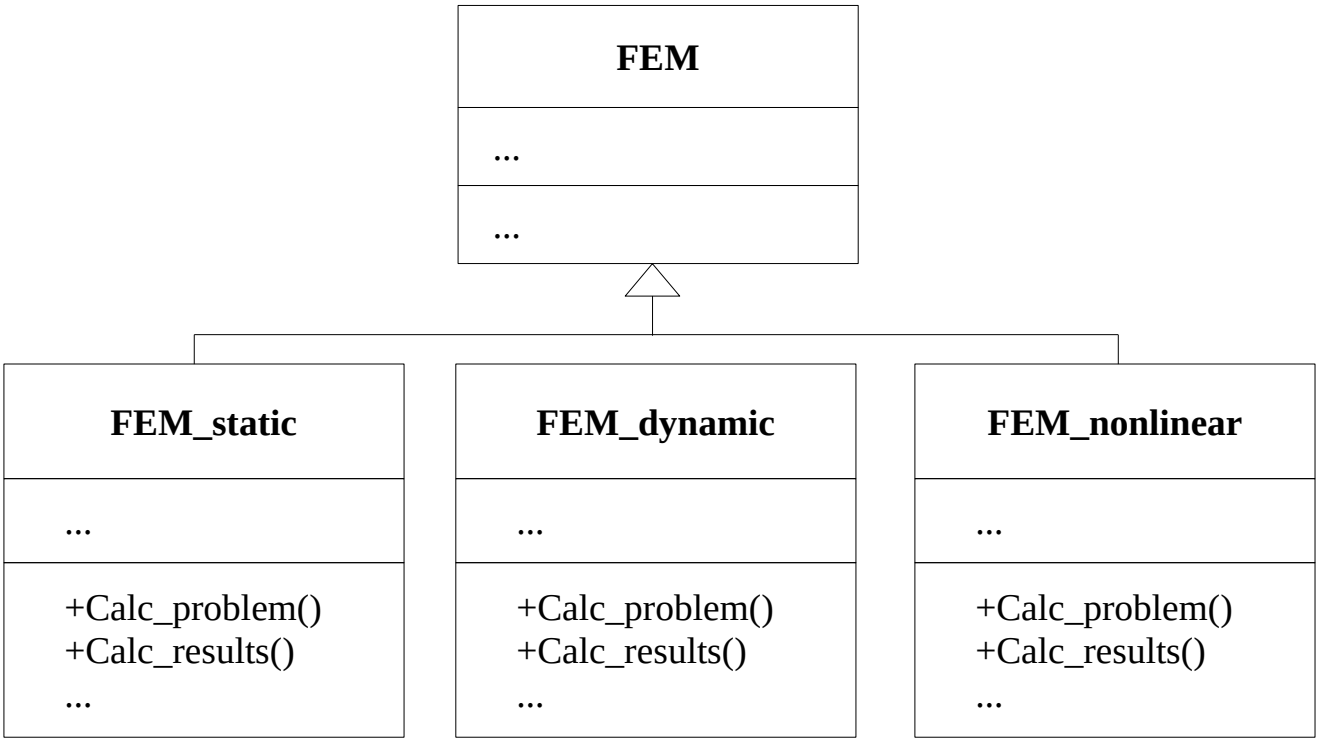

Рис. 2.13 – Ієрархія класів скінченно-елементного розрахунку

Всі ці класи успадковують загальну скінченно-елементну функціональність від базового абстрактного класу FEM, але в них безпосередньо визначаються методи Calc_problem() і Calc_results(), в яких безпосередньо реалізується відповідна обчислювальна схема кожного конкретного методу.

Детально реалізація цих класів буде розглянута у наступному розділу.

2.4 Архітектура постпроцесора

Аналіз результатів чисельного розрахунку із застосуванням МСЕ на практиці ускладнюється двома основними чинниками:

– значними обсягами числових даних, які отримуються при розв’язанні складних задач у тривимірній постановці (особливо в динаміці), що істотно ускладнює процедуру їх аналізу;

– необхідністю розрахунку додаткових даних на основі раніше отриманих результатів (наприклад, обчислення інтенсивності напружень на основі раніше розрахованих переміщень в задачах механіки).

Для автоматизації вирішення цих завдань на практиці застосовують спеціалізоване програмне забезпечення, яке по аналогії з препроцесором і процесором прийнято називати постпроцесором. За допомогою постпроцесорів частіше за все чисельні результати розрахунку певним чином візуалізуються [65]. У більшості випадків з цією метою здійснюється побудова відповідних графіків, ліній рівня або кольорових картин розподілу величини, що досліджується, по області розрахунку (рис. 2.14).

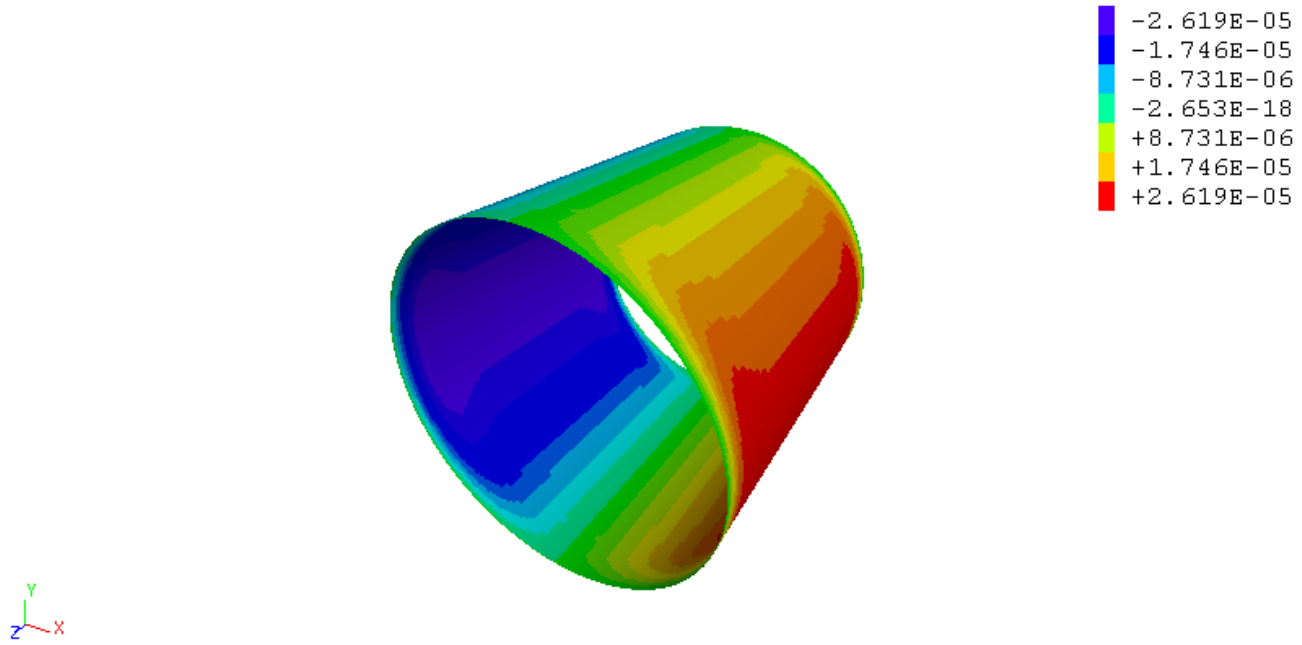

Рис. 2.14 – Приклад візуалізації розподілу переміщень по області розрахунку

Оскільки для створення такого зображення достатньо реалізувати візуалізацію лише поверхні області розрахунку, яка складається при застосуванні МСЕ з певної кількості граничних елементів (ГЕ), то достатньо буде створити алгоритм візуалізації розподілу функції по окремому трикутнику, так як ГЕ будь-

якої форми може бути представлений у вигляді кінцевої множини трикутних елементів.

Візуалізація кожного трикутника відбувається у відповідності до наступного алгоритму. Обчислюється кількість ліній рівня (кольорових переходів) між максимальним і мінімальним вузловим значенням функції, після чого вихідний ГЕ розбивається на деяку кількість трикутників, які мають однаковий колір, тобто у їх вершинах досліджувана функція F має однакові значення, або такі, що відповідають одному й тому самому індексу кольору (рис. 2.15).

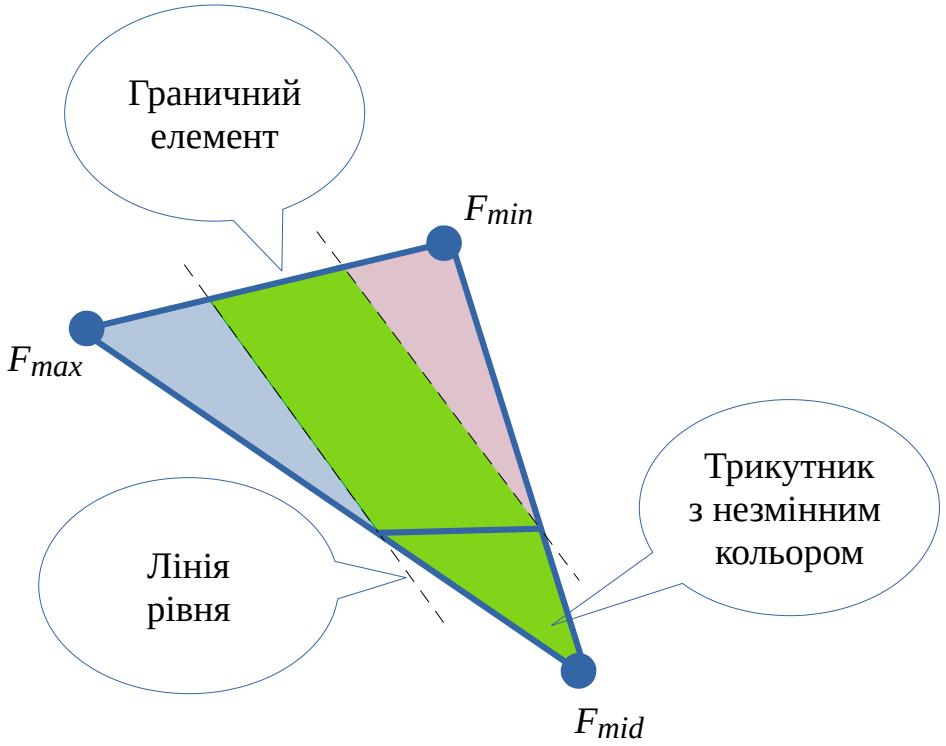

Рис. 2.15 – Схема візуалізації граничного елемента

Оскільки сучасні відеокарти реалізують швидке відтворення великої кількості трикутників, то візуалізація розподілу певної функції по складній тривимірній області буде відбуватися з прийнятною для користувача швидкістю, особливо при використанні бібліотеки OpenGL [66, 67], яка підтримує графічну акселерацію.

Алгоритм візуалізації трикутника із застосуванням псевдокоду може бути описана наступним чином.

Алгоритм *Візуалізація трикутника*

procedure *PreVisualizationTriangle* (*Coordinate*, *F*)

$F=\left[F_{min}, F_{mid}, F_{max}\right]$ – вузлові значення функції, що аналізується

$Coordinate=\left[x_0, y_0, z_0, x_1, y_1, z_1, x_2, y_2, z_2\right]$ – координати вершин

V^1, V^2 – допоміжні вектори для зберігання координат трикутників

begin

$[C_{min}, C_{mid}, C_{max}]=GetColor\left(F_{min}, F_{mid}, F_{max}\right)$

if $C_{min}=C_{mid}$ **and** $C_{mid}=C_{max}$ **then**

VisualizationTriangle(*Coordinate*, C_{min})

return

end if

$n=C_{max}-C_{min}+1$

$[H_x, H_y, H_z]=[\left(x_{max}-x_{min}\right)/n, \left(y_{max}-y_{min}\right)/n, \left(z_{max}-z_{min}\right)/n]$

$H_c=\left(C_{max}-C_{min}\right)/n$

while $i \in \left[0, n-1\right]$ **do**

$V^1 \leftarrow [x_0+i\cdot H_x, y_0+i\cdot H_y, z_0+i\cdot H_z, C_{min}+i\cdot H_c]$

end while

$V^1 \leftarrow [x_2, y_2, z_2, C_{max}]$

$n=C_{mid}-C_{min}+1$

$[H_x, H_y, H_z]=[\left(x_{max}-x_{min}\right)/n, \left(y_{max}-y_{min}\right)/n, \left(z_{max}-z_{min}\right)/n]$

$H_c=\left(C_{mid}-C_{min}\right)/n$

while $i \in \left[1, n-1\right]$ **do**

$V^2 \leftarrow [x_0+i\cdot H_x, y_0+i\cdot H_y, z_0+i\cdot H_z, C_{min}+i\cdot H_c]$

end while

$V^2 \leftarrow [x_1, y_1, z_1, C_{mid}]$

$n = C_{max} - C_{mid} + 1$

$[h_x, h_y, h_z] = [(x_{max} - x_{mid})/n, (y_{max} - y_{mid})/n, (z_{max} - z_{mid})/n]$

$h_c = (C_{max} - C_{min})/n$

while $i \in [1, n-1]$ **do**

$V^2 \leftarrow [x_1 + i \cdot h_x, y_1 + i \cdot h_y, z_1 + i \cdot h_z, C_{mid} + i \cdot h_c])$

end while

while $i \in [0, len(V^1) - 2]$ **do**

if $i < len(V^2)$ **then**

$Coordinate = [V^1_{i+1,0}, V^1_{i+1,1}, V^1_{i+1,2}, V^1_{i,0}, V^1_{i,1}, V^1_{i,2}, V^2_{i,0}, V^2_{i,1}, V^1_{i,2}]$

VisualizationTriangle(*Coordinate*, $V^1_{i,3}$)

end if

if $i + 1 < len(V^2)$ **then**

$Coordinate = [V^1_{i+1,0}, V^1_{i+1,1}, V^1_{i+1,2}, V^2_{i,0}, V^2_{i,1}, V^2_{i,2}, V^2_{i+1,0}, V^2_{i+1,1}, V^1_{i+1,2}]$

VisualizationTriangle(*Coordinate*, $V^2_{i+1,3}$)

end if

end while

end procedure

В даному алгоритмі процедура *VisualizationTriangle*() реалізує безпосередню обробку однокольорового трикутника (наприклад, його візуалізацію або збереження у файлі). Процедура *GetColor*() повертає параметри кольору, якому відповідає задане значення функції, що аналізується.

Для розрахунку додаткових результатів в загальному випадку слід реалізувати певний механізм запиту у користувача співвідношень (формули), які описують необхідну додаткову функцію. Наприклад, якщо відомі незалежні компоненти тензору напружень [68] σ_{xx}, σ_{yy}, σ_{zz}, σ_{xy}, σ_{xz}, σ_{yz}, то обчислити

інтенсивність деформацій, яка широко використовується в механіці руйнування [69], можна за допомогою наступного співвідношення:

$$\sigma_i = 0.71\sqrt{(\sigma_{xx}-\sigma_{yy})^2+(\sigma_{xx}-\sigma_{zz})^2+(\sigma_{yy}-\sigma_{zz})^2+6(\sigma_{xy}^2+\sigma_{xz}^2+\sigma_{yz}^2)}.$$

У загальному випадку для обчислення таких співвідношень, заданих користувачем за допомогою деякого формального способу, наприклад, DSL (domain-specific language) [70], необхідно реалізовувати відповідний транслятор [71]. В даному випадку, оскільки для реалізації системи обрано інтерпретовану мову Python [], користувач може задати необхідну йому формулу в термінах безпосередньо цієї мови.

Отже постпроцесор системи скінченно-елементного аналізу може мати наступну типову відкриту архітектуру (рис. 2.16).

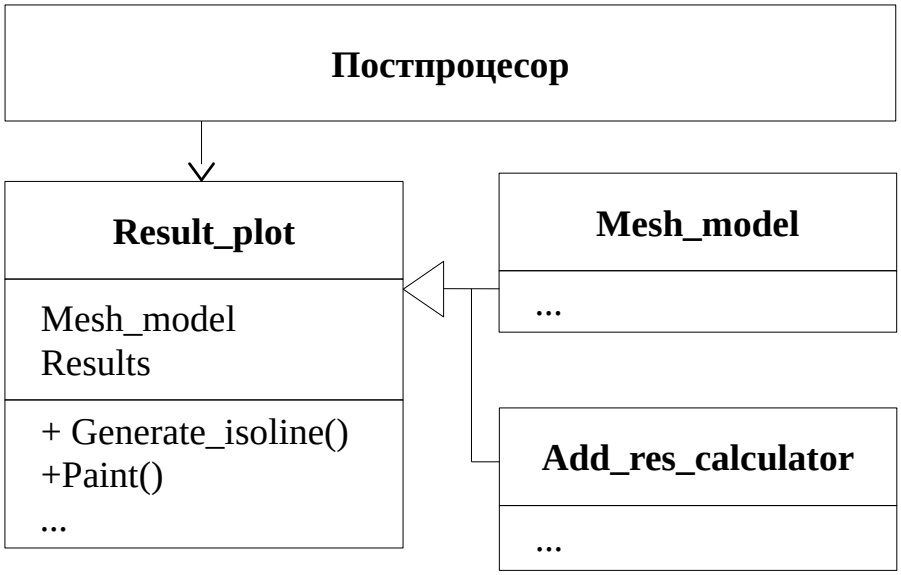

Рис. 2.16 – Загальна архітектура постпроцесора

Тут клас Result_plot реалізує візуалізацію результатів розрахунку, отриманих при застосуванні МСЕ. Його основними компонентами є дискретна модель (Mesh_model) конструкції, що розраховується, і масив результатів розрахунку (Results). В класі реалізується процедура Generate_isoline() побудови ліній рівня

(однокольорових трикутників), а також процедура безпосередньої візуалізації результатів – Paint().

Висновки до розділу 2

Підсумовуючи результати, які були отримані у другому розділі, можна зробити висновок, що для реалізації повнофункціональної об’єктно-орієнтованої бібліотеки скінченно-елементного розрахунку достатньо спроєктувати три набори взаємопов’язаних класів:

1) препроцесора – опису геометричної моделі (CSG, BREP тощо) вихідного об’єкту розрахунку, а також його дискретної скінченно-елементної моделі;

2) процесора – ієрархічної множини, що описують базовий абстрактний клас скінченно-елементного розрахунку, і його нащадків – класів, які реалізують кожен конкретний вид розрахунку (статика, динаміка, нелінійність тощо);

3) постпроцесора – множини класів, призначених для зберігання чисельних результатів розрахунку, прив’язаних до дискретної моделі об’єкта, а також автоматизації розрахунку додаткових функцій і візуалізації отриманих даних.

Основні наукові і практичні результати даного розділу опубліковано в роботах [46, 47, 73].

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ВІДКРИТОЇ ОБ’ЄКТНО-ОРІЄНТОВАНОЇ АРХІТЕКТУРА СКІНЧЕННО-ЕЛЕМЕНТНОГО МОДЕЛЮВАННЯ

3.1 Основні патерни розробки наукового програмного забезпечення

При промисловій розробці сучасного ПЗ все частіше застосовуються патерни або шаблони програмування – типові способи вирішення проблем, що часто виникають під час проектування програм [74]. Патерни, на відміну від бібліотек, не можуть бути напряму застосовані, оскільки вони не є програмним кодом. Це лише загальні принципи розв’язання типових проблем, що виникають при розробці складного ПЗ.

Патерни проєктування є розвитком концепції об’єктно-орієнтованого програмування (ООП) [75]. Опис патерну в загальному випадку складається з наступних компонентів:

– опис проблеми, для вирішення якої він призначений;

– опис переваг його застосування;

– структура та ієрархія класів, які забезпечують вирішення проблеми;

– конкретний приклад застосування шаблону;

– опис можливих особливостей та проблем, що можуть виникнути при реалізації патерну;

– опис зв’язків з іншими патернами проєктування [74].

Застосування патернів при розробці ПЗ дозволяє отримати низку переваг:

– уніфікація початкового коду та термінології, що дає можливість покращити рівень комунікації з партерами та спростити супровід ПЗ;

– використання вже апробованих іншими програмістами підходів та рішень, що дає змогу прискорити розробку та зменшити час налагодження програм.

Патерни проєктування розрізняються як за складністю їх практичного використання, так і ступенем охоплення проєктованого ПЗ.

Умовно всі такі шаблони можна поділити на дві великі групи: низькорівневі та високорівневі. Низькорівневі патерни (так звані “ідіоми” проєктування) орієнтовані на використання конкретної мови програмування [74, 76]. Високорівневі (“архітектурні”) патерни не залежать від мови програмування і можуть бути використані спільно з будь-якою об’єктно-орієнтованою мовою програмування.

Проте більш зручною є класифікація шаблонів проєктування у відповідності до їх призначення:

– твірні – використовуються для узагальнення процесу створення нових екземплярів класів (об’єктів);

– структурні – описують процедуру створення з наявних класів похідних або більших за розмірами структур даних;

– поведінкові – описують способи взаємодії між екземплярами класів.

Найбільш поширеними твірними шаблонами, які забезпечують ефективне і безпечне створення нових екземплярів класів, є:

– Factory Method – описує загальну процедуру створення екземплярів батьківського класу, яка дозволяє дочірнім класам змінювати типи їх об’єктів при створенні;

– Abstract Factory – описує процедуру створення сімейства класів без прив’язки їх до конкретних типів;

– Builder – описує покрокову процедуру створення складних екземплярів класів із використанням уніфікованого початкового коду для отримання різних реалізацій об’єктів;

– Singleton – описує процедуру створення унікального екземпляру класу та механізму доступу до нього;

– Prototype – описує алгоритм створення копій поточного екземпляру без урахування нюансів його реалізації.

Структурні шаблони описують процедуру створення зручних в реалізації та підтримці ієрархій класів. Самими поширеними серед них є такі:

– Adapter – реалізує можливість взаємодії між об’єктами з різними інтерфейсами;

– Bridge – реалізує можливість поділу класів на дві окремі ієрархічні категорії: абстракцію та реалізацію, що дає змогу незалежної модифікації початкового коду в кожній гілці;

– Composite – реалізує можливість об’єднання класів в єдину структуру;

– Decorator – забезпечує можливість динамічного додавання до класу нової функціональності за рахунок застосування спеціалізованих класів-“обгорток” [74, 77, 78].

Шаблони поведінки описують реалізацію ефективної та безпечної комунікації між екземплярами класів. Найбільш поширеними серед них є:

– Iterator – описує процедуру обходу певної колекції об’єктів без врахування їхньої внутрішньої структури (дуже популярний на сьогодні спосіб ітерування по колекціям, реалізований, наприклад, в бібліотеці STL [79]);

– Mediator – дозволяє зменшити кількість зв’язків між класами за рахунок створення спеціального класу-посередника;

– Observer – реалізує можливість певним об’єктам стежити за поведінкою та станом інших та реагувати на необхідні події;

– Strategy – описує реалізацію групування схожих алгоритмів в деякому класі, що дає можливість динамічно замінювати один алгоритм на інший;

– Visitor – дозволяє виконати додавання до програми нових операцій без зміни класів, над екземплярами яких вони можуть виконуватися;

– State – описує реалізацію можливості екземплярам класів змінювати їх поведінку в контексті їхнього поточного стану [74, 77, 78].

Таким чином, використання патернів проєктування дозволяє не тільки збільшити продуктивність праці програмістів при розробці ПЗ, а й зменшити час на його супровід. Де-факто, використання патернів проєктування на сьогодні є

стандартом розробки програм в IT-галузі. Аналіз наявних популярних шаблонів показав, що вони в основному базуються на парадигмі ООП [75]. Саме тому в даному розділі буде розглянуто конструювання ієрархічної системи класів для реалізації бібліотеки скінченно-елементного аналізу.

3.2 Проєктування класів для інкапсуляції різних типів скінченних елементів

Одними з найбільш фундаментальних елементів будь якої програми скінченно-елементного аналізу є структури даних, які описують СЕ. Оскільки типів СЕ існує достатньо багато (одно-, двох- та тривимірні, СЕ оболонок та пластин, серендипові СЕ і т.п. [1, 16, 17, 23, 64]), то, в першу чергу, для практичної реалізації МСЕ із використанням ООП слід подбати про розробку ієрархічної системи класів, що інкапсулюють різні типи СЕ.

Оскільки будь-яка функція $F(x,y,z)$ на СЕ може бути апроксимована наступним чином:

$$F(x,y,z)=\sum_{i=1}^{n} F_i N_i(x,y,z),$$

де F_i – відомі вузлові значення функції $F(x,y,z)$ у вершинах СЕ;

$N_i(x,y,z)$ – функції форми СЕ [1];

n – кількість вершин СЕ.

Функції форми стандартних СЕ (наприклад, трикутників або тетраедрів) знаходяться з наступного співвідношення:

$$N_i(x_j, y_j, z_j) \equiv \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases} \quad (3.1)$$

Тобто функція форми $N_i(x, y, z)$ дорівнює одиниці у i-ому вузлі елемента і нулю – у всіх інших.

Вид функції форми, як витікає з її назви, в першу чергу, залежить від форми елемента.

Розглянемо вид функцій форм найбільш поширених на практиці стандартних СЕ. Так, функція форми двовимірного трикутного СЕ (рис. 3.1) має наступний вигляд:

$$N_i(x, y) = c_1 + c_2 x + c_3 y, \quad (3.2)$$

де c_i $(i = \overline{1..3})$ – дійсні коефіцієнти, що залежать від координат вершин (вузлів) трикутника, і визначаються зі співвідношення (3.1).

Аналогічно, функція форми чотирикутного двовимірного елемента (рис. 3.2) описується наступною формулою:

$$N_i(x, y) = c_1 + c_2 x + c_3 y + c_4 xy. \quad (3.3)$$

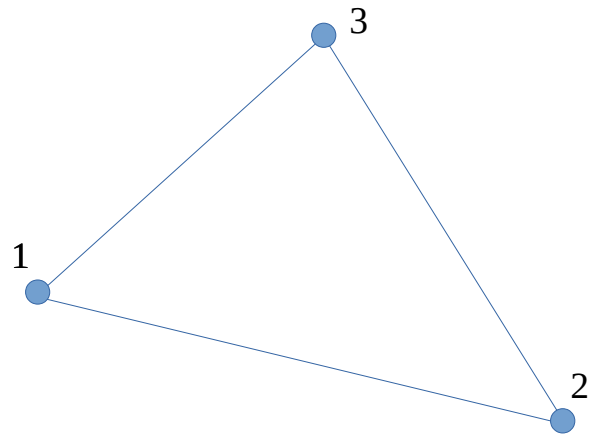

Рис. 3.1 – Трикутний скінченний елемент

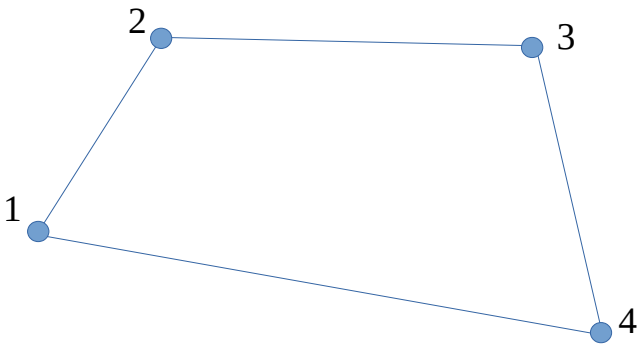

Рис. 3.2 – Чотирикутний елемент

Функція форми тривимірного тетраедрального СЕ (рис. 3.3) має наступний вигляд:

$$N_i(x,y)=c_1+c_2 x+c_3 y+c_4 xy. \qquad (3.4)$$

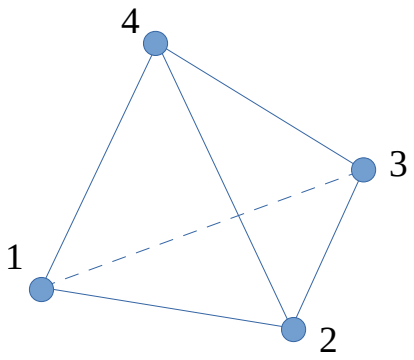

Рис. 3.3 – Тетраедральний елемент

Тут слід підкреслити, що чотирикутний двовимірний і тетраедральний тривимірний елементи мають однакову кількість коефіцієнтів, що потребує певної обробки при програмній реалізації для запобігання виникненню плутанини.

Ще одним поширеним СЕ, який часто використовується на практиці, є тривимірний кубічний (гексаедральний) елемент (рис. 3.4). Він описується наступною функцією форми:

$$N_i(x,y,z)=c_1+c_2x+c_3y+c_4z+c_5xy+c_6xz+c_7yz+c_8xyz. \quad (3.5)$$

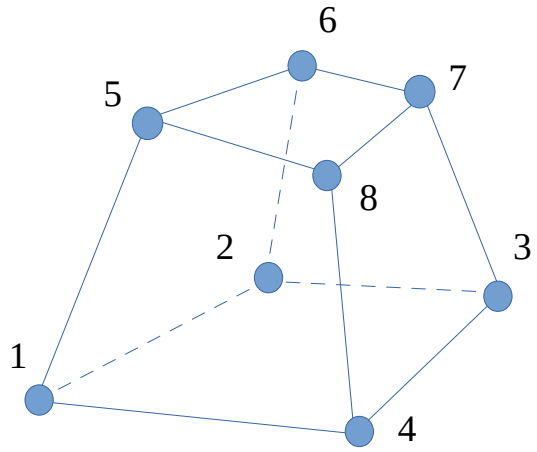

Рис. 3.4 – Гексаедральний елемент

З аналізу співвідношень (3.2)-(3.5) можна зробити висновок, що для узагальненого опису стандартних СЕ слід реалізувати структуру даних, яка б дозволяла приймати в якості параметрів координати вузлів СЕ, на їх основі розраховувати коефіцієнти функцій форми СЕ, а потім обчислювати локальні матриці жорсткості, маси та демпфування. Для цього в структурі, що описує СЕ, слід передбачити зберігання таких параметрів, як кількість вузлів елементу та кількість ступенів свободи у кожному вузлі, їх пружні та фізичні характеристики і т.п.

Для врахування різноманіття СЕ в бібліотеці PyFEM було реалізовано ієрархічну структуру класів (рис. 3.5). Базовим є абстрактний клас TFE, що описує найбільш фундаментальні властивості ізопараметричного скінченного елемента [1]: кількість вузлів (розмірність); площа перерізу для одновимірних, товщина для двовимірних або об’єм для тривимірних елементів; пружні властивості; температура; коефіцієнт теплового розширення; густина; коефіцієнт демпфування; локальні матриці жорсткості, маси та демпфування, параметри квадратур для чисельного інтегрування тощо.

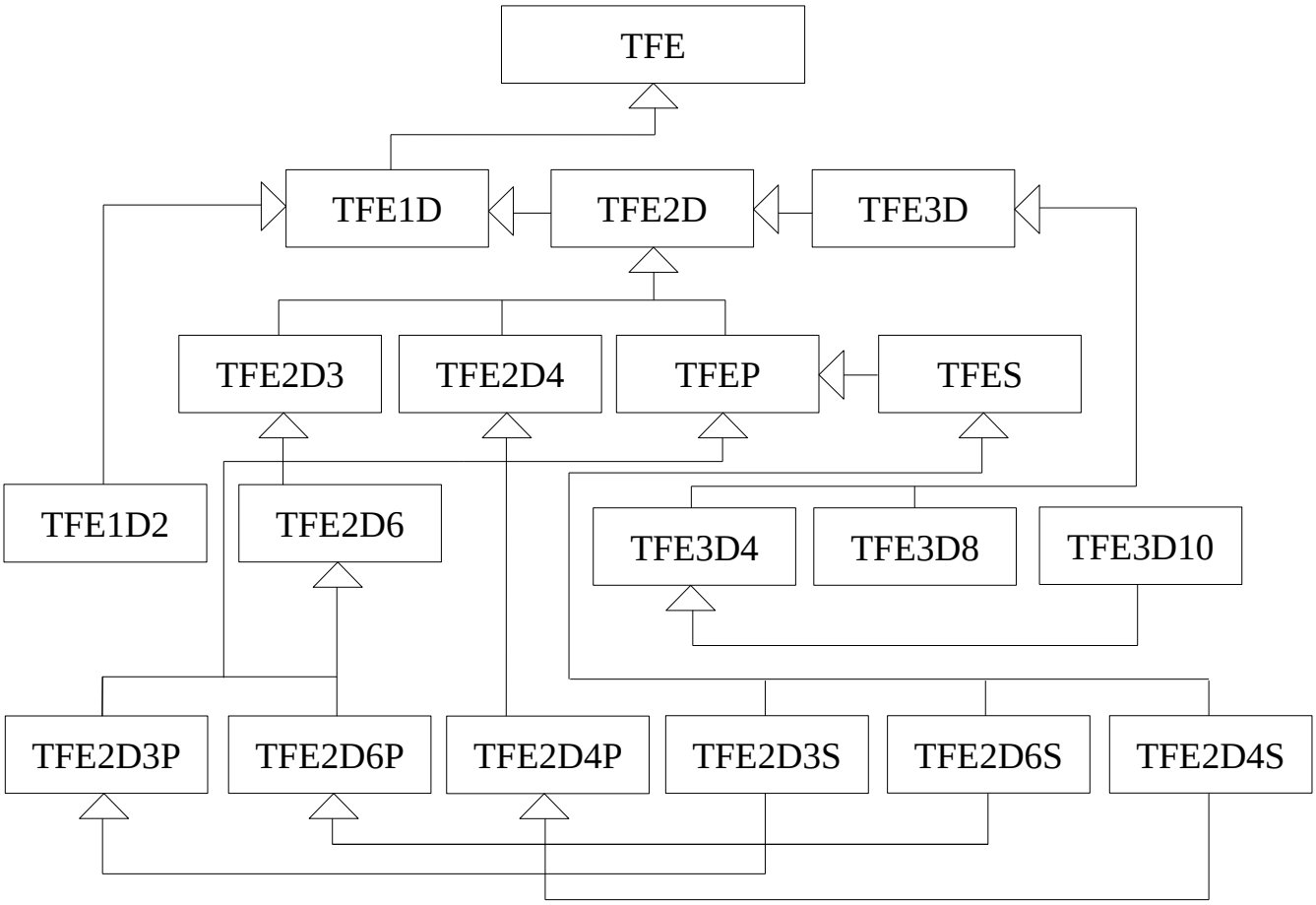

Рис. 3.5 – Ієрархія класів, що описують скінченні елементи

Клас TFE є абстрактним, оскільки в ньому визначено, але не реалізовано процедуру побудови локальних матриць, оскільки вона залежить від типу конкретного елемента.

Опис класу на мові Python має наступний вигляд.

```
# Абстрактний базовий клас, що описує ізопараметричний скінчений елемент
class TFE:
    def __init__(self):
        self.size = 0           # Розмірність елемента
        self.freedom = 0        # Кількість ступенів свободи
        self.e = []             # Модуль Юнга
        self.m = []             # Коефіцієнт Пуассона
        self.thickness = 1      # Площа перетину (1d), товщина (2d) або об’єм (3d)
        self.alpha = 0          # Параметр температурного розширення
```

```
    self.dT = 0             # Різниця температур
    self.density = 0        # Густина матеріалу
    self.damping = 0        # Коефіцієнт демпфування
    self.x = []             # Координати вузлів елемента
    self.K = []             # Локальна матриця жорсткості
    self.M = []             # ... маси
    self.C = []             # ... демпфування
    self.load = []          # Локальний вектор навантаження
    self.a = []             # Коефіцієнти функцій форми
    self._xi = []           # Параметри квадратур для чисельного інтегрування
    self._eta = []          # ...
    self._psi = []          # ...
    self._w = []            # ...
```

Реалізація методів класу TFE виконується наступним чином.

```
# Визначення параметрів
def set_young_modulus(self, e):
    self.e = e
def set_poisson_ratio(self, m):
    self.m = m
def set_thickness(self, t):
    self.thickness = t
def set_damping(self, d):
    self.damping = d
def set_density(self, d):
    self.density = d
def set_temperature(self, dt):
    self.dT = dt
def set_alpha(self, a):
    self.alpha = a
```

```
# Ініціалізація координат
def set_coord(self, x):
    self.x = array(x)
    self._create()

# Перевірка коректності параметрів елемента
def _check(self):
    if not len(self.e):
        raise TException('elasticity_err')

# Абстрактні методи класу
# Обчислення функцій форми
@abstractmethod
def _create(self):
    raise NotImplementedError('Method TFE._create() is pure virtual')

# Матриця пружних властивостей
@abstractmethod
def _elastic_matrix(self):
    raise NotImplementedError('Method TFE._elastic_matrix() is pure virtual')

# Обчислення стандартних результатів
@abstractmethod
def calc(self, u):
    raise NotImplementedError('Method TFE.calc() is pure virtual')

# Обчислення матриць жорсткості, маси та демпфування
@abstractmethod
def generate(self, is_static=True):
    raise NotImplementedError('Method TFE.generate() is pure virtual')
```

Похідні від TFE абстрактні класи TFE1D, TFE2D та TFE3D реалізують побудову локальних матриць для стандартних одно-, дво- та тривимірних

елементів. Ці реалізації не містять відповідних процедур побудови функцій форм скінченних елементів.

Реалізація абстрактного класу TFE1D для роботи з одновимірними СЕ має наступний вигляд.

```
# Абстрактний одновимірний елемент
class TFE1D(TFE):
    def __init__(self):
        super().__init__()
        self.freedom = 1

    # Узагальнена процедура обчислення стандартних результатів
    def calc(self, u):
            ...
    # Узагальнена процедура формування локальних матриць
    def generate(self, is_static=True):
            ...

     # Абстрактні методи реалізації функцій форми та їх похідних
    @abstractmethod
    def _shape(self, i, j):
        raise NotImplementedError('Method TFE1D._shape() is pure virtual')
    @abstractmethod
    def _shape_dxi(self, i):
        raise NotImplementedError('Method TFE1D._shape_dxi() is pure virtual')
    @abstractmethod
    def _dx(self, i, j):
        raise NotImplementedError('Method TFE1D._dx() is pure virtual')
```

Аналогічно абстрактний клас TFE2D для роботи з двовимірними елементами може бути реалізований наступним чином.

```
# Абстрактний ізопараметричний двовимірний скінченний елемент
class TFE2D(TFE1D):
    def __init__(self):
        super().__init__()
        self.freedom = 2
        ...
```

Так само клас TFE3D для роботи зі стандартними тривимірними елементами описується наступним чином.

```
# Абстрактний тривимірний скінченний елемент
class TFE3D(TFE2D):
    def __init__(self):
        super().__init__()
        self.freedom = 3
        ...
```

В цих класах виконана реалізація універсального метода generate(), який здійснює безпосередню побудову локальних матриць жорсткості, маси і демпфування для одно-, дво- та тривимірного випадку. А також метода calc(), який виконує обчислення стандартних результатів СЕ, таких, як компоненти тензорів деформації та напруження. Їх реалізація однакова для всіх типів СЕ відповідної розмірності. Обчислення ж функцій форми та їх похідних, які необхідні для побудови локальних матриць, реалізується у відповідних класах-нащадках від цих базових класів.

Для реалізації одновимірного СЕ в бібліотеці PyFEM розроблено клас TFE1D2, який інкапсулює ізопараметричний одновимірний лінійний двовузловий елемент. Його повний опис на мові Python має наступний вигляд.

```
# Лінійний (двовузловий) одновимірний скінченний елмент
class TFE1D2(TFE1D):
```

```
    def __init__(self):
        super().__init__()
        self.size = 2
        # Параметри квадратур
        self._xi = [-0.774596669241483, 0, 0.774596669241483]
        self._w = [5 / 9, 8 / 9, 5 / 9]

    def _create(self):
        if abs(self.x[1][0] - self.x[0][0]) == 0.0:
            raise TException('incorrect_fe_err')
        self.a = zeros((self.size, self.size))
        self.a[0][0] = self.x[1][0]/(self.x[1][0] - self.x[0][0])
        self.a[0][1] = -1.0/(self.x[1][0] - self.x[0][0])
        self.a[1][0] = self.x[0][0]/(self.x[0][0] - self.x[1][0])
        self.a[1][1] = -1.0/(self.x[0][0] - self.x[1][0])

    def _dx(self, i, j):
        return self.a[j][1]

    # Ізопараметричні функції форми та їх похідні
    def _shape(self, i, j):
        return array([(1 - self._xi[i]) / 2, (1 + self._xi[i]) / 2])[j]
    def _shape_dxi(self, i):
        return array([-1 / 2, 1 / 2])
```

Аналогічним чином описуються двовимірні елементи лінійного тривузлового трикутника (TFE2D3), квадратичного шестивузлового трикутника (TFE2D6) і білінійного чотирьохвузлового чотирикутника (TFE2D4).

```
# Лінійний (тривузловий) трикутний скінченний елемент
class TFE2D3(TFE2D):
    def __init__(self):
        super().__init__()
```

```
        self.size = 3
        # Параметри квадратур
        ...

    def _create(self):
        ...

    # Похідні функцій форми
    def _dx(self, i, j):
        return self.a[j][1]
    def _dy(self, i, j):
        return self.a[j][2]

    # Ізопараметричні функції форми та їх похідні
    def _shape(self, i, j):
        return array([1 - self._xi[i] - self._eta[i], self._xi[i], self._eta[i]])[j]
    def _shape_dxi(self, i):
        return array([-1, 1, 0])
    def _shape_deta(self, i):
        return array([-1, 0, 1])

# Квадратичний (шостивузловий) трикутний скінченний елемент
#class TFE2D6(TFE2D3, TFE2D):
class TFE2D6(TFE2D3):
    def __init__(self):
        super().__init__()
        self.size = 6
        # Параметри квадратур Гаусса
        ...

    def _create(self):
        ...
```

```
    # Похідні функцій форми
    def _dx(self, i, j):
        return self.a[j][1] + self.a[j][3] * self.x[i][1] + 2 * self.a[j][4] * self.x[i][0]
    def _dy(self, i, j):
        return self.a[j][2] + self.a[j][3] * self.x[i][0] + 2 * self.a[j][5] * self.x[i][1]

    # Ізопараметричні функції форми та їх похідні
    def _shape(self, i, j):
        s = array([1 - self._xi[i] - self._eta[i], self._xi[i], self._eta[i]])
        return array([s[0] * (2 * s[0] - 1), s[1] * (2 * s[1] - 1), s[2] * (2 * s[2] - 1), 4 * s[0] * s[1],
                      4 * s[1] * s[2], 4 * s[0] * s[2]])[j]
    def _shape_dxi(self, i):
        return array([-3 + 4 * self._xi[i] + 4 * self._eta[i], 4 * self._xi[i] - 1, 0,
                      -8 * self._xi[i] + 4 - 4 * self._eta[i], 4 * self._eta[i], -4 * self._eta[i]])
    def _shape_deta(self, i):
        return array([-3 + 4 * self._xi[i] + 4 * self._eta[i], 0, 4 * self._eta[i] - 1, -4 * self._xi[i],
                      4 * self._xi[i], -8 * self._eta[i] + 4 - 4 * self._xi[i]])

# Білінійний чотирьохвузловий двовимірний скінченний елемент
class TFE2D4(TFE2D):
    def __init__(self):
        super().__init__()
        self.size = 4
        # Параметри квадратур Гаусса
        ...

    def _create(self):
        ...
    # Похідні функцій форми
    def _dx(self, i, j):
        return self.a[j][1] + self.a[j][3] * self.x[i][1]
    def _dy(self, i, j):
        return self.a[j][2] + self.a[j][3] * self.x[i][0]
```

```
    # Ізопараметричні функції форми та їх похідні
    def _shape(self, i, j):
        return array([
            0.25 * (1.0 - self._xi[i]) * (1.0 - self._eta[i]),
            0.25 * (1.0 + self._xi[i]) * (1.0 - self._eta[i]),
            0.25 * (1.0 + self._xi[i]) * (1.0 + self._eta[i]),
            0.25 * (1.0 - self._xi[i]) * (1.0 + self._eta[i])
        ])[j]
    def _shape_dxi(self, i):
        return array([
            -0.25 * (1.0 - self._eta[i]),
            0.25 * (1.0 - self._eta[i]),
            0.25 * (1.0 + self._eta[i]),
            -0.25 * (1.0 + self._eta[i])
        ])
    def _shape_deta(self, i):
        return array([
            -0.25 * (1.0 - self._xi[i]),
            -0.25 * (1.0 + self._xi[i]),
            0.25 * (1.0 + self._xi[i]),
            0.25 * (1.0 - self._xi[i])
        ])
```

Так само описуються і стандартні тривимірні елементи. Так, наприклад, лінійний чотирьохвузловий тетраедральний елемент (TFE3D4) у PyFEM написано таким чином.

```
# Лінійний (чотирьохвузловий) тетраедральний скінченний елемент
class TFE3D4(TFE3D):
    def __init__(self):
        super().__init__()
        self.size = 4
```

```
    # Параметри квадратур Гаусса
    self._xi = [1 / 4, 1 / 2, 1 / 6, 1 / 6, 1 / 6]
    self._eta = [1 / 4, 1 / 6, 1 / 2, 1 / 6, 1 / 6]
    self._psi = [1 / 4, 1 / 6, 1 / 6, 1 / 2, 1 / 6]
    self._w = [-4 / 30, 9 / 120, 9 / 120, 9 / 120, 9 / 120]

def _create(self):
    a, self.a = zeros((self.size, self.size)), zeros((self.size, self.size))
    for j in range(self.size):
        b = array([0.0, 0.0, 0.0, 0.0])
        for i in range(self.size):
            a[i][0] = 1.0
            a[i][1] = self.x[i][0]
            a[i][2] = self.x[i][1]
            a[i][3] = self.x[i][2]
        b[j] = 1.0
        try:
            x = solve(a, b)
        except LinAlgError:
            raise TException('incorrect_fe_err')
        self.a[j] = list(x)

# Похідні функцій форми
def _dx(self, i, j):
    return self.a[j][1]
def _dy(self, i, j):
    return self.a[j][2]
def _dz(self, i, j):
    return self.a[j][3]

# Ізопараметричні функції форми та їх похідні
def _shape(self, i, j):
    return array([1 - self._xi[i] - self._eta[i] - self._psi[i], self._xi[i], self._eta[i], self._psi[i]])[j]
def _shape_dxi(self, i):
```

```
        return array([-1, 1, 0, 0])
    def _shape_deta(self, i):
        return array([-1, 0, 1, 0])
    def _shape_dpsi(self, i):
        return array([-1, 0, 0, 1])
```

Аналогічно описуються квадратичний десятивузловий тетраедральний елемент (TFE3D10) і білінійний восьмивузловий гексаедральний СЕ.

```
# Квадратичний (десятивузловий) тетраедральний скінченний елемент
class TFE3D10(TFE3D4):
    def __init__(self):
        super().__init__()
        self.size = 10
    ...

# Восьмивузловий гексаедральний скінченний елемент
class TFE3D8(TFE3D):
    def __init__(self):
        super().__init__()
        self.size = 8
        # Параметри квадратур Гаусса
        ...
    ...
```

Нестандартні СЕ для опису пластин і оболонок, які в загальному випадку реалізуються достатньо складними структурами даних, в запропонованій архітектурі тепер описуються дуже просто. СЕ трикутної (TFE2D3P, TFE2D6P) і чотирикутної (TFE2D4P) пластин описуються так.

```
# Лінійний трикутний скінченний елемент пластини
class TFE2D3P(TFEP, TFE2D3):
```

```
    def __init__(self):
        super().__init__()

# Квадратичний трикутний скінченний елемент пластини
class TFE2D6P(TFEP, TFE2D6):
    def __init__(self):
        super().__init__()

# Білінійний чотирикутний скінченний елемент пластини
class TFE2D4P(TFEP, TFE2D4):
    def __init__(self):
        super().__init__()
```

Скінченні елементи для моделювання оболонкових конструкцій на прикладі трикутного елементу описуються так.

```
# Лінійний трикутний скінченний елемент оболонки
class TFE2D3S(TFES, TFE2D3P):
    def __init__(self):
        super().__init__()
        self.global_x = zeros((3, 3))

    def _create(self):
        self.T = create_transform_matrix(self.x)
        self.global_x = self.x
        self.x = array([self.T.dot(self.x[0, :]), self.T.dot(self.x[1, :]), self.T.dot(self.x[2, :])])
        TFE2D3._create(self)
```

Таким чином, логічна схема взаємозв’язків між класами, що реалізують скінченний елемент у бібліотеці PyFEM (на прикладі двовимірного випадку), може бути представлена наступним чином (рис. 3.6).

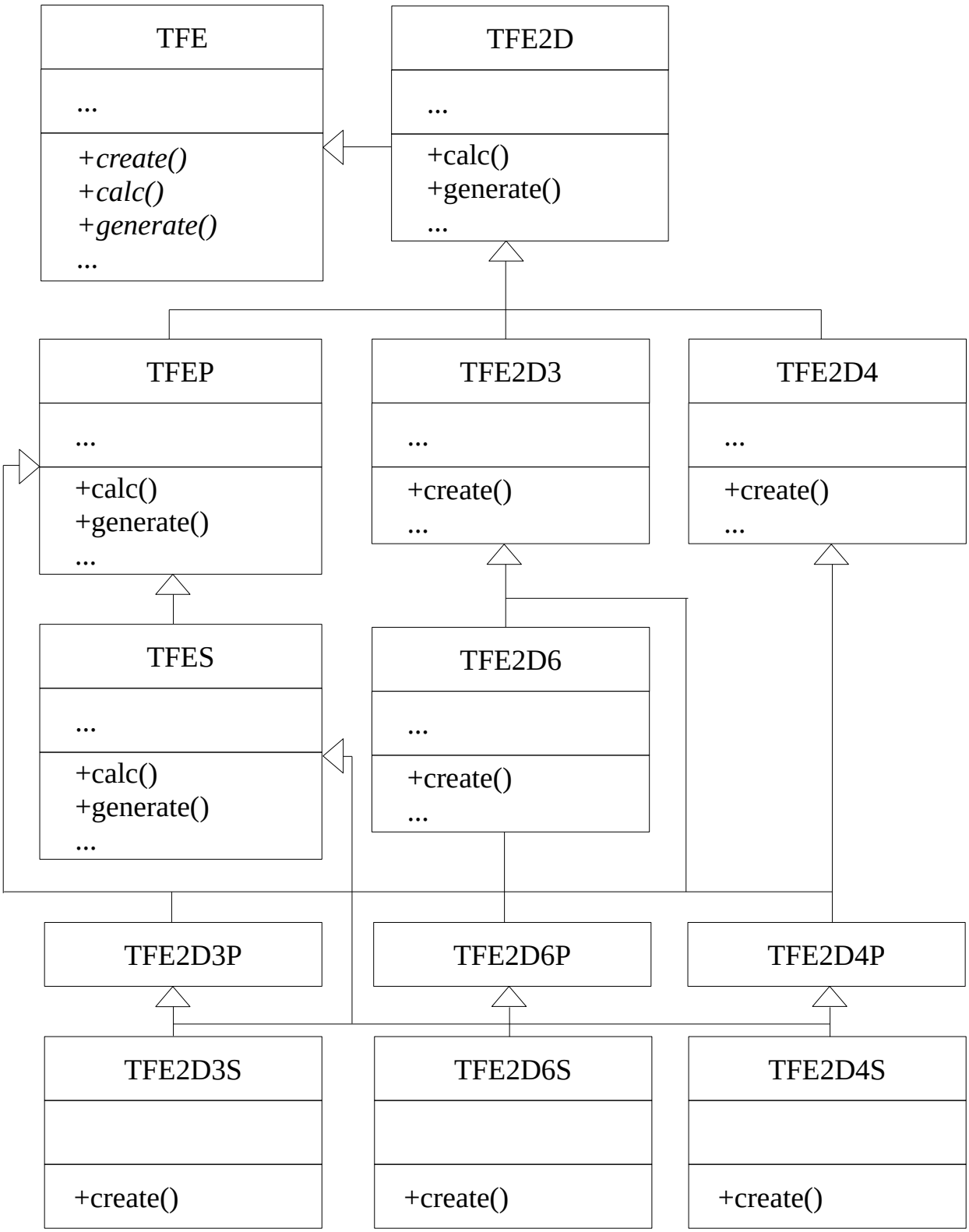

Рис. 3.6 – Ієрархія класів, що реалізують стандартні двовимірні елементи, а також СЕ пластин та оболонок

Така архітектура дозволяє, по-перше, забезпечити повторне використання коду за рахунок уніфікації методів створення СЕ (create()), побудови локальних матриць жорсткості, маси і демпфування (generate()), а також розрахунку стандартних результатів (calc()). А, по-друге, істотно полегшити додавання нових типів СЕ до бібліотеки.

Початковий код реалізації розрахунків оболонкових СЕ наведено у Додатку Б.

3.3 Проєктування класів для реалізації скінченно-елементного розрахунку

Всі необхідні співвідношення для практичної реалізації МСЕ при розв’язанні задач статики можна отримати з використанням варіаційного принципу д’Аламбера-Лагранжа [80, 81]:

$$\delta(\Pi - A) = 0 . \qquad (3.6)$$

Тут $\Pi = \int_{\Omega} \sum_{i,j} \sigma_{ij} \varepsilon_{ij} d\Omega$ – кінетична енергія об’єкта розрахунку;

$A = \int_{\Omega} \sum_{i} X_i u_i d\Omega + \int_{\Gamma} \sum_{i} \overline{X}_i u_i d\Gamma$ – робота зовнішніх (X_i) та внутрішніх ($\overline{X}_i$) сил, що діють на об’єкт розрахунку;

σ_{ij} – компоненти тензору напружень;

ε_{ij} – компоненти тензору деформацій;

u_i – компоненти вектору переміщень;

Ω – область, яку займає об’єкт розрахунку;

Γ – границя Ω.

Аналогічно, для розв’язання задач динаміки розрахункові співвідношення можна безпосередньо вивести з варіаційного принципу Гамільтона-Остроградського [81]:

$$\delta\left(K-\Pi-A\right)=0\,. \tag{3.7}$$

Тут K – кінетична енергія об’єкту розрахунку.

Розглянемо процедуру отримання розрахункової схеми розв’язання задач статики із застосуванням МСЕ. Співвідношення (3.6), яке фактично описує закон збереження енергії, можна записати таким чином:

$$\delta\left(\int\limits_{\Omega}\sum_{i,j}\sigma_{ij}\,\varepsilon_{ij}\,d\,\Omega-\int\limits_{\Omega}\sum_{i}X_i\,u_i\,d\,\Omega-\int\limits_{\Gamma}\sum_{i}\overline{X}_i\,u_i\,d\,\Gamma\right)=0\,. \tag{3.8}$$

Для виведення необхідних для чисельного розрахунку співвідношень слід скористатися законом Гуку, який виражає залежність напружень від деформацій, співвідношеннями Коші, які визначають залежність деформацій від переміщень, і співвідношення для апроксимації переміщень [82]:

$$\begin{gathered}\sigma_{ij}=\sigma_{ij}\left(\varepsilon_{ij}\right),\\ \varepsilon_{ij}=\varepsilon_{ij}\left(u_i\right),\\ u_i=u_i\left(x_1,x_2,x_3\right).\end{gathered} \tag{3.9}$$

У МСЕ частіше за все для визначення компонент вектору переміщень застосовується така їх апроксимація у вузлах елементів [83]:

$$u_i\left(x_{1,}\,x_{2,}\,x_3\right)=\sum_{k=1}^{n}u_{i,k}\,N_k\left(x_{1,}\,x_{2,}\,x_3\right). \tag{3.10}$$

Тут $u_{i,k}$ – вузлові значення переміщень, що потребують обчислення;

$N_k(x_1, x_2, x_3)$ – функції форми, притаманні для обраного типу елемента [1],

n – кількість вершин елемента.

Підставляючи (3.10) у (3.9), отримаємо такі формули для деформацій та напружень:

$$\varepsilon_{ij} = \sum_{k=1}^{n} u_{i,k} \frac{\partial N_k}{\partial x_i} + (1-\delta_{ij}) u_{j,k} \frac{\partial N_k}{\partial x_j}, \tag{3.11}$$

$$\sigma_{ij} = K\left(\sum_{k=1}^{n} u_{i,k} \frac{\partial N_k}{\partial x_i} + (1-\delta_{ij}) u_{j,k} \frac{\partial N_k}{\partial x_j}\right) + \lambda\left(\sum_{m=1}^{3}\sum_{k=1}^{n} u_{m,k} \frac{\partial N_k}{\partial x_m}\right),\ i = j, \tag{3.12}$$

$$\sigma_{ij} = G\left(\sum_{k=1}^{n} u_{i,k} \frac{\partial N_k}{\partial x_i} + u_{j,k} \frac{\partial N_k}{\partial x_j}\right),\ i \neq j, \tag{3.13}$$

де δ_{ij} – символ Кронекера;

K, G, λ – числові параметри, що залежать від пружних характеристик об’єкту розрахунку.

Далі із використанням співвідношень (3.8), (3.11)-(3.13) отримаємо таке співвідношення:

$$\delta(\alpha_1 u_1^2 + \alpha_2 u_1 u_2 + \alpha_3 u_1 u_3 + \alpha_4 u_2^2 + ... + \alpha_q u_2 u_3) = 0. \tag{3.14}$$

Тут α_l – числові коефіцієнти, що залежать від координат поточного елемента та пружних характеристик об’єкта розрахунку. Ця варіаційна задача може бути розв’язана, наприклад, із застосуванням методу варіації параметрів [84]. Отже, вихідна задача (3.6) із урахуванням граничних умов буде зведена до СЛАР наступного виду:

$$\boldsymbol{K}\,\boldsymbol{u}=\boldsymbol{l}\,, \tag{3.15}$$

де $\boldsymbol{K}$ – симетрична (в більшості випадків також розріджена) матриця коефіцієнтів СЛАР;

$\boldsymbol{u}$ – вектор шуканих вузлових переміщень;

$\boldsymbol{l}$ – вектор правої частини СЛАР, що визначає вузлові навантаження, які діють на об’єкт розрахунку.

Так само можна побудувати всі необхідні розрахункові співвідношення для задач динаміки, які описуються співвідношенням (3.7).

Розглянемо реалізацію скінченно-елементного розрахунку для задач статики і динаміки, який базується на наведені вище співвідношення. Для врахування можливості розв’язання різних типів задач в бібліотеці PyFEM було створено ієрархічну структуру класів (рис. 3.7), які реалізують відповідну функціональність.

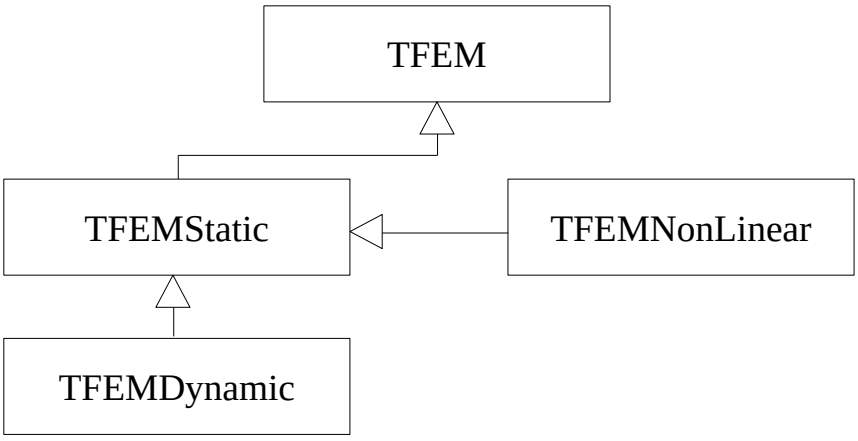

Рис. 3.7 – Ієрархія класів, що реалізують скінченно-елементний розрахунок

Базовим тут є абстрактний клас TFEM. Він містить спільні для всіх типів розрахунків наступні структури даних: дискретну модель області розрахунку, параметри розрахунку і результати розрахунку.

Його реалізація на Python виглядає так.

```
# Абстрактний базовий клас, що реалізує метод скінченних елементів
class TFEM:
    def __init__(self):
        self.mesh = TMesh()                 # Дискретна модель області розрахунку
        self.params = TFEMParams()      # Параметри розрахунку
        self.results = []                       # Список результатів розрахунку
        self.progress = TProgress()       # Індикатор прогресу розрахунків

    # Абстрактні методи
    @abstractmethod
    def _calc_problem(self):
        raise NotImplementedError('Method TFEM._calc_problem is pure virtual')
    # Додавання локальної матриці жорсткості (мас, демпфування) до глобальної
    @abstractmethod
    def __assembly(self, fe, index):
        raise NotImplementedError('Method TFEM.__assembly is pure virtual')
    # Обчислення допоміжних результатів
    @abstractmethod
    def _calc_results(self):
        raise NotImplementedError('Method TFEM._calc_results is pure virtual')
    # Пряме розв’язання СЛАР
    @abstractmethod
    def _solve_direct(self):
        raise NotImplementedError('Method TFEM._solve_direct is pure virtual')
    # Наближене ...
    @abstractmethod
    def _solve_iterative(self):
        raise NotImplementedError('Method TFEM._solve_iterative is pure virtual')
    # Розв’язання СЛАР
    def _solve(self):
        ret = False
        if self.params.solve_method == 'direct':
            ret = self._solve_direct()
        elif self.params.solve_method == 'iterative':
```

```
        ret = self._solve_iterative()
    return ret
# Створення скінченного елемента
def create_fe(self):
    fe = TFE()
    if self.mesh.fe_type == 'fe_1d_2':
        fe = TFE1D2()
    elif self.mesh.fe_type == 'fe_2d_3':
        fe = TFE2D3()
    elif self.mesh.fe_type == 'fe_2d_4':
        fe = TFE2D4()
    elif self.mesh.fe_type == 'fe_2d_6':
        fe = TFE2D6()
    elif self.mesh.fe_type == 'fe_3d_4':
        fe = TFE3D4()
    elif self.mesh.fe_type == 'fe_3d_8':
        fe = TFE3D8()
    elif self.mesh.fe_type == 'fe_3d_10':
        fe = TFE3D10()
    elif self.mesh.fe_type == 'fe_2d_3_p':
        fe = TFE2D3P()
    elif self.mesh.fe_type == 'fe_2d_6_p':
        fe = TFE2D6P()
    elif self.mesh.fe_type == 'fe_2d_4_p':
        fe = TFE2D4P()
    elif self.mesh.fe_type == 'fe_3d_3_s':
        fe = TFE2D3S()
    elif self.mesh.fe_type == 'fe_3d_6_s':
        fe = TFE2D6S()
    elif self.mesh.fe_type == 'fe_3d_4_s':
        fe = TFE2D4S()
    return fe
# Запуск розрахунку
def calc(self):
```

```
        try:
            # Перевірка наявності та відповідності необхідних параметрів розрахунку
            self.params.check_params()
            ret = self._calc_problem()
        except TException as err:
            ret = False
            err.print_error()
        return ret
    # Вибір сітки
    def set_mesh(self, mesh):
        self.mesh = mesh
    # Вибір параметрів розрахунку
    def set_params(self, params):
        self.params = params
```

Тут слід зазначити, що для створення об’єкту, який описує СЕ, використано фабричний метод, який притаманний багатьом патернам проєктування, і, насамперед, Builder.

Похідний від TFEM клас TFEMStatic реалізує стандартний розрахунок задачі статики із застосуванням МСЕ. Його реалізація у PyFEM має наступний вигляд.

```
# Клас, що реалізує розрахунок задачі статики
class TFEMStatic(TFEM):
    def __init__(self):
        super().__init__()
        self._global_matrix_stiffness = lil_matrix((0, 0))  # Глобальна матриця жорсткості
        self._global_load = []                               # Глобальний вектор навантаження
        self._fe_thickness = []                              # Товщина кожного елемента

    # Процедура розрахунку статичної задачі
    def _calc_problem(self):
```

```
    ...
# Додавання локальної матриці жорсткості до глобальної
def __assembly(self, fe, index):
    ...
# Обчислення зосереджених навантажень
def _use_concentrated_load(self, t=0):
    ...
# Обчислення поверхневих навантажень
def _use_surface_load(self, t=0):
    ...
# Обчислення об'ємних навантажень
def _use_volume_load(self, t=0):
    ...
# Обчислення навантажень поверхневого тиску
def _use_pressure_load(self, t=0):
    ...
# Обчислення допоміжних результатів
def _calc_results(self, t=0):
    ...
# Завдання граничних умов
def _set_boundary_condition(self, i, j, val):
    ...
# Врахування граничних умов
def _use_boundary_condition(self):
    ...
# Пряме рішення СЛАР
def _solve_direct(self):
    ...
# Наближене рішення СЛАР
def _solve_iterative(self):
    ...
# Налаштування параметрів скінченного елементу
def _set_fe(self, fe, fe_index):
    ...
```

...

Аналогічним чином можна описати похідний від TFEMStatic клас TFEMDynamic для розв’язання задач динаміки.

```
# Клас, що реалізує розрахунок динамічної задачі
class TFEMDynamic(TFEMStatic):
    def __init__(self):
        super().__init__()
        self._global_matrix_mass = lil_matrix((0, 0))       # Глобальна матриця маси
        self._global_matrix_damping = lil_matrix((0, 0))  # Глобальна матриця демпфування

    # Розрахунок динамічної задачі
    def _calc_problem(self):
        …
    # Обробка початкових умов
    def __prepare_initial_condition(self):
        …
    # Обчислення напружень, деформацій, швидкостей та прискорень
    def __calc_dynamic_results(self, u0, ut0, utt0, t):
        …
    # Додавання локальних матриць жорсткості, маси та демпфування до глобальної
    def __assembly(self, fe, index):
        …
    # Формування лівої частини рівняння квазістатичної рівноваги
    def __create_dynamic_matrix(self):
        …
    # Формування правої частини рівняння квазістатичної рівноваги
    def __create_dynamic_vector(self, u0, ut0, utt0, t):
        …
```

Аналогічним чином описуються інші похідні класи, що реалізують розрахунок певного типу задачі.

3.4 Інтерфейс бібліотеки PyFEM

Центральною частиною бібліотеки PyFEM є клас TObject, який інкапсулює об’єкт розрахунку (рис. 3.8).

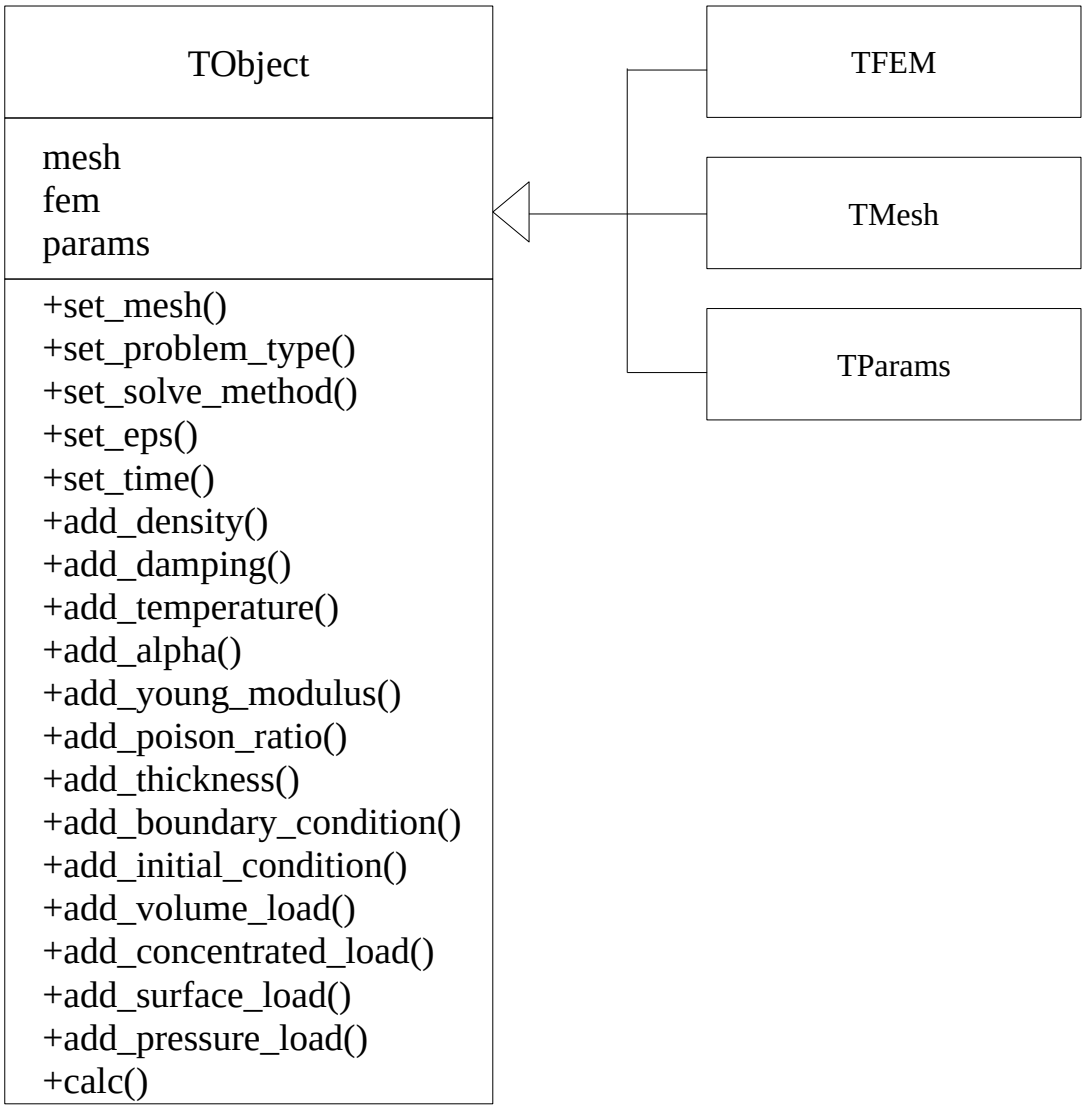

Рис. 3.8 – Інтерфейс бібліотеки PyFEM

Фактично цей клас є високорівневою обгорткою для класу TFEM та його спадкоємців, а також класів, що описують скінченно-елементну сітку і параметри розрахунку.

Він реалізує інтерфейс користувача доступу до бібліотеки PyFEM. Керуючи методами даного класу, користувач може визначити новий об’єкт розрахунку, задати його скінченно-елементну модель, пружні та фізичні характеристики, крайові умови та навантаження, а також інші необхідні для розрахунку параметри. Користувач також може вибрати спосіб вирішення СЛАР (прямий або ітераційний), формат виведення результатів розрахунку у файл чи на екран і таке інше. Приклади застосування цього класу будуть наведені у наступному розділі.

Висновки до розділу 3

Отже, на основі проведених досліджень можна стверджувати, що на сьогодні одна з найбільш поширених парадигм програмування, яка частіше за все використовується на практиці для розробки складного (у тому числі і наукового) ПЗ, є об’єктно-орієнтований підхід. Аналіз найбільш поширених на сьогодні патернів проєктування та розробки програмного забезпечення показав, що використання патерну Builder і фабричних методів дозволяють ефективно реалізовувати гнучкий механізм уніфікованого створення об’єктів, що описують різні типи СЕ, а також об’єктів, які реалізують розрахунок потрібних користувачу типів задач.

Запропонована у третьому розділі об’єктно-орієнтована архітектура бібліотеки скінченно-елементного розрахунку PyFEM дозволяє легко розширювати її функціональність для врахування нових типів СЕ, а також нових алгоритмів скінченно-елементного розрахунку.

Основні наукові і практичні результати даного розділу опубліковано в роботах [45-48, 85].

4 ОБЧИСЛЮВАЛЬНІ ЕКСПЕРИМЕНТИ ТА ПРИКЛАДИ ЗАСТОСУВАННЯ БІБЛІОТЕКИ СКІНЧЕННО-ЕЛЕМЕНТНОГО АНАЛІЗУ

4.1 Одновимірні задачі

Розглянемо приклад розв’язання класичної задачі опору матеріалів про розтягування-стиснення стрижня (рис. 4.1).

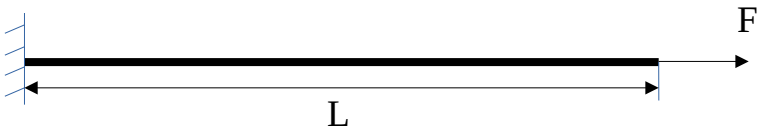

Рис. 4.1 – Задача про розтягування стрижня

Розв’язання цієї задачі за допомогою бібліотеки PyFEM можна виконати наступним чином.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from core.fem_defs import *
from core.fem_object import TObject
from plot.plot3d import TPlot


def body1d(res_name):
    # Створення об’єкту розрахунку
    obj = TObject()
    # Завантаження скінченно-елементної моделі
```

```
    if obj.set_mesh('mesh/body1d.trpa'):
        # Вибір типу задачі - статика чи динаміка
        obj.set_problem_type('static')
        # Вибір методу розв’язання СЛАР - прямий чи ітераційний
        obj.set_solve_method('direct')
        # Пружні характеристики
        obj.add_young_modulus('203200')
        obj.add_poisson_ratio('0')
        # Площа перерізу
        obj.add_thickness('0.01')
        # Граничні умови
        obj.add_boundary_condition(DIR_X, '0', 'x == 0')
        # Зосереджене навантаження
        obj.add_concentrated_load(DIR_X, '1', 'x == 3')
        if obj.calc():
            # Вивід результатів у разі успіху розрахунку
            obj.print_result()
            obj.save_result(res_name)
            # Запуск постпроцесора
            TPlot(res_name)
            return True
        return False

if __name__ == '__main__':
    body1d('body1d')
```

Як видно з наведеного початкового коду задача розв’язувалася при наступних пружних параметрах: $E = 203200$ Па, $\nu = 0$. Довжина стрижня $L = 3$ м, площа перерізу – 0.5 м. Зосереджене навантаження, яке було прикладено до одного (вільного) кінця стрижня $F = 1$ Н, другий кінець стрижня було жорстко затиснено. Скінченно-елементна модель стрижня складається з трьох елементів. Результати розрахунку цієї задачі наведено на рис. 4.2.

```
D:\Work\python\pyfem>python fem_test.py
Object: body1d
Points: 4
FE: 3 - one-dimensional linear element (2 nodes)
Assembling global stiffness matrix... 100%
Computation of concentrated load... 100%
Use of boundary conditions... 100%
Solving of equation system... 100%
Calculation results... 100%
**************** Success! ****************
Lead time 0.008970 sec
| N  (            x) |            U |          Exx |          Sxx |
| 1  ( +0.00000E+00) | +0.00000E+00 | +4.92126E-04 | +1.00000E+02 |
| 2  ( +1.00000E+00) | +4.92126E-04 | +4.92126E-04 | +1.00000E+02 |
| 3  ( +2.00000E+00) | +9.84252E-04 | +4.92126E-04 | +1.00000E+02 |
| 4  ( +3.00000E+00) | +1.47638E-03 | +4.92126E-04 | +1.00000E+02 |

|                    |            U |          Exx |          Sxx |
|              min:  | +0.00000E+00 | +4.92126E-04 | +1.00000E+02 |
|              max:  | +1.47638E-03 | +4.92126E-04 | +1.00000E+02 |
```

Рис. 4.2 – Результат розрахунку задачі про розтягнення стрижня

Порівняння результатів розрахунку з відомою аналітичною формулою $\Delta l = frac\, N \cdot l\, E \cdot a$, де Δl – подовження вільного торця стрижня, N – сила, E – модуль Юнга, a – площа перетину стрижня [86], дало наступний результат. Аналітично отримане значення подовження складає 0.001476378 м, розрахункове – 0.001476380 м. Відносна похибка в даному випадку складає менше 1%.

Візуалізація розподілу функції подовження (переміщення) по стрижню, отримана за допомогою постпроцесора бібліотеки PyFEM, наведена на рис. 4.3.

4.2 Двовимірні задачі

Розглянемо приклад розв’язання ще одної тестової задачі про вигин балки (рис. 4.4).

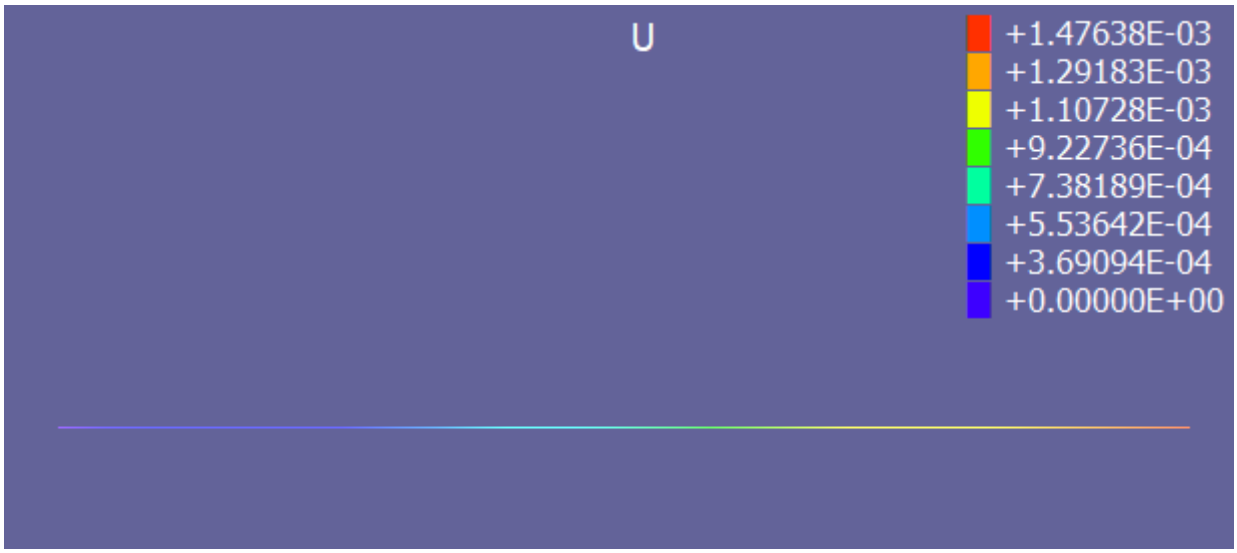

Рис. 4.3 – Розподіл переміщень по стрижню

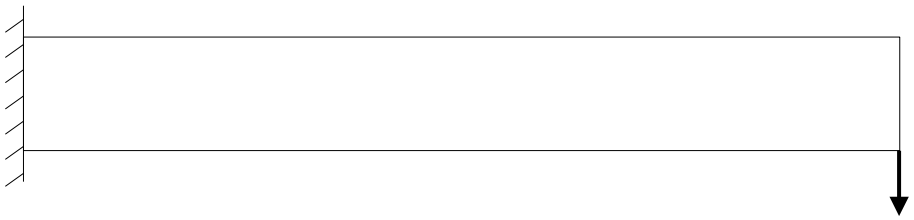

Рис. 4.4 – Задача про вигин балки

Дискретна модель, отримана із застосуванням вільного генератора сіток Gmsh [30], наведена на рис. 4.5. Вона складається з 295 вузлів і 452 СЕ у формі лінійних тривузлових трикутників.

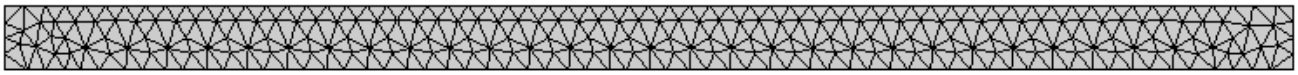

Рис. 4.5 – Дискретна модель балки

Задача розв’язувалася при наступним геометричних параметрах: довжина балки 10 м, ширина – 0.5 м, товщина – 1 м. Пружні параметри балки: модуль Юнга – $6.5 \cdot 10^{10}$ Па, коефіцієнт Пуассона – 0.3. Зосереджена навантаження, прикладене до нижнього краю вільного кінця, дорівнює $1.0 \cdot 10^{6}$ Н.

Опис відповідної функції, яка реалізує розрахунок цієї задачі, має такий вигляд.

```
def console(res_name):
    obj = TObject()
    if obj.set_mesh('mesh/console.trpa'):
        obj.set_problem_type('static')
        obj.set_solve_method('direct')
        obj.add_young_modulus('6.5E+10')
        obj.add_poisson_ratio('0.3')
        obj.add_boundary_condition(DIR_X | DIR_Y, '0', 'x == 0')
        obj.add_concentrated_load(DIR_Y, '-1.0E+6', 'x == 10 and y == -0.25')
        if obj.calc():
            # obj.print_result()
            obj.save_result(res_name)
            TPlot(res_name)
            return True
        return False
```

Результат розрахунку задачі наведено на рис. 4.6.

```
Object: console
Points: 295
FE: 452 - linear triangular element (3 nodes)
Assembling global stiffness matrix... 100%
Computation of concentrated load... 100%
Use of boundary conditions... 100%
Solving of equation system... 100%
Calculation results... 100%
*************** Success! ***************
Lead time 0.824001 sec
```

Рис. 4.6 – Результат розрахунку задачі про вигин балки

Отриманий в результаті розрахунку розподіл основних параметрів напружено-деформованого стану по області балки наведено на рис. 4.7 – 4.11

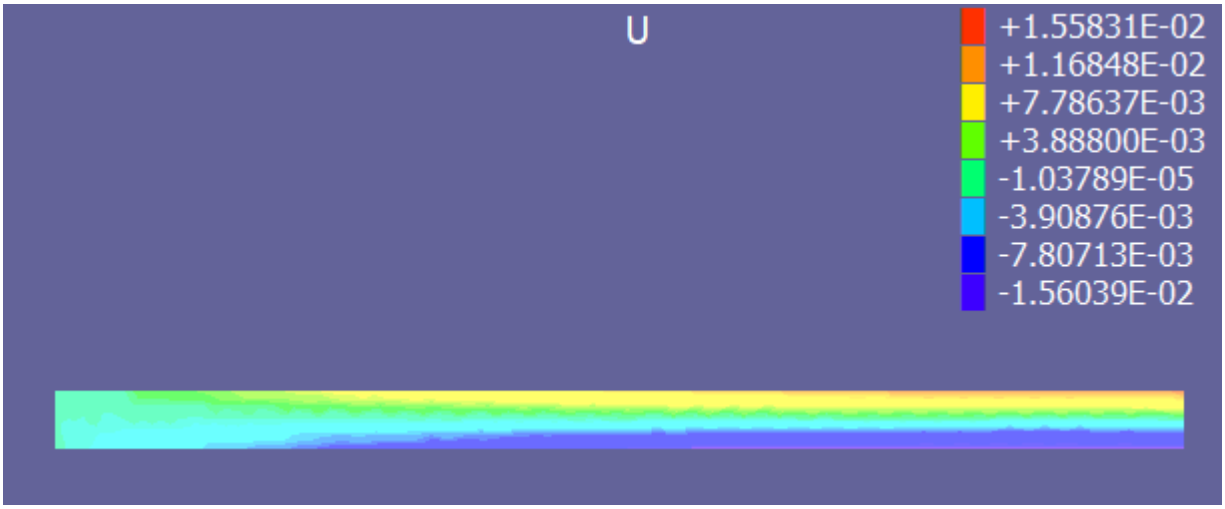

Рис. 4.7 – Розподіл компонент вектору переміщень U по балці

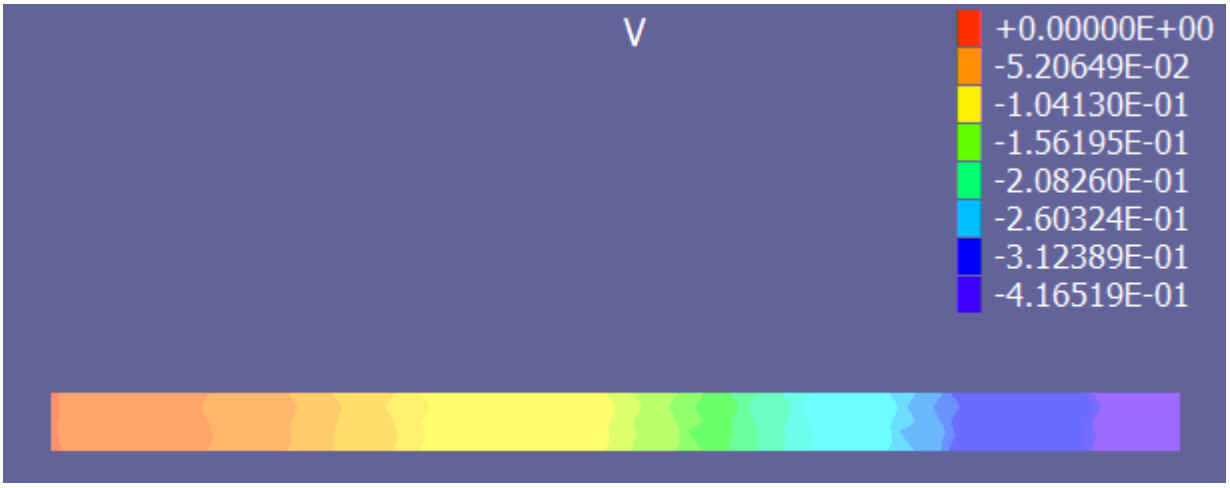

Рис. 4.8 – Розподіл компонент вектору переміщень V

Порівняння отриманих результатів з відомим аналітичним розв’язком цієї задачі [87] показало, що максимальна відносна похибка не перевищує 2%, що є цілком прийнятним для чисельного наближеного розрахунку.

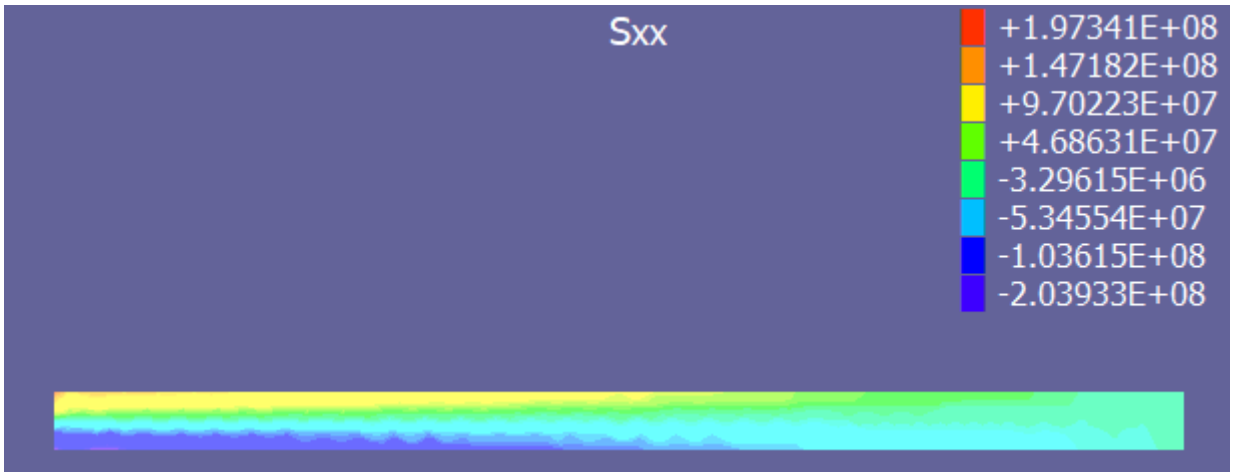

Рис. 4.9 – Розподіл компоненти тензору напружень σ_{xx}

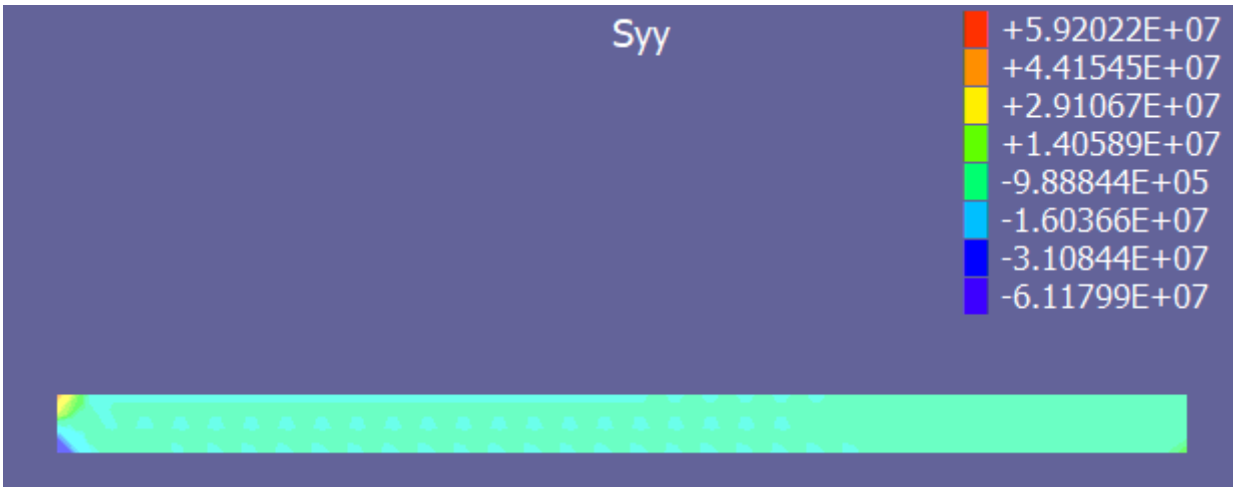

Рис. 4.10 – Розподіл компоненти тензору напружень σ_{yy}

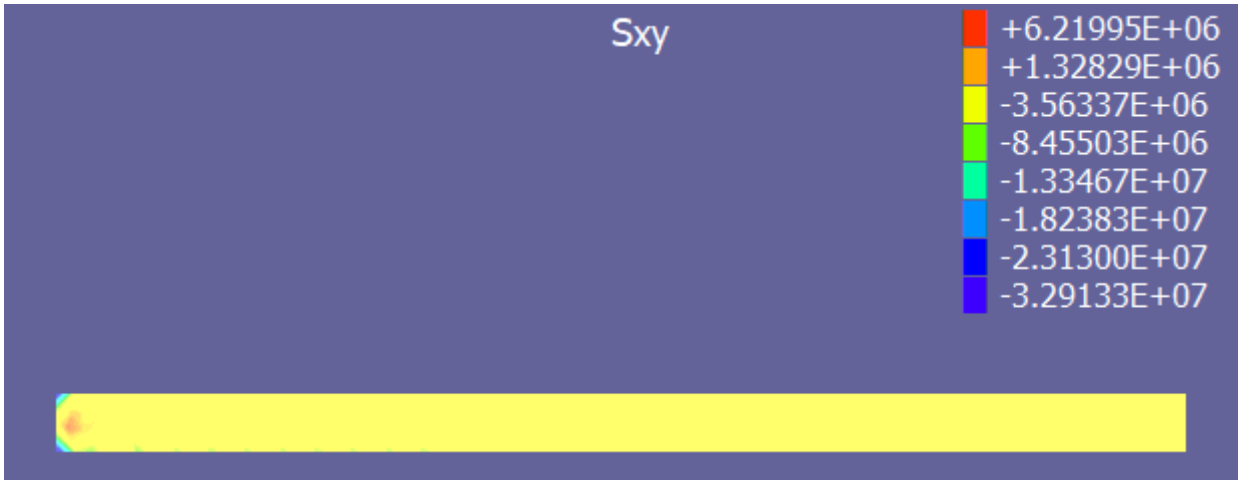

Рис. 4.11 – Розподіл компоненти тензору напружень σ_{xy}

4.3 Пластини та оболонки

Розрахунок тонких пластин та оболонок із застосуванням МСЕ на практиці є доволі непростою задачею, оскільки потребує програмної реалізації спеціальних типів СЕ. Завдяки запропонованій у попередньому розділі архітектурі додавання нових типів елементів у бібліотеку PyFEM є доволі простою задачею. На сьогодні у PyFEM реалізовані класи, що описують наступні СЕ оболонок і пластин: лінійний трикутний елемент пластини і оболонки, білінійний чотирикутний елемент пластини і оболонки, квадратичний трикутний СЕ пластини та оболонки.

Розглянемо приклад розв’язання задачі про вигин квадратної пластини, края якої жорстко затиснені, а навантаження (зосереджене або поверхневе) діє в напрямку, ортогональному площині пластини (рис. 4.12).

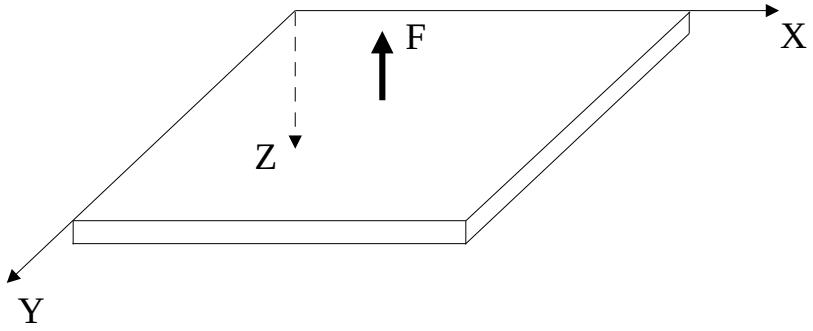

Рис. 4.12 – Задача про прогин пластини

Задача розв’язувалася при наступних параметрах: модуль Юнга – 203200 МПа, коефіцієнт Пуассона – 0.27, довжина бічної сторони пластинки – 1 м, товщина – 0.01 м, рівномірно розподілене по поверхні навантаження – 0.05 МН/м^2. В якості дискретної моделі було обрано рівномірну сітку 200x200 квадратів.

Початковий код функції, що описує цю задачу, на мові Python має наступний вигляд.

```
def plate4(res_name):
```

```
obj = TObject()
if obj.set_mesh('mesh/plate4.trpa'):
    obj.set_problem_type('static')
    obj.set_solve_method('direct')
    obj.add_young_modulus('203200')
    obj.add_poisson_ratio('0.27')
    obj.add_thickness('0.01')
    obj.add_boundary_condition(DIR_X | DIR_Y | DIR_Z, '0',
                               'x == -0.5 or x == 0.5 or y == -0.5 or y == 0.5')
    obj.add_surface_load(DIR_X, '0.05')
    if obj.calc():
        obj.print_result()
        obj.save_result(res_name)
        TPlot(res_name)
        return True
    return False
```

Результат роботи цієї функції наведено на рис. 4.13. Візуалізація розподілу функції прогину по області пластини наведена на рис. 4.14.

```
Object: plate4
Points: 40401
FE: 40000 - plate element (4 nodes)
Assembling global stiffness matrix... 100%
Computation of surface load... 100%
Use of boundary conditions... 100%
Solving of equation system... 100%
Calculation results... 100%
*************** Success! ***************
Lead time 210.750305 sec
```

Рис. 4.13 – Процес розрахунку квадратної пластини

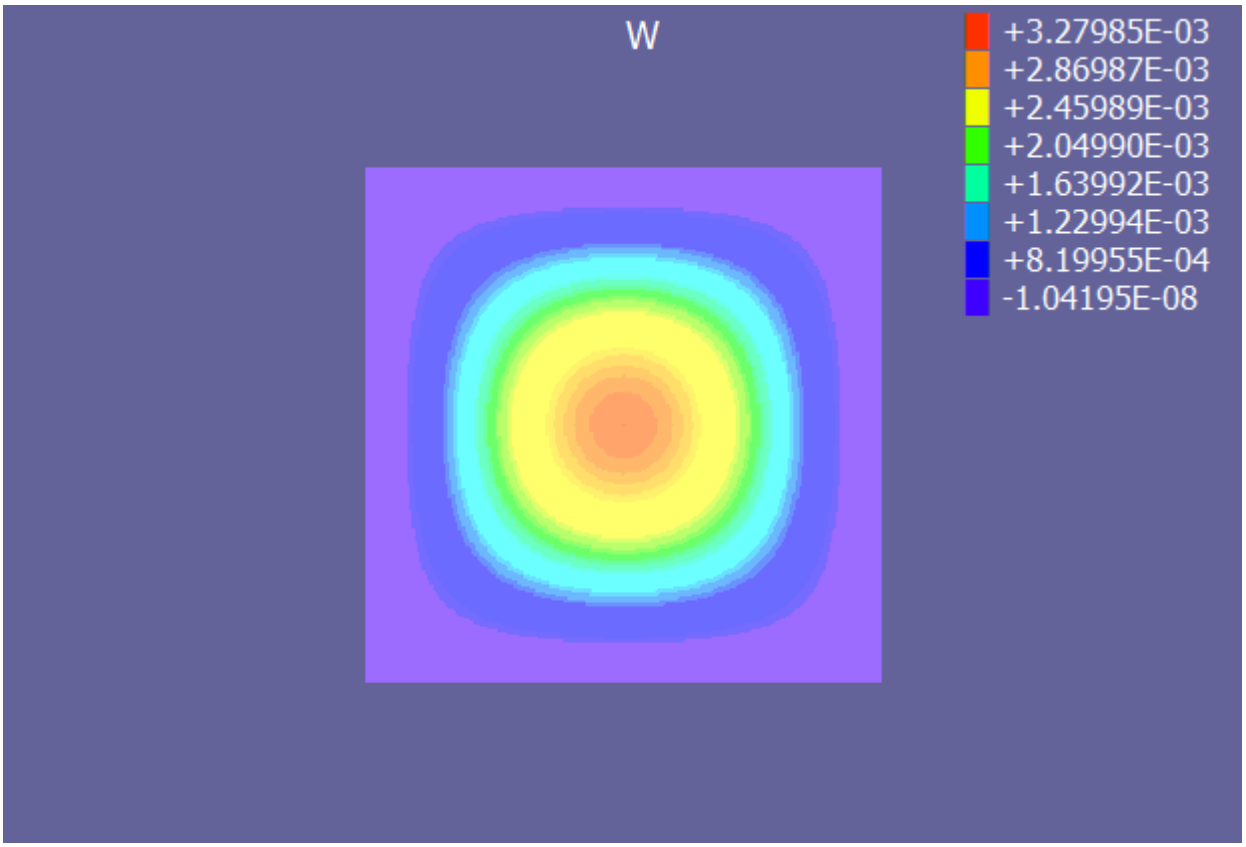

Рис. 4.14 – Розподіл прогину по області пластини

Як видно з даного рисунку, максимальний прогин знаходиться в центрі пластини і дорівнює 3.27985E-03 м. Аналітичний розв'язок цієї задачі [88] дає наступне співвідношення для максимального прогину пластини: $w = 0.00126 \cdot \frac{12 \cdot q \cdot a^4 \cdot (1-\nu^2)}{(E \cdot h^3)}$, де q – навантаження, що діє на пластину, E, ν – її пружні характеристики, a – довжина бічної сторони, а h – товщина пластини. Розрахунок із застосуванням цього співвідношення дає значення $w = 0.0034492499999999996$ м. Тобто відносна похибка наближеного скінченно-елементного розрахунку становить близько 5%.

Для верифікації отриманих результатів було виконано розрахунок цієї задачі із використанням системи скінченно-елементного аналізу qzCAD [14]. Розподіл прогинів по пластинці, отриманий в qzCAD, наведено на рис. 4.15. Порівняння

отриманих результатів показує їх абсолютну тотожність (за умови використання однакової сітки).

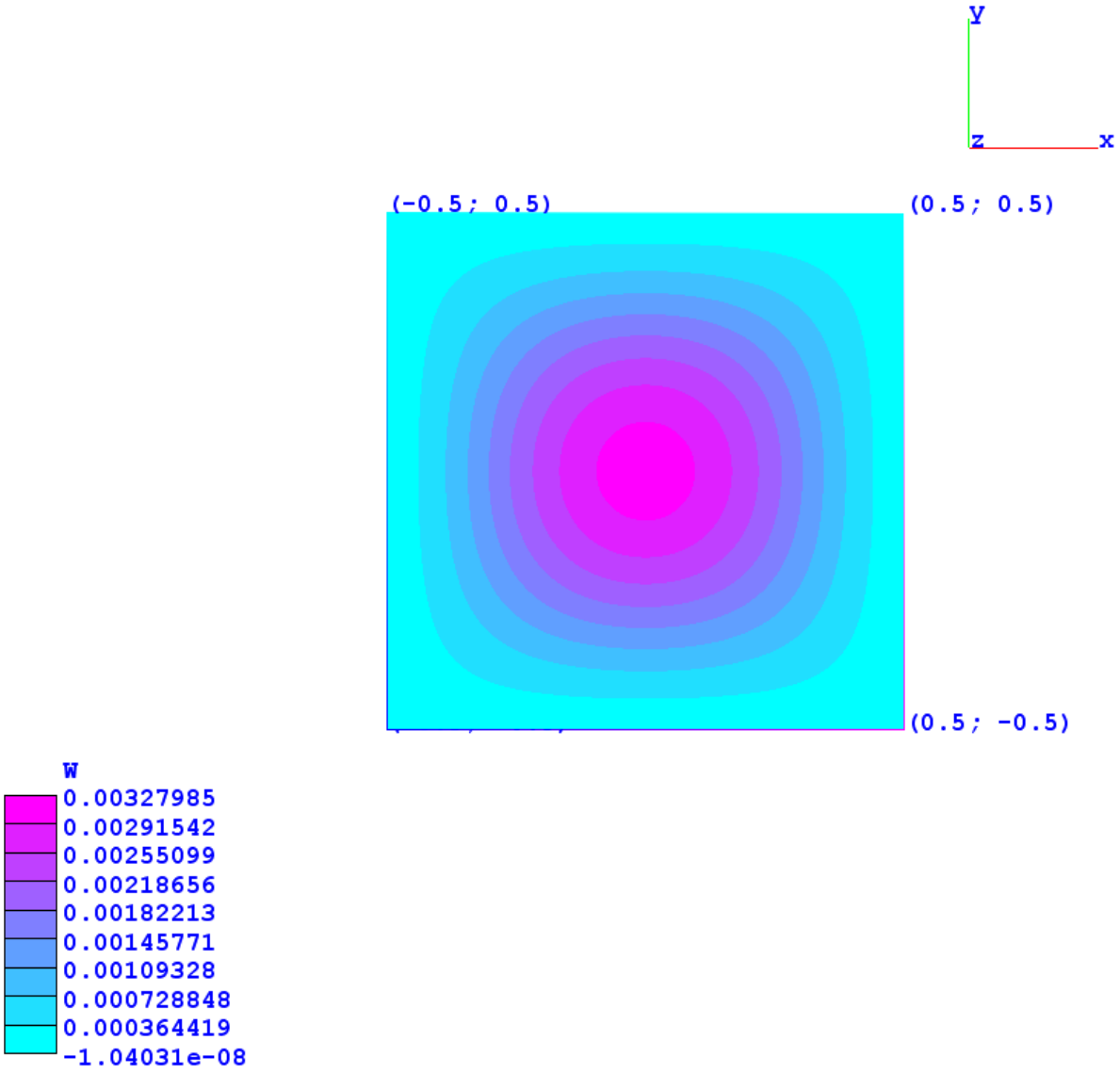

Рис. 4.15 – Візуалізація прогинів пластини, побудована у постпроцесорі qzCAD

Для тестування можливості розв’язання задач про знаходження параметрів напружено-деформованого стану оболонкових конструкцій було розглянуто задачу про циліндричну конструкцію, яка знаходиться під дією внутрішнього тиску, і торці якої жорстко затиснуті (рис. 4.11).

Розрахунок цієї задачі виконувався із застосуванням різних сіток. Спочатку розрахунок виконувався на сітці, побудованій із застосуванням трикутних елементів, яка містить 328 вузлів і 600 СЕ (рис. 4.17).

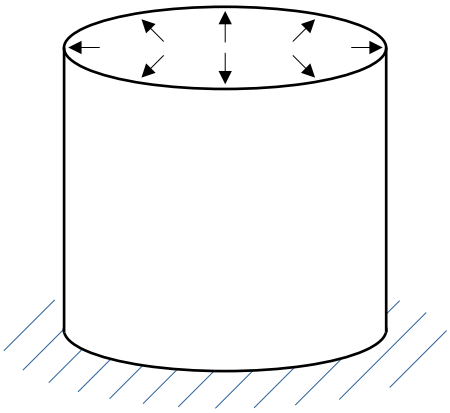

Рис. 4.16 – Труба з жорстко затисненими торцями під дією внутрішнього тиску (з урахуванням симетрії)

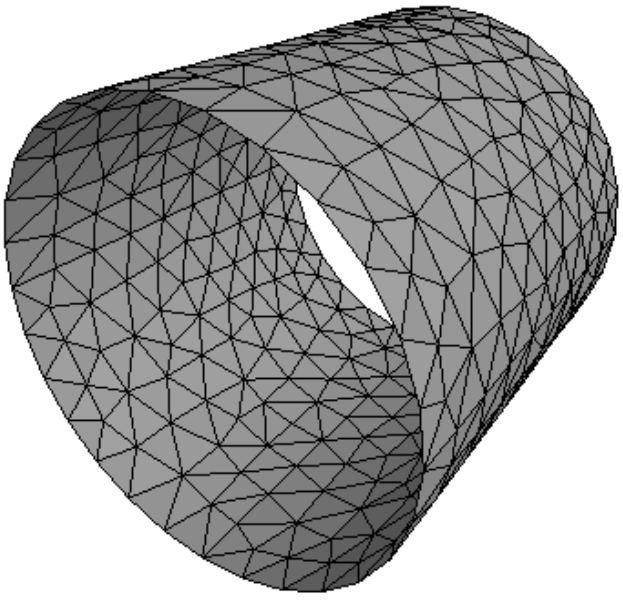

Рис. 4.17 – Скінченно-елемента модель оболонки, побудована із використанням лінійних трикутників

Задача розв’язувалася при тих самих, як і у попередньому випадку, пружних параметрах і значенні навантаження. Геометричні ж параметри бралися наступними: радіус оболонки – 1.99 м, висота – 4.014 м, товщина стінки 0.0369 м.

Результати розрахунку (фрагмент) наведено на рис. 4.18-4.19. Розподіл компоненти вектору переміщень u по області розрахунку, отриманий із застосуванням постпроцесора бібліотеки PyFEM, наведено на рис. 4.20.

```
Object: shell-tube-3
Points: 328
FE: 600 - triangular shell element (3 nodes)
Assembling global stiffness matrix... 100%
Computation of pressure load... 100%
Use of boundary conditions... 100%
Solving of equation system... 100%
Calculation results... 100%
*************** Success! ***************
Lead time 1.432780 sec
|   N  (           x,           y,           z) |           U |           V |           W |
|   1  ( -1.99000E+00, +0.00000E+00, +0.00000E+00) | +0.00000E+00 | +0.00000E+00 | +0.00000E+00 |
|   2  ( +0.00000E+00, +1.99000E+00, +0.00000E+00) | +0.00000E+00 | +0.00000E+00 | +0.00000E+00 |
|   3  ( +1.99000E+00, +0.00000E+00, +0.00000E+00) | +0.00000E+00 | +0.00000E+00 | +0.00000E+00 |
|   4  ( +0.00000E+00, -1.99000E+00, +0.00000E+00) | +0.00000E+00 | +0.00000E+00 | +0.00000E+00 |
```

Рис. 4.18 – Результати розрахунку труби під дією внутрішнього тиску

```
| 324  ( -3.88230E-01, -1.95176E+00, +1.78400E+00) | -4.94208E-06 | -2.44888E-05 | +9.98168E-08 |
| 325  ( -7.93149E-01, -1.82511E+00, +1.75483E+00) | -1.01991E-05 | -2.31406E-05 | +1.10971E-07 |
| 326  ( -1.04125E+00, -1.69584E+00, +3.83708E-01) | -8.49493E-06 | -1.38128E-05 | +8.19004E-07 |
| 327  ( -5.86169E-01, -1.90171E+00, +2.58564E-01) | -3.27011E-06 | -1.08608E-05 | +5.36332E-07 |
| 328  ( -3.88230E-01, -1.95176E+00, +2.23000E+00) | -4.84543E-06 | -2.40737E-05 | -1.55561E-07 |

|                                          |           U |           V |           W |
|                                    min:  | -2.56648E-05 | -2.56648E-05 | -9.80105E-07 |
|                                    max:  | +2.56648E-05 | +2.56648E-05 | +8.89345E-07 |
```

Рис. 4.19 – Результати розрахунку труби (продовження)

Для верифікації отриманих результатів аналогічний розрахунок виконувався за допомогою системи qzCAD. Тут бралася рівномірна сітка чотирикутних СЕ, яка складається зі 100×50 елементів.

Отриманий в цій програмі розподіл відповідної компоненти вектору переміщень наведено на рис. 4.21.

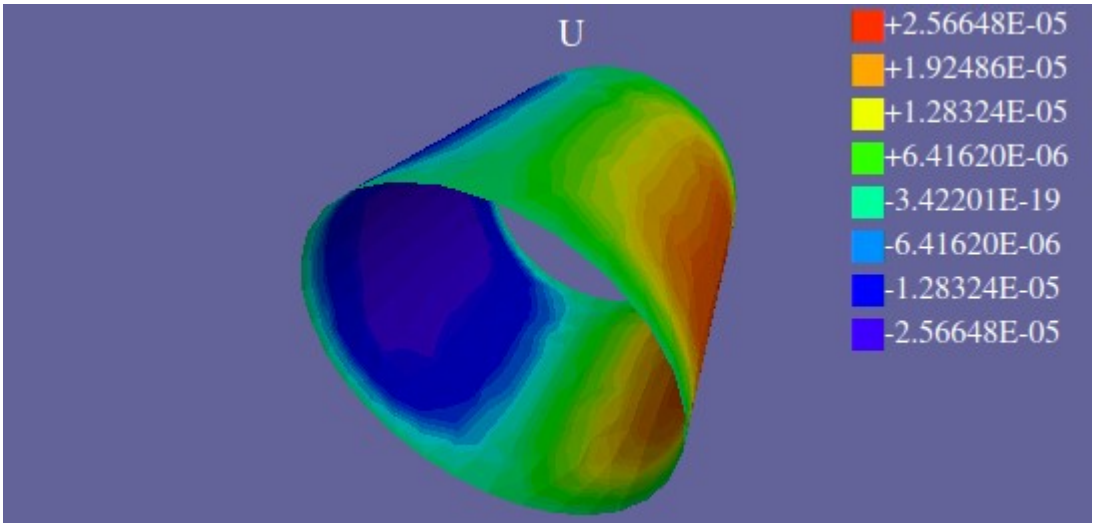

Рис. 4.20 – Розподіл переміщень *u* по конструкції

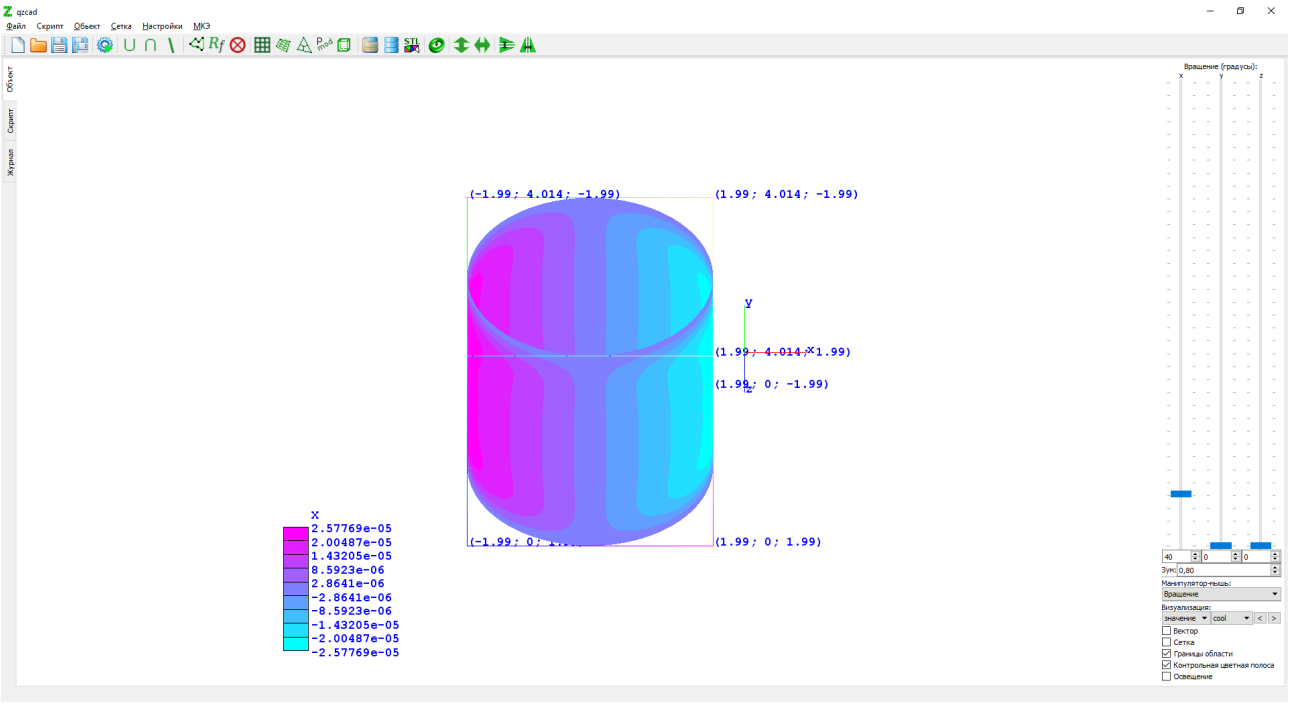

Рис. 4.21 – Розподіл переміщень *u* по оболонці, отриманий у qzCAD

Як видно з цього рисунку, отримані результати добре узгоджуються (відносна похибка становить 4%). Для перевірки точності роботи бібліотеки PyFEM було також виконано розрахунок цієї задачі із застосуванням дискретної

моделі, побудованої із застосуванням квадратичних трикутних СЕ. Результат роботи програми у цьому випадку наведено на рис. 4.22.

```
Object: shell-tube6
Points: 25752
FE: 12748 - triangular shell element (6 nodes)
Assembling global stiffness matrix... 100%
Computation of pressure load... 100%
Use of boundary conditions... 100%
Solving of equation system... 100%
Calculation results... 100%
*************** Success! ***************
Lead time 781.490397 sec
```

Рис. 4.22 – Результат розрахунку оболонки з використанням квадратичного СЕ

Візуалізація розподілу компоненти вектору переміщень u і w по конструкції наведена на рис. 4.23, 4.24.

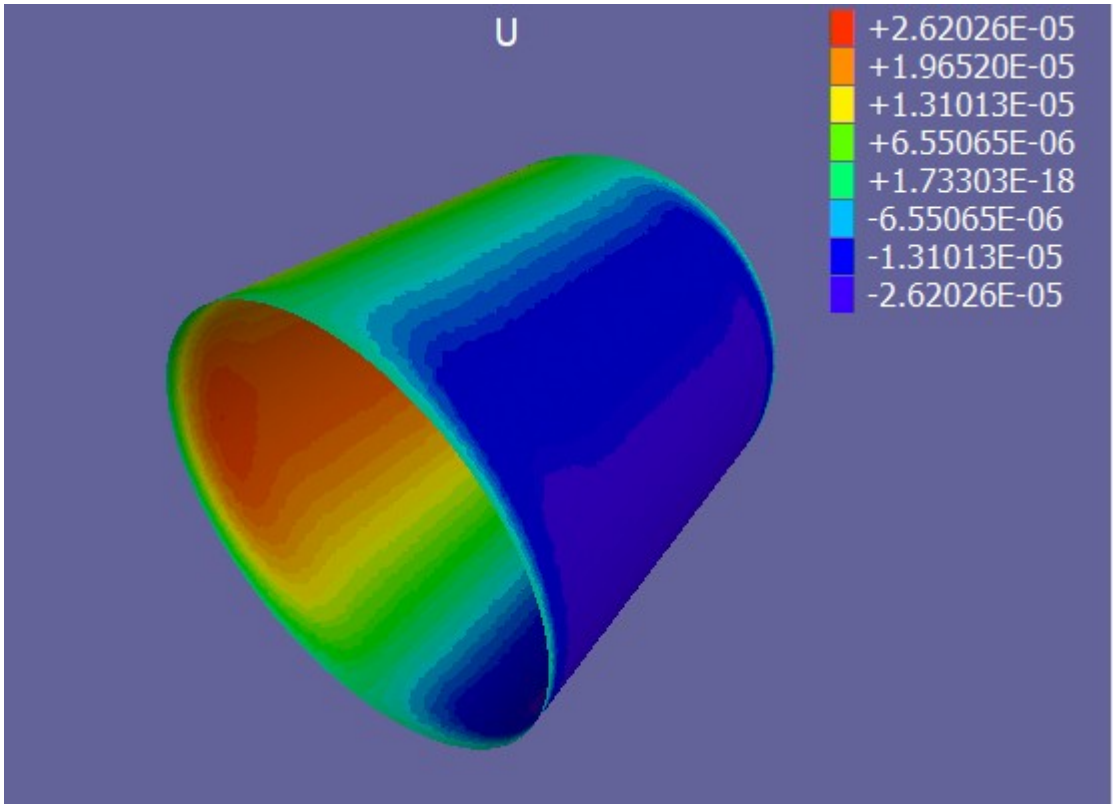

Рис. 4.23 – Розподіл переміщень u при використанні квадратичного СЕ

Розподіл компоненти тензору напружень σ_{xx} по оболонковій конструкції наведено на рис. 4.25.

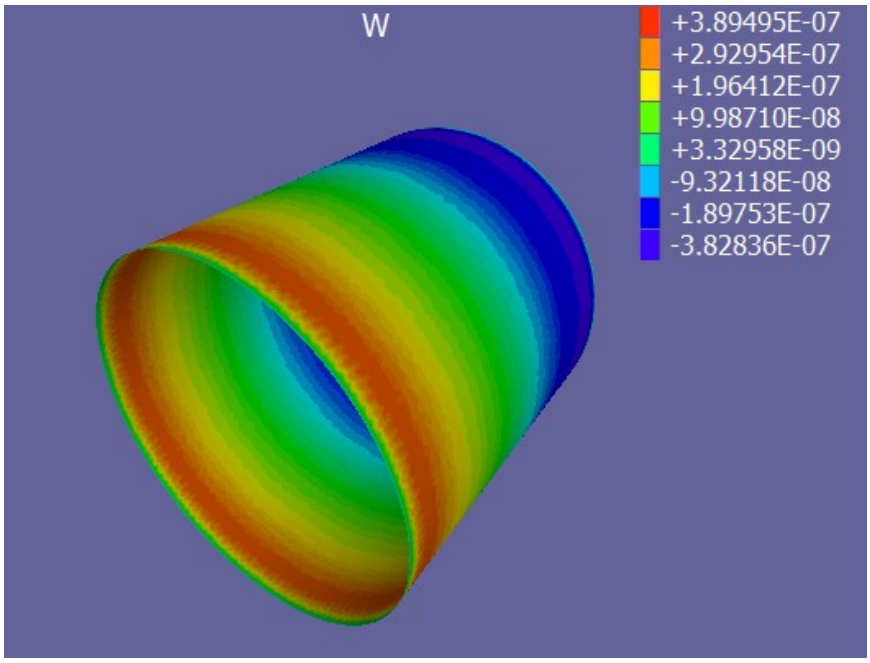

Рис. 4.24 – Розподіл переміщень w при використанні квадратичного СЕ

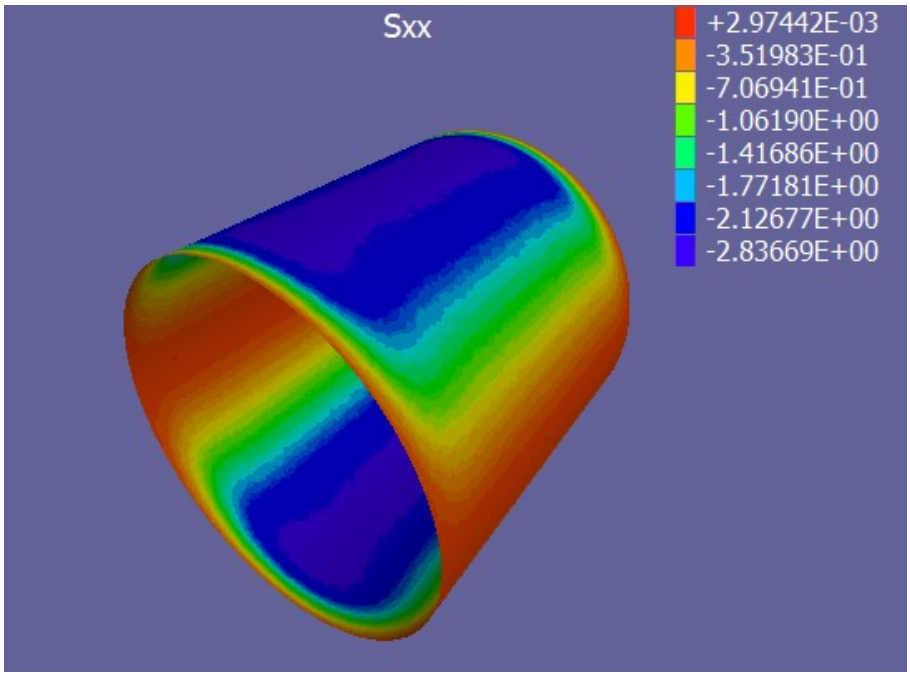

Рис. 4.25 – Розподіл компоненти тензору напружень σ_{xx} по конструкції

4.4 Тривимірні задачі

Ще одним прикладом розв’язання задач із застосуванням бібліотеки PyFEM є розрахунок задачі про трубу, що знаходиться під внутрішнім тиском, у тривимірній постановці.

Для виконання такого розрахунку із застосуванням генератору сітки Netgen [22] була побудована наступна дискретна модель (рис. 4.26), яка складається з 11845 вузлів і 34783 СЕ у формі лінійного тетраедра.

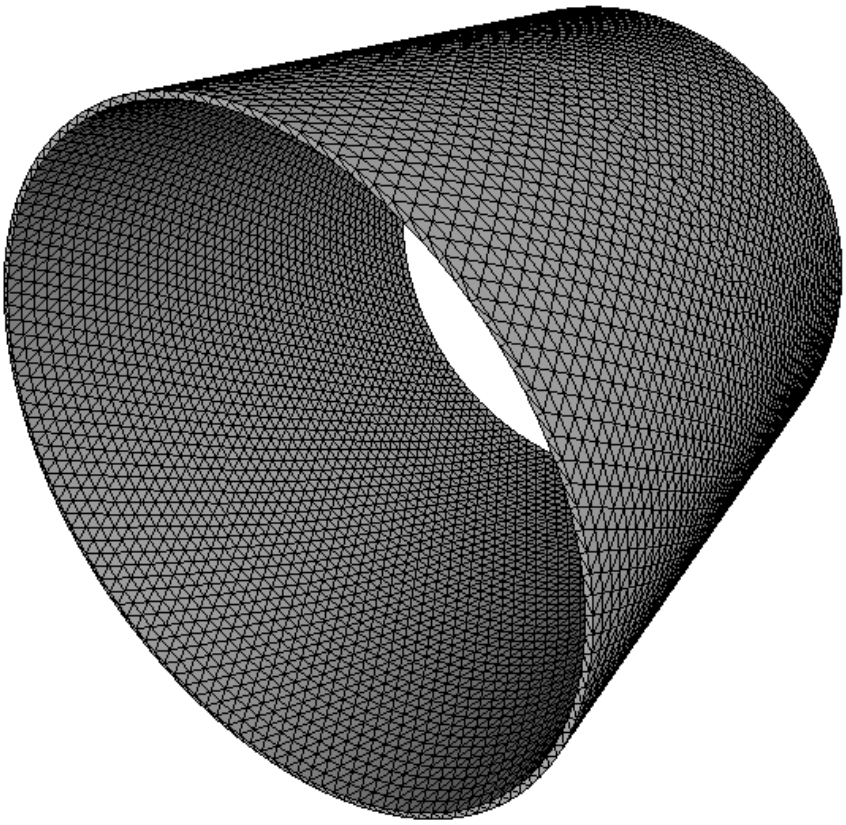

Рис. 4.26 – Дискретна модель труби у тривимірному випадку

Для розрахунку цієї задачі було написано наступну програму.

```
def tube_test(res_name):
    obj = TObject()
    if obj.set_mesh('mesh/tube-solid-test.trpa'):
        obj.set_problem_type('static')
        obj.set_solve_method('direct')
        obj.add_young_modulus('203200')
        obj.add_poisson_ratio('0.27')
        obj.add_boundary_condition(DIR_X | DIR_Y | DIR_Z, '0', 'z == 0 or z == 4.014')
        obj.add_surface_load(DIR_X, '0.05*cos(atan2(y,x))',
                                    '(abs(x**2 + y**2 - 1.99**2) <= 1.0E-3)')
        obj.add_surface_load(DIR_Y, '0.05*sin(atan2(y,x))',
                                    '(abs(x**2 + y**2 - 1.99**2) <= 1.0E-3)')
        if obj.calc():
            # obj.print_result()
            obj.save_result(res_name)
            TPlot(res_name)
            return True
        return False
```

Результат роботи цієї програми наведено на рис. 4.27. Візуалізація розподілу переміщень u по конструкції на рис. 4.28.

```
Object: tube-solid-test
Points: 11845
FE: 34783 - linear tetrahedron (4 nodes)
Assembling global stiffness matrix... 100%
Computation of surface load... 100%
Use of boundary conditions... 100%
Solving of equation system... 100%
Calculation results... 100%
*************** Success! ***************
Lead time 160.148524 sec
```

Рис. 4.27 – Результат розрахунку труби у тривимірному випадку

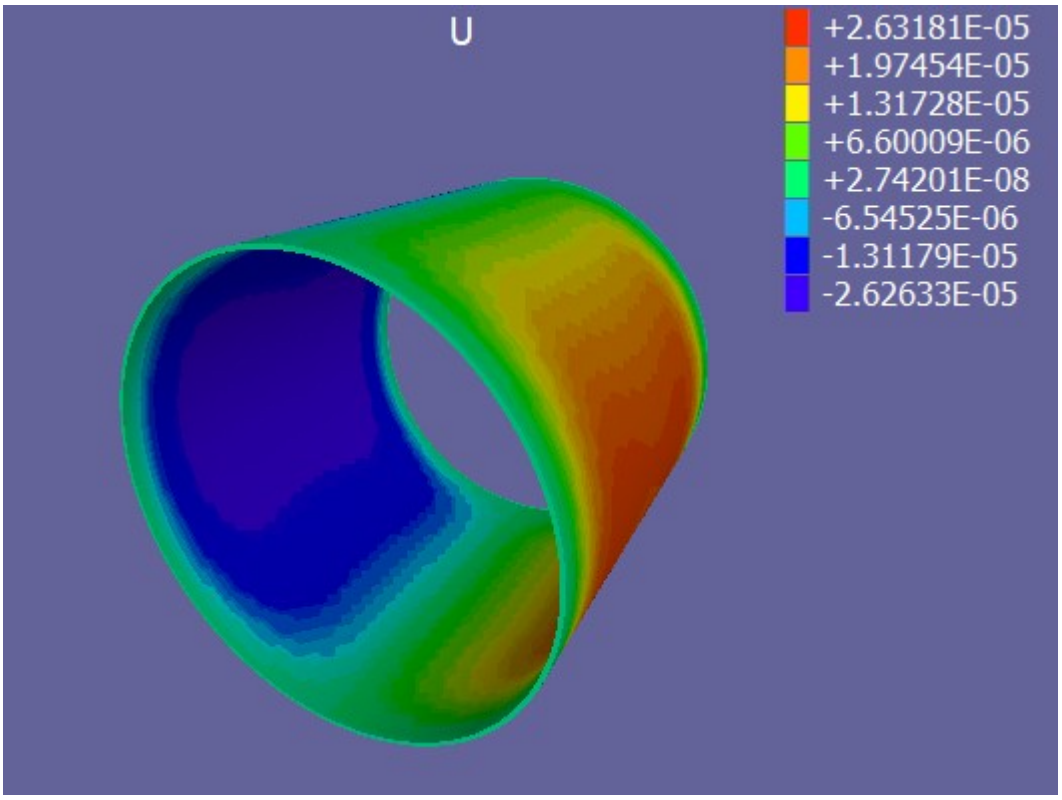

Рис. 4.28 – Розподіл переміщень u по трубі у тривимірній постановці

Відносна похибка триманого результату у порівнянні з використанням оболонкового квадратичного СЕ становить 4%, що може свідчити як про якість розрахунку, так і коректність програмної реалізації МСЕ у PyFEM.

Як ще один числовий приклад розглянемо задачу визначення напружено-деформованого стану лопатки ротора турбіни. Розрахунок проводився для наступних матеріалів лопатки:

а) Titanium Ti-6Al-4V, який має наступні пружні характеристики: модуль Юнга – 1.14E05 МПА, коефіцієнт Пуассона – 0.342;

б) інтерметалідний сплав на основі алюмінідів титану Ti-Al-Nb з фізичними такими характеристиками: модуль Юнга – 0.95E05 МПА, коефіцієнт Пуассона – 0.300.

Розрахунок поводився із використанням дискретної моделі, яка складається з 22897 вузлів і 90663 СЕ у формі лінійного тетраедра (рис. 4.29).

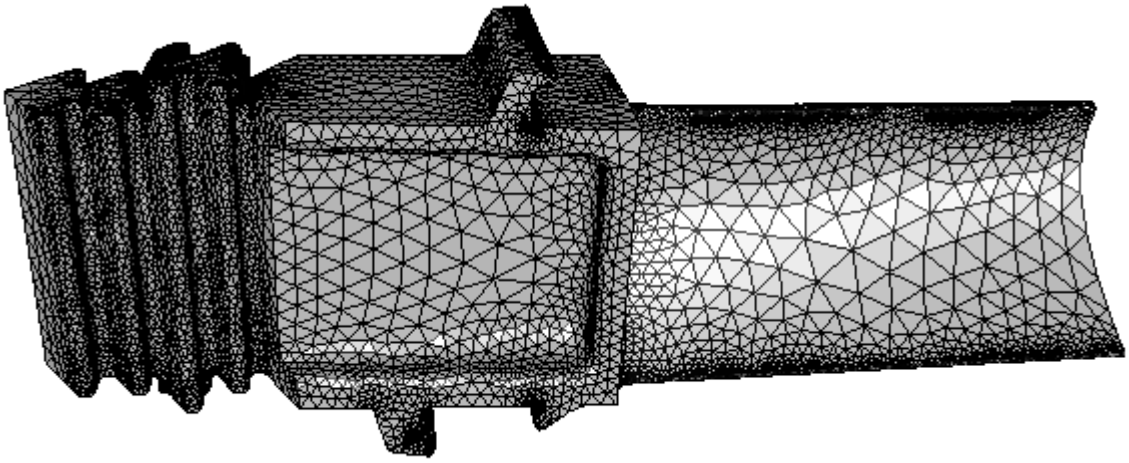

Рис. 4.29 – Дискретна модель лопатки

Для розрахунку лопатки написана наступна програма (варіант б).

```
def blade(res_name):
    obj = TObject()
    if obj.set_mesh('mesh/blade.trpa'):
        obj.set_problem_type('static')
        obj.set_solve_method('direct')
        obj.add_young_modulus('95000')
        obj.add_poisson_ratio('0.3')
        obj.add_boundary_condition(DIR_X | DIR_Y | DIR_Z, '0', 'z < 0')
        obj.add_volume_load(DIR_X | DIR_Y, '-0.05')
        if obj.calc():
            obj.save_result(res_name)
            TPlot(res_name)
            return True
        return False
```

Тук реалізовано розрахунок у лінійно-пружній постановці при об’ємному навантаженні 0,05 МПа. Температурні напруги не враховувалися.

На рис. 4.30, 4.31 наведено розподіл переміщень u по об’єму лопаток для обох варіантів розрахунку.

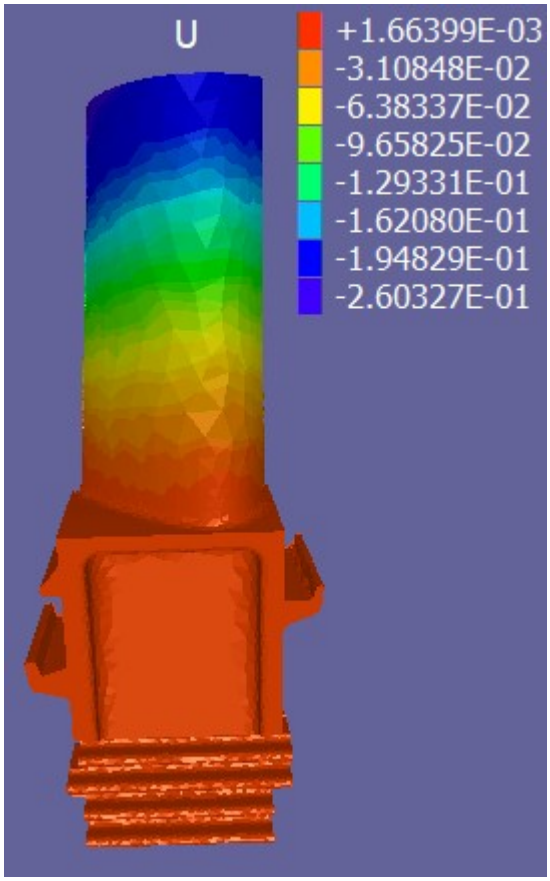

Рис. 4.30 – Titanium Ti-6Al-4V

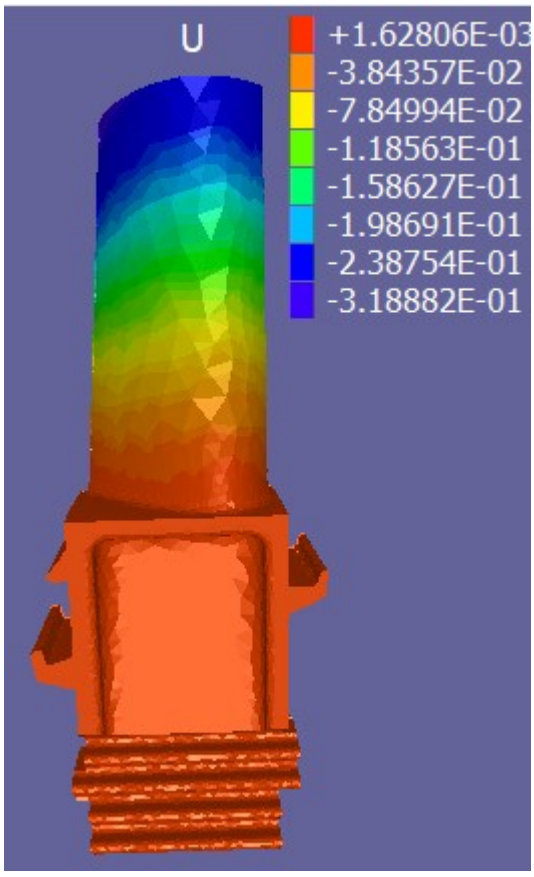

Рис. 4.31 – Інтерметалідний сплав на основі алюмінідів титану Ti-Al-Nb

Окрім наведених у четвертому розділі прикладів розв’язання низки модельних задач також виконувалося дослідження застосування бібліотеки PyFEM для дослідження напружено-деформованого стану інших конструкцій, наприклад, [45, 89].

4.5 Нестаціонарні задачі

Бібліотека PyFEM за рахунок її відкритої архітектури дозволяє легко додавати до неї класи, які реалізують різномінітні алгоритми розв’язання задач спеціальних типів. Так у третьому розділі було описано похіддний від TFEM клас TFEMDynamic, який реалізує розв’язання задач динаміки.

В якості прикладу застосування цього класу було розв’язано задачу про пошук параметрів напружено-деформованого стану двотаврової балки з круговими отворами, яка знаходиться під дією змінного у часі об’ємного навантаження. Дискретна модель балки наведена на рис. 4.32.

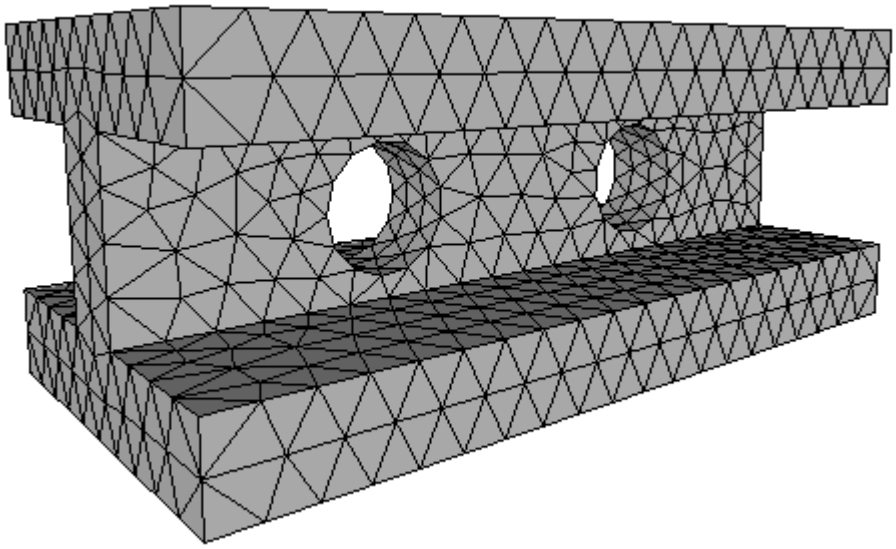

Рис. 4.32 – Дискретна модель двотаврової балки

Задача розв’язувалася при наступних безрозмірних параметрах: модуль Юнга – 6.5E+10, коефіцієнт Пуассона – 0.3, щільність матеріалу – 1.0E+3. На конструкцію діє змінне у часі об’ємне навантаження, яке описується законом – -1.0E+5·cos(t). Розрахунок виконувався на інтервалі часу t від 0.0 до 1.0 з кроком 0.25 при нульових початкових умовах.

Відповідна функція, що описує такий тип розрахунку, має наступний вигляд.

```
def beam_dynamic(res_name):
```

```
obj = TObject()
if obj.set_mesh('mesh/beam.trpa'):
    obj.set_problem_type('dynamic')
    obj.set_solve_method('iterative')
    obj.add_young_modulus('6.5E+10')
    obj.add_poisson_ratio('0.3')
    obj.add_density('1.0E+3')
    obj.set_time(0, 1.0, 0.25)
    obj.add_boundary_condition(DIR_X | DIR_Y | DIR_Z, '0', 'y == 0')
    obj.add_volume_load(DIR_Y, '-1.0E+5*cos(t)')
    obj.add_initial_condition(INIT_U, '0')
    obj.add_initial_condition(INIT_V, '0')
    obj.add_initial_condition(INIT_W, '0')
    obj.add_initial_condition(INIT_UT, '0')
    obj.add_initial_condition(INIT_VT, '0')
    obj.add_initial_condition(INIT_WT, '0')
    obj.add_initial_condition(INIT_UTT, '0')
    obj.add_initial_condition(INIT_VTT, '0')
    obj.add_initial_condition(INIT_WTT, '0')
    if obj.calc():
        # obj.print_result('mesh/' + obj.object_name() + '.res')
        obj.save_result(res_name)
        TPlot(res_name)
        return True
    return False
```

В результаті розрахунку отримано значення основних параметрів напружено-деформованого стану, які змінюються у часі. Так, наприклад, значення компоненти вектора переміщень *v*, яка відповідає у даному випадку напрямку дії навантаження, у початковий і кінцевий моменти часу наведено на рис. 4.33, 4.34.

Висновки до розділу 4

У четвертому розділі наведено приклади застосування бібліотеки PyFEM для розв’язання різних типів пружних задач у статичній та динамічній постановці.

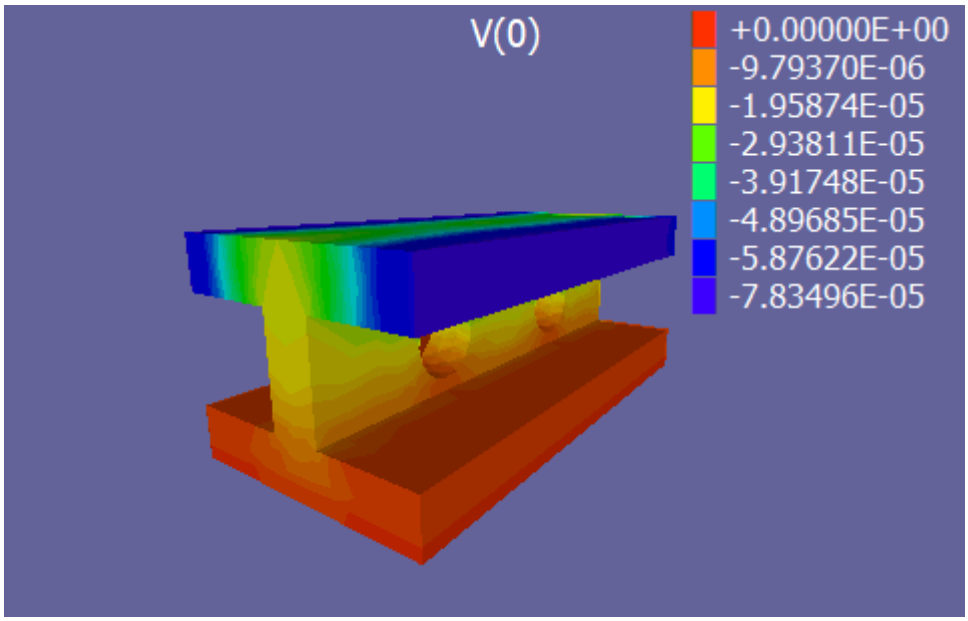

Рис. 4.33 – Розподіл переміщень v при t = 0.00

Для більшості наведених задач виконане порівняння отриманих чисельних результатів їх розв’язання або з відомими аналітичними розв’язками, або з результатами розв’язання цих задач в інших системах скінченно-елементного аналізу. Виявлено, що максимальна відносна похибка не перевищує 5%, що може свідчити про ефективність запропонованого підходу до реалізації відкритої об’єктно-орієнтованої архітектури та якості її програмної реалізації.

Основні наукові і практичні результати четвертого розділу опубліковано в роботах [45, 48, 85, 89].

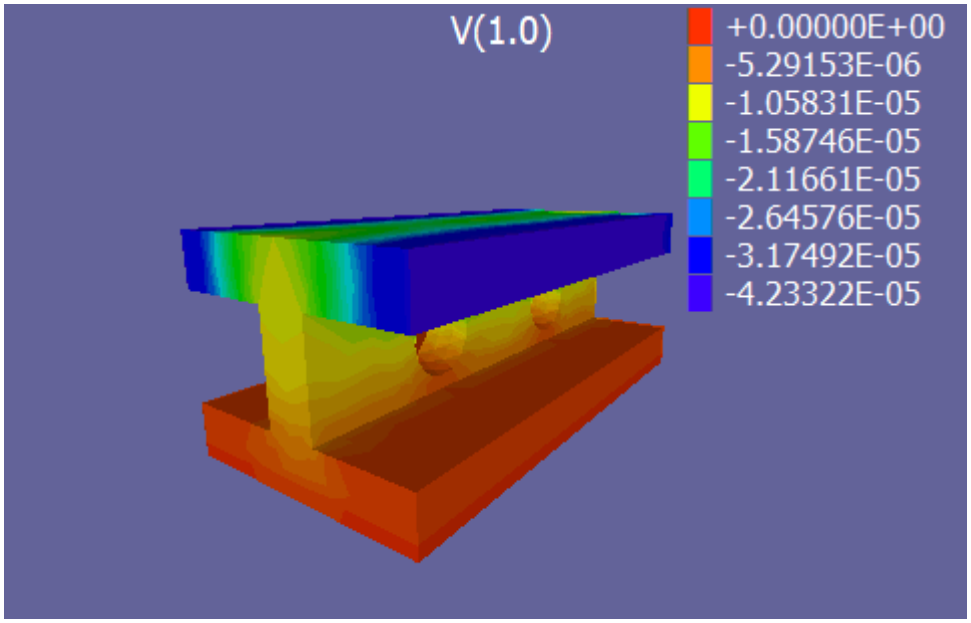

Рис. 4.34 – Розподіл переміщень v при t = 1.00

ВИСНОВКИ

У дисертаційній роботі розв’язано актуальну науково-технічну проблему підвищення ефективності розробки систем скінченно-елементного аналізу крайових задач за рахунок створення відкритої об’єктно-орієнтованої архітектури, яка дозволяє, на відміну від існуючих аналогів, легко вносити корективи у наявні та додавати нові методи розрахунку, що дає можливість істотно підвищити функціональність таких систем.

Отримано наступні наукові результати:

– проведено критичний аналіз наявних систем скінченно-елементного аналізу, а також їх програмної архітектури, який показав необхідність створення відкритої архітектури відповідних програмних засобів, яка б дозволяла полегшити їх супровід і вдосконалення за рахунок можливості додавання нових структурних елементів (класів);

– розроблено відкриту архітектуру системи скінченно-елементного аналізу, яка дозволяє користувачу розширювати її можливості за рахунок використання нових типів скінченних елементів та алгоритмів розрахунку;

– на основі запропонованої архітектури із застосуванням мови програмування Python було розроблено об’єктно-орієнтовану бібліотеку скінченно-елементного аналізу PyFEM;

– виконано низку обчислювальних експериментів, які підтвердили ефективність запропонованих підходів при скінченно-елементному моделюванні.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Zienkiewicz O. C., Taylor R. L., Zhu J. Z. The Finite Element Method: Its Basis and Fundamentals. Sixth edition. Butterworth-Heinemann, 2016. 753 p.

2. Design and Engineering Simulation | SIMULIA – Dassault Systemes. URL: https://www.3ds.com/products-services/simulia/ (дата звернення: 12.07.2022)

3. Engineering Simulation & 3D Design Software | Ansys. URL: https://www.ansys.com/ (дата звернення: 12.07.2022)

4. COMSOL Multiphysics ® Modelling Software. URL: https://www.comsol.com/ (дата звернення: 12.07.2022)

5. The dial.II Finite Element Library. URL: https://www.dealii.org/ (дата звернення: 12.07.2022)

6. Front Page | DIANA FEA. URL: https://dianafea.com/ (дата звернення: 24.07.2022)

7. FreeFEM – An open-source PDE Solver using The Finite Element Method: URL: https://freefem.org/ (дата звернення: 12.07.2022)

8. MATLAB – MathWorks – MATLAB & Simulink. URL: https://www.mathworks.com/products/matlab.html (дата звернення: 29.07.2022)

9. MSC Nastran – Multidisciplinary Structural Analysis. URL: https://www.mscsoftware.com/product/msc-nastran (дата звернення: 19.08.2022)

10. OpenCAD The Programmers Solid 3D CAD Modeller. URL: https://www.openscad.org/ (дата звернення: 12.08.2022)

11. SOLIDWORKS. URL: https://www.solidworks.com/ (дата звернення: 12.08.2022)

12. Top Finite Element Analysis (FEA) Software List, Reviews, Comparison & Price | TEC. URL: https://www3.technologyevaluation.com/sd/category/finite-element-analysis-fea (дата звернення: 12.08.2022)

13. List of finite element software packages. URL: https://en.wikipedia.org/wiki/List_of_finite_element_software_packages (дата звернення: 24.07.2022)

14. qzCAD. URL: https://github.com/qzcad (дата звернення: 02.08.2022)

15. QFEM – The Simple FEM Solver. URL: https://github.com/SeregaGomen/QFEM (дата звернення: 02.08.2022)

16. Киричевский В. В. Метод конечных элементов в механике эластомеров. Киев: Наук. думка, 2002. 653 с.

17. Метод конечных элементов: теория, алгоритмы, реализация / В.А.Толок и др. Киев: Наук. думка, 2003. 316 с.

18. Autodesk. Finite Element Analysis Software (FEA software). URL: https://www.autodesk.com/solutions/finite-element-analysis?geoNavigationPreferredSite=US (дата звернення: 04.08.2022).

19. IEEE. Innovation at work. How the Finite Element Method (FEM) and Finite Element Analysis (FEA) Work Together. URL: https://innovationatwork.ieee.org/how-the-finite-element-method-fem-and-finite-element-analysis-fea-work-together/ (дата звернення: 04.08.2022).

20. Elmer FEM – open source multiphysical simulation software. URL: http://www.elmerfem.org/blog/ (дата звернення: 04.08.2022).

21. FENRIS. A library for advanced finite element computations in Rust. URL: https://crates.io/crates/fenris (дата звернення: 04.08.2022).

22. Netgen/NGSolve. URL: https://ngsolve.org/ (дата звернення: 04.08.2022).

23. Метод конечных элементов в вычислительном комплексе “МІРЕЛА+” / В. В. Киричевский и др. Киев: Наук. думка, 2005. 403 с.

24. fem – the finite element method rust library. URL: https://github.com/SeregaGomen/fem (дата звернення: 06.08.2022).

25. fem_2d – 2D Finite Element Method Toolkit. URL: fem_2d - crates.io: Rust Package Registry (дата звернення: 06.08.2022).

26. Функциональный подход к геометрическому моделированию технических систем / С. В. Чопоров и др. Запорожье: Запорожский национальный университет, 2016. 176 с.

27. Juettler B., Piene R. Geometric modeling and algebraic geometry. London: Springer, 2008. 227 p.

28. Frey P. J.,George P.-L. Mesh Generation Application to Finite Elements. London: ISTE Publishing Company, 2019. 817 p.

29. ANSA – The advanced CAE pre-processing software for complete model build up. URL: https://www.beta-cae.com/ansa.htm (дата звернення: 07.08.2022)

30. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. URL: https://gmsh.info/ (дата звернення: 07.08.2022)

31. GRUMMP Home Page. URL: http://tetra.mech.ubc.ca/GRUMMP/ (дата звернення: 07.08.2022)

32. TetGen – A Quality Tetrahedral Mesh Generator and a 3D Delaunay Triangulator. URL: https://wias-berlin.de/software/index.jsp?id=TetGen&lang=1 (дата звернення: 07.08.2022)

33. SAL – Numerical Analysis – Discrete Methods & Related Tools. URL: http://www.sai.msu.su/sal/B/2/ (дата звернення: 07.08.2022)

34. GetFEM – An open-source finite element library. URL: http://getfem.org/ (дата звернення: 11.08.2022)

35. hp-FEM. URL: https://www.hpfem.org/hermes/ (дата звернення: 11.08.2022)

36. MFEM – Finite Element Discretization Library. URL: https://mfem.org/ (дата звернення: 11.08.2022)

37. OFELI Library – An Object Oriented Finite Element Library. URL: http://ofeli.org/ (дата звернення: 11.08.2022)

38. Open System for Earthquake Engineering Simulation – Home Page. URL: https://opensees.berkeley.edu/ (дата звернення: 11.08.2022)

39. Гоменюк С. И. Объектно-ориентированные модели и методы анализа механических процессов. Никополь: Никопольская коммунальная типография, 2004. 311 с.

40. FEMDS – Fast Emergency Management Dispatching System. URL: http://www.femds.de/ (дата звернення: 14.08.2022)

41. Метод конечных элементов в вычислительном комплексе “МІРЕЛА+” / В. В. Киричевский и др. Киев: Наук. думка, 2005. 403 с.

42. Post-processor for FEM solver "MIRELA". URL: https://github.com/SeregaGomen/MIRELA (дата звернення: 14.08.2022)

43. GNS ANIMATOR 4. URL: https://www.cdh-ag.com/en/software/gns-software/gns-animator-4.html (дата звернення: 14.08.2022)

44. KISSsoft’s FEM Post Processor. URL: https://www.kisssoft.com/en/news-and-events/newsroom/kisssofts-fem-post-processor (дата звернення: 14.08.2022)

45. Морозов Д. М., Юрєчко В. З., Гнєздовський О. В. Скінченно-елементний підхід до визначення напружено-деформованого стану пористого матеріалу засобами Pyton. *Енергоефективність в будівництві та архітектурі: науково-технічний збірник*. 2017. Вип. 9. С. 174-178.

46. Игнатченко М. С., Кудин А. В., Гнездовский А. В. Объектно-ориентированная реализация библиотеки конечно-элементного анализа на языке программирования Python. *Вісник Запорізького національного університету. Фізико-математичні науки*. 2020. № 1. С. 138-147.

47. Игнатченко М. С., Гнездовский А. В. Объектно-ориентированное моделирование задач механики. *Сучасні проблеми машинобудування*. Конференція молодих вчених та спеціалістів: зб. тез доп. Харків: Інститут проблем машинобудування НАН України, 2016. С. 23-24.

48. Гнєздовський О. В. Застосування мови програмування Python для розробки систем скінченно-елементного аналізу. *Актуальні проблеми математики та інформатики*: Збірка тез доповідей Тринадцятої Всеукраїнської,

двадцятої регіональної наукової конференції молодих дослідників (м. Запоріжжя, 17 червня 2022 р.). Запоріжжя: ЗНУ, 2022. С. 8-9.

49. Lanczos C. The Variational Principles of Mechanics. London: Dover, 1986. 464 p.

50. Frey P. J., George P.-L. Mesh Generation application to finite elements. Oxfod: Hermes Science, 2000. 816 p.

51. Daniel S.H. Lo. Finite Element Mesh Generation. Boca Raton: CRC Press, 2015. 618 p.

52. George P. L. Automatic Mesh Generation and Finite Element Computation. London: Elsevier Science, 1996. 190 p.

53. Shewchuk J. R. Delaunay Refinement Mesh Generation. Pittsburgh: Carnegie Mellon University, 1997. 207 p.

54. Chen M., Tucker J. V. Constructive volume geometry // Computer Graphics Forum. 2000. Vol. 19, Iss. 4. P. 281–293.

55. Stroud I. Boundary representation modelling techniques. London: Springer, 2006. 808 P.

56. Рвачев В. Л. Теория R-функций и некоторые ее приложения. Киев: Наук. думка, 1982. 106 с.

57. Pasko A., Adzhiev V., Sourin A. Savchenko V. Function representation in geometric modeling: concepts, implementation and applications. *The visual computer*. 1995. Vol. 11. P. 429-446.

58. Гоменюк С. І., Чопоров С. В., Аль-Атамнех Б. Г. М. Математичне моделювання геометричних об’єктів у паралельних комп’ютерних системах: монографія. Херсон: Видавничий дім “Гельветика”, 2018. 112 с.

59. Чопоров С. В. Дискретные модели форм технических объектов: монография. Херсон: Видавничий дім “Гельветика”, 2018. 324 с.

60. Booch G., Rumbaugh J., Jacobson I. Unified Modeling Language User Guide. London: Addison-Wesley Professional, 2005. 496 p.

61. Lorensen W. E., Cline H. E. Marching Cubes: A high resolution 3D surface construction algorithm. URL: https://dl.acm.org/doi/10.1145/37402.37422 (дата звернення: 04.02.2023)

62. Maple C. Geometric design and space planning using the marching squares and marching cube algorithms. Proc. 2003 Intl. Conf. Geometric Modeling and Graphics. pp. 90-95.

63. Pseudocode standard. URL: http://users.csc.calpoly.edu/~jdalbey/SWE/pdl_std.html (дата звернення: 27.04.2023)

64. Хомченко А. Н. Стандартные серендиповы многочлены и линейчатые поверхности / А. Н. Хомченко, Е. И. ,Литвиненко, И. А. Астионенко // Комп'ютерно- інтегровані технології: освіта, наука, виробництво. № 6. Луцьк: ЛНТУ, 2011. С. 266-269.

65. Healy K. Data Visualization: A Practical Introduction. New Jersey: Princeton University Press, 2018. 296 p.

66. OpenGL - The Industry Standard for High Performance Graphics. URL: https://www.opengl.org/ (дата звернення: 07.10.2023)

67. WebGL Fundamentals. URL: https://webglfundamentals.org/ (дата звернення: 07.10.2023)

68. Божидарник В. В., Сулим Г. Т. Елементи теорії пружності. Львів: Світ, 1994. 560 с.

69. Ясній П. В. Механіка руйнування зварних конструкцій. Тернопіль: ТДТУ, 2006. 100 с.

70. Parr T. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages. London: Pragmatic Bookshelf, 2010. 389 p.

71. Mogensen T. Introduction to Compiler Design. Lëtzebuerg: Springer, 2017. 263 p.

72. Welcome to Python.org. URL: https://www.python.org (дата звернення: 08.09.2023)

73. Морозов Д. Н., Гнездовский А. В., Мыльцев А. М., Толок А. В. Когнитивная компьютерная графика в процессе решения оптимизационных задач математического моделирования. Прикладна геометрія та інженерна графіка. 2010. Вип. 86. С. 112-117.

74. Рефакторинг і патерни проєктування. URL: https://refactoring.guru/uk (дата звернення: 08.09.2023)

75. Beeger R., Züllighoven H., Bäumer D. Object-Oriented Construction Handbook: Developing Application-Oriented Software with the Tools & Materials Approach. Morgan Kaufmann, 2005. 544 p.

76. Coplien J. O. Advanced C++ Programming Styles and Idioms. Programming Styles and Idioms. USA: Addison-Wesley, 1992. 520 p.

77. Mathematical and computer modelling of engineering systems : Collective monograph / In edition by Corresponding Member of the National Academy of Sciences of Ukraine V. S. Hudramovich. Riga, Latvia : “Baltija Publishing”, 2020. 164 p.

78. Alexandrescu A. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, 2001. 366 p.

79. O'Dwyer A. Mastering the C++17 STL. Make full use of the standard library components in C++17. Birmingham: Packt Publishing Ltd, 2017. 384 p.

80. Федорченко А. М. Теоретична механіка. Київ: Вища школа, 1975. 516 с.

81. Lanczos C. The Variational Principles of Mechanics. Dover Publications, 2012. 464 p.

82. Gere J. M., Timoshenko S. P. Mechanics of Materials. Springer US, 1991. 827 p.

83. Segerlind L. J. Applied Finite Element Analysis. Willey, 1984. 440 p.

84. Variation of Parameters. URL: https://mathworld.wolfram.com/VariationofParameters.html (дата звернення: 08.09.2023)

85. Hniezdovskyi O., Kudin O., Belokon Y., Kruglyak D., Ilin S. Designing an object-oriented architecture for the finite element simulation of structural elements. Eastern-European Journal of Enterprise Technologies, 2022, 6(2-120), pp. 78-84 (SCOPUS).

86. Писаренко Г. С., Квітка О. Л., Уманський Е. С. Опір матеріалів. Підручник. Київ: Вища школа, 1993. 655 с.

87. Тимошенко С. П. Курс теории упругости. Киев: Наукова думка, 1972. 508 с.

88. Timoshenko S., Woinowsky-Krieger S., Theory of plates and shells, McGraw-Hill New York, 1959. 580 p.

89. Богуславська А. М., Гребенюк С. М, Морозов Д. М., Гнєздовський О. В. Розрахунок напружено-деформованого стану металевих паль. Вісник Запорізького національного університету. Фізико-математичні науки. 2020. № 2. С. 14-19.

ДОДАТКИ

Додаток А

Список опублікованих праць за темою дисертації

1) Морозов Д. Н., Гнездовский А. В., Мыльцев А. М., Толок А. В. Когнитивная компьютерная графика в процессе решения оптимизационных задач математического моделирования. *Прикладна геометрія та інженерна графіка*. 2010. Вип. 86. С. 112-117.

2) Морозов Д. М., Юрєчко В. З., Гнєздовський О. В. Скінченно-елементний підхід до визначення напружено-деформованого стану пористого матеріалу засобами Pyton. *Енергоефективність в будівництві та архітектурі: науково-технічний збірник*. 2017. Вип. 9. С. 174-178.

3) Игнатченко М. С., Кудин А. В., Гнездовский А. В. Объектно-ориентированная реализация библиотеки конечно-элементного анализа на языке программирования Python. *Вісник Запорізького національного університету. Фізико-математичні науки*. 2020. № 1. С. 138-147.

4) Богуславська А. М., Гребенюк С. М, Морозов Д. М., Гнєздовський О. В. Розрахунок напружено-деформованого стану металевих паль. *Вісник Запорізького національного університету. Фізико-математичні науки*. 2020. № 2. С. 14-19.

5) Игнатченко М. С., Гнездовский А. В. Объектно-ориентированное моделирование задач механики. *Сучасні проблеми машинобудування*. Конференція молодих вчених та спеціалістів: зб. тез доп. Харків: Інститут проблем машинобудування НАН України, 2016. С. 23-24.

6) Гнєздовський О. В. Застосування мови програмування Python для розробки систем скінченно-елементного аналізу. *Актуальні проблеми математики та інформатики*: Збірка тез доповідей Тринадцятої Всеукраїнської, двадцятої регіональної наукової конференції молодих дослідників (м. Запоріжжя, 17 червня 2022 р.). Запоріжжя: ЗНУ, 2022. С. 8-9.

7) Hniezdovskyi O., Kudin O., Belokon Y., Kruglyak D., Ilin S. Designing an object-oriented architecture for the finite element simulation of structural elements. Eastern-European Journal of Enterprise Technologies, 2022, 6(2-120), pp. 78-84 (SCOPUS).

Додаток Б

Реалізація побудови локальних матриць жорсткості, маси та демпфування для трикутних лінійних оболонкових скінчених елементів

```
# Абстрактний ізопараметричний двовимірний скінченний елемент
class TFE2D(TFE1D):
    def __init__(self):
        super().__init__()
        self.freedom = 2

    def _elastic_matrix(self):
        # Матрица упругих свойст
        d = array([
            [1.0, self.m[0], 0.0],
            [self.m[0], 1.0, 0.0],
            [0.0, 0.0, 0.5 * (1.0 - self.m[0])]
        ]) * self.e[0]/(1.0 - self.m[0]**2)
        return d

    # Формування локальної матриці жорсткості
    def generate(self, is_static=True):
        self._check()
        self.K = zeros((self.freedom * self.size, self.freedom * self.size))
        self.load = zeros(self.freedom * self.size)
        if not is_static:
            self.M = zeros((self.freedom * self.size, self.freedom * self.size))
            self.C = zeros((self.freedom * self.size, self.freedom * self.size))
        # Інтегрування за формулою Гауса
        for i in range(len(self._w)):
            # Ізопараметричні функції форми та їх похідні
            # Матриця Якобі
            jacobi = array([
                [sum(self._shape_dxi(i) * self.x[:, 0]), sum(self._shape_dxi(i) * self.x[:, 1])],
                [sum(self._shape_deta(i) * self.x[:, 0]), sum(self._shape_deta(i) * self.x[:, 1])]
            ])
            # Якобіан
            jacobian = det(jacobi)
            inv_jacobi = inv(jacobi)
            shape_dx = inv_jacobi[0, 0] * self._shape_dxi(i) +
                       inv_jacobi[0, 1] * self._shape_deta(i)
            shape_dy = inv_jacobi[1, 0] * self._shape_dxi(i) +
                       inv_jacobi[1, 1] * self._shape_deta(i)
```

```
        # Матриці градієнтів і функцій форм
        b = zeros([3, self.freedom * self.size])
        c = zeros([self.freedom, self.freedom * self.size])
        for j in range(self.size):
            b[0][self.freedom * j + 0] = b[2][self.freedom * j + 1] = shape_dx[j]
            b[1][self.freedom * j + 1] = b[2][self.freedom * j + 0] = shape_dy[j]
            if not is_static:
                c[0][self.freedom * j + 0] = c[1][self.freedom * j + 1] = self._shape(i, j)
        # Обчислення компонентів локальної матриці жорсткості
        self.K += (b.conj().transpose().dot(self._elastic_matrix()).dot(b) * self.thickness *
                   abs(jacobian) * self._w[i])
        if self.dT != 0 and self.alpha != 0:
            t_load = array([self.alpha * self.dT, self.alpha * self.dT, 0])
            self.load += b.conj().transpose().dot(self._elastic_matrix()).dot(t_load) * \
                         abs(jacobian) * self._w[i]
        if not is_static:
            self.M += (c.conj().transpose().dot(c)) * self.thickness * abs(jacobian) * self._w[i] *
                self.density
            self.C += (c.conj().transpose().dot(c)) * self.thickness * abs(jacobian) * self._w[i] *
                self.damping

# Обчислення деформацій та напружень
def calc(self, u):
    res = zeros((6, self.size))
    for i in range(self.size):
        # Матриця градієнтів
        b = zeros([3, self.freedom * self.size])
        for j in range(self.size):
            b[0][j * self.freedom + 0] = b[2][j * self.freedom + 1] = self._dx(i, j)
            b[1][j * self.freedom + 1] = b[2][j * self.freedom + 0] = self._dy(i, j)
        e = b.dot(u)
        s = self._elastic_matrix().dot(e)
        for j in range(3):
            res[j][i] += e[j]
            res[j + 3][i] += s[j]
    return res

@abstractmethod
def _create(self):
    raise NotImplementedError('Method TFE2D._create() is pure virtual')

@abstractmethod
def _dy(self, i, j):
    raise NotImplementedError('Method TFE2D._dy() is pure virtual')

@abstractmethod
def _shape_deta(self, i):
    raise NotImplementedError('Method TFE._shape_deta() is pure virtual')
```

```
# Абстрактный КЭ пластины
class TFEP(TFE2D):
    def __init__(self):
        super().__init__()
        self.freedom = 3

    def _extra_elastic_matrix(self):
        return array([
            [1.0, 0.0],
            [0.0, 1.0],
        ]) * self.e[0]/(2.0 + 2.0 * self.m[0])

    def calc(self, u):
        res = zeros((12, self.size))
        for i in range(self.size):
            # Матрица градиентов
            bm = zeros([3, self.freedom * self.size])
            bp = zeros([2, self.freedom * self.size])
            for j in range(self.size):
                shape = 1 if i == j else 0
                 bm[0][self.freedom * j + 2] = bm[2][self.freedom * j + 1] = bp[0][self.freedom * j + 0]
= self._dx(i, j)
                 bm[1][self.freedom * j + 1] = bm[2][self.freedom * j + 2] = bp[1][self.freedom * j + 0]
= self._dy(i, j)
                bp[0][self.freedom * j + 2] = bp[1][self.freedom * j + 1] = shape
            em = bm.dot(u)
            ep = bp.dot(u)
            sm = self._elastic_matrix().dot(em) * self.thickness * 0.5
            sp = self._extra_elastic_matrix().dot(ep)
            res[0][i] += em[0]  # Exx
            res[1][i] += em[1]  # Eyy
            res[3][i] += em[2]  # Exy
            res[4][i] += ep[0]  # Exz
            res[5][i] += ep[1]  # Eyz
            res[6][i] += sm[0]  # Sxx
            res[7][i] += sm[1]  # Syy
            res[9][i] += sm[2]  # Sxy
            res[10][i] += sp[0]  # Sxz
            res[11][i] += sp[1]  # Syz
        return res

    # Формирование локальной матрицы жесткости
    def generate(self, is_static=True):
        self._check()
        self.K = zeros((self.freedom * self.size, self.freedom * self.size))
        self.load = zeros(self.freedom * self.size)
        if not is_static:
            self.M = zeros((self.freedom * self.size, self.freedom * self.size))
            self.C = zeros((self.freedom * self.size, self.freedom * self.size))
        # Интегрирование по прямоугольнику [-1; 1] x [-1; 1] (по формуле Гаусса)
```

```
        for i in range(len(self._w)):
            # Матрица Якоби
            jacobi = array([
                [sum(self._shape_dxi(i) * self.x[:, 0]), sum(self._shape_dxi(i) * self.x[:, 1])],
                [sum(self._shape_deta(i) * self.x[:, 0]), sum(self._shape_deta(i) * self.x[:, 1])]
            ])
            # Якобиан
            jacobian = det(jacobi)
            inv_jacobi = inv(jacobi)
                        shape_dx = inv_jacobi[0, 0] * self._shape_dxi(i) + inv_jacobi[0, 1] *
self._shape_deta(i)
                        shape_dy = inv_jacobi[1, 0] * self._shape_dxi(i) + inv_jacobi[1, 1] *
self._shape_deta(i)
            # Изопараметрические матрицы градиентов и функций форм
            bm = zeros([3, self.freedom * self.size])
            bp = zeros([2, self.freedom * self.size])
            c = zeros([self.freedom, self.freedom * self.size])
            for j in range(self.size):
                bm[0][self.freedom * j + 2] = bm[2][self.freedom * j + 1] = bp[0][self.freedom * j + 0]
= shape_dx[j]
                bm[1][self.freedom * j + 1] = bm[2][self.freedom * j + 2] = bp[1][self.freedom * j + 0]
= shape_dy[j]
                bp[0][self.freedom * j + 2] = bp[1][self.freedom * j + 1] = self._shape(i, j)
                if not is_static:
                    c[0][self.freedom * j + 0] = c[1][self.freedom * j + 1] = c[2][self.freedom * j + 2] = \
                        self._shape(i, j)
            self.K += (bm.conj().transpose().dot(self._elastic_matrix()).dot(bm) * self.thickness **
3 / 12.0 +
                       bp.conj().transpose().dot(self._extra_elastic_matrix()).
                       dot(bp) * self.thickness * 5.0 / 6.0) * abs(jacobian) * self._w[i]
            if self.dT != 0 and self.alpha != 0:
                # t_load = array([0, 0, 0]) * self.alpha * self.dT
                # t_load1 = array([0, 1]) * self.alpha * self.dT
                # self.load += ((bm.conj().transpose().dot(self._elastic_matrix()).dot(t_load) +
                #                bp.conj().transpose().dot(self._extra_elastic_matrix()).dot(t_load1)) *
                #                abs(jacobian) * self._w[i])
                self.load += array(self.size * [self.e[0] * (1.0 - self.m[0])/(1.0 + self.m[0])/(1.0 - 2.0 *
self.m[0]),
                                                0, 0]) * self.alpha * self.dT * abs(jacobian) * self._w[i]
            if not is_static:
                self.M += (c.conj().transpose().dot(c)) * abs(jacobian) * self._w[i] * self.density *
self.thickness
                self.C += (c.conj().transpose().dot(c)) * abs(jacobian) * self._w[i] * self.damping *
self.thickness

# Абстрактний скінченний елемент оболонки
class TFES(TFEP):
    def __init__(self):
        super().__init__()
        self.freedom = 6
```

```
        self.T = zeros((3, 3))  # Матриця перетворення глобальних координат на локальні
        self.global_x = []      # Глобальні координати СЕ

    def calc(self, u):
        res = zeros((12, self.size))
        # Підготовка матриці перетворення
        m = prepare_transform_matrix(self.size, self.freedom, self.T)
        # Перетворення глобальних переміщень на локальні
        lu = m.dot(u)
        # Обчислення вузлових локальних деформацій та напру
        for i in range(self.size):
            bm = zeros([3, self.freedom * self.size])
            bp = zeros([3, self.freedom * self.size])
            bc = zeros([2, self.freedom * self.size])
            for j in range(self.size):
                shape = 1 if i == j else 0
                bm[0][self.freedom * j + 0] = bm[2][self.freedom * j + 1] = bp[2][self.freedom * j + 4]
= \
                    bp[0][self.freedom * j + 3] = bc[0][self.freedom * j + 2] = self._dx(i, j)
                bm[1][self.freedom * j + 1] = bm[2][self.freedom * j + 0] = bp[1][self.freedom * j + 4]
= \
                    bp[2][self.freedom * j + 3] = bc[1][self.freedom * j + 2] = self._dy(i, j)
                bc[0][self.freedom * j + 3] = bc[1][self.freedom * j + 4] = shape
            dm = bm.dot(lu)
            dp = bp.dot(lu)
            dc = bc.dot(lu)
            sm = self._elastic_matrix().dot(dm)
            sp = self._elastic_matrix().dot(dp) * self.thickness * 0.5
            sc = self._extra_elastic_matrix().dot(dc)

            local_d = array([[dm[0] + dp[0], dm[2] + dp[2], dc[0]],
                             [dm[2] + dp[2], dm[1] + dp[1], dc[1]],
                             [dc[0], dc[1], 0]])
            local_s = array([[sm[0] + sp[0], sm[2] + sp[2], sc[0]],
                             [sm[2] + sp[2], sm[1] + sp[1], sc[1]],
                             [sc[0], sc[1], 0]])
            global_d = self.T.conj().transpose().dot(local_d).dot(self.T)
            global_s = self.T.conj().transpose().dot(local_s).dot(self.T)

            res[0][i] += global_d[0][0]    # Exx
            res[1][i] += global_d[1][1]    # Eyy
            res[2][i] += global_d[2][2]    # Ezz
            res[3][i] += global_d[0][1]    # Exy
            res[4][i] += global_d[0][2]    # Exz
            res[5][i] += global_d[1][2]    # Eyz
            res[6][i] += global_s[0][0]    # Sxx
            res[7][i] += global_s[1][1]    # Syy
            res[8][i] += global_s[2][2]    # Szz
            res[9][i] += global_s[0][1]    # Sxy
            res[10][i] += global_s[0][2]   # Sxz
```

```
            res[11][i] += global_s[1][2]   # Syz
        return res

    def generate(self, is_static=True):
        self._check()
        self.K = zeros((self.freedom * self.size, self.freedom * self.size))
        self.load = zeros(self.freedom * self.size)
        if not is_static:
            self.M = zeros((self.freedom * self.size, self.freedom * self.size))
            self.C = zeros((self.freedom * self.size, self.freedom * self.size))
        # Інтегрування за трикутником [0,0]-[1,0]-[0,1] (за формулою Гауса)
        for i in range(len(self._w)):
            # Ізопараметричні функції форми та їх похідні
            # Матриця Якобі
            jacobi = array([
                [sum(self._shape_dxi(i) * self.x[:, 0]), sum(self._shape_dxi(i) * self.x[:, 1])],
                [sum(self._shape_deta(i) * self.x[:, 0]), sum(self._shape_deta(i) * self.x[:, 1])]
            ])
            # Якобіан
            jacobian = det(jacobi)
            inv_jacobi = inv(jacobi)
                        shape_dx = inv_jacobi[0, 0] * self._shape_dxi(i) + inv_jacobi[0, 1] *
self._shape_deta(i)
                        shape_dy = inv_jacobi[1, 0] * self._shape_dxi(i) + inv_jacobi[1, 1] *
self._shape_deta(i)
            # Матриці градієнтів і функцій форми
            bm = zeros([3, self.freedom * self.size])
            bp = zeros([3, self.freedom * self.size])
            bc = zeros([2, self.freedom * self.size])
            c = zeros([self.freedom, self.freedom * self.size])
            for j in range(self.size):
                bm[0][self.freedom * j + 0] = bm[2][self.freedom * j + 1] = bp[0][self.freedom*j+3] = \
                    bp[2][self.freedom * j + 4] = bc[0][self.freedom * j + 2] = shape_dx[j]
                bm[1][self.freedom * j + 1] = bm[2][self.freedom * j + 0] = bp[1][self.freedom*j+4] = \
                    bp[2][self.freedom * j + 3] = bc[1][self.freedom * j + 2] = shape_dy[j]
                bc[0][self.freedom * j + 3] = bc[1][self.freedom * j + 4] = self._shape(i, j)
                if not is_static:
                    c[0][self.freedom * j + 0] = c[1][self.freedom * j + 1] = c[2][self.freedom * j + 2] = \
                        c[3][self.freedom * j + 3] = c[4][self.freedom * j + 4] = c[5][self.freedom*j+5] = \
                        self._shape(i, j)
            # Обчислення компонентів локальної матриці жорсткості
            self.K += (bm.conj().transpose().dot(self._elastic_matrix()).dot(bm) * self.thickness +
                       bp.conj().transpose().dot(self._elastic_matrix()).dot(bp) * self.thickness ** 3
                       / 12.0 +
                       bc.conj().transpose().dot(self._extra_elastic_matrix()).dot(bc) * self.thickness *
                       5 / 6) * abs(jacobian) * self._w[i]
            # Обчислення стовпця навантаження
            if self.dT != 0 and self.alpha != 0:
                t_load = array([1, 1, 0]) * self.alpha * self.dT
                self.load += ((bm.conj().transpose().dot(self._elastic_matrix()).dot(t_load)) *
```

```
                        abs(jacobian) * self._w[i])
            if not is_static:
                self.M += (c.conj().transpose().dot(c)) * abs(jacobian) * self._w[i] * self.density *
                        self.thickness
                self.C += (c.conj().transpose().dot(c)) * abs(jacobian) * self._w[i] * self.damping *
                        self.thickness

        # Пошук максимального діагонального елемента
        singular = 0
        for i in range(len(self.K)):
            if self.K[i][i] > singular:
                singular = self.K[i][i]
        singular *= 1.0E-3

        # Усунення сингулярності
        for i in range(self.size):
            self.K[self.freedom * (i + 1) - 1][self.freedom * (i + 1) - 1] = singular
            if not is_static:
                self.M[self.freedom * (i + 1) - 1][self.freedom * (i + 1) - 1] = \
                    self.C[self.freedom * (i + 1) - 1][self.freedom * (i + 1) - 1] = singular

        # Підготовка матриці перетворення
        m = prepare_transform_matrix(self.size, self.freedom, self.T)
        # Перетворення з локальних координат на глобальні
        self.K = m.conj().transpose().dot(self.K).dot(m)
        self.load = m.conj().transpose().dot(self.load)
        if not is_static:
            self.M = m.conj().transpose().dot(self.M).dot(m)
            self.C = m.conj().transpose().dot(self.C).dot(m)

# Лінійний (трихузловий) трикутний СЕ
class TFE2D3(TFE2D):
    def __init__(self):
        super().__init__()
        self.size = 3
        self._xi = [0, 1 / 2, 1 / 2]
        self._eta = [1 / 2, 0, 1 / 2]
        self._w = [1 / 6, 1 / 6, 1 / 6]

    def _create(self):
        det0 = self.x[2][1] * self.x[1][0] - self.x[2][1] * self.x[0][0] - self.x[0][1] * self.x[1][0] - \
               self.x[1][1] * self.x[2][0] + self.x[1][1] * self.x[0][0] + self.x[0][1] * self.x[2][0]
        if math.fabs(det0) < eps:
            raise TException('incorrect_fe_err')
        index = [[2, 1], [0, 2], [1, 0]]
        self.a = zeros((self.size, self.size))
        for i in range(self.size):
            det1 = self.x[index[i][0]][1] * self.x[index[i][1]][0] - self.x[index[i][1]][1] *
                    self.x[index[i][0]][0]
            det2 = self.x[index[i][1]][1] - self.x[index[i][0]][1]
```

```
            det3 = self.x[index[i][0]][0] - self.x[index[i][1]][0]
            self.a[i][0] = det1/det0
            self.a[i][1] = det2/det0
            self.a[i][2] = det3/det0

    # Похідні від функцій форм
    def _dx(self, i, j):
        return self.a[j][1]

    def _dy(self, i, j):
        return self.a[j][2]

    # Ізопараметричні функції форми та їх похідні
    def _shape(self, i, j):
        return array([1 - self._xi[i] - self._eta[i], self._xi[i], self._eta[i]])[j]

    def _shape_dxi(self, i):
        return array([-1, 1, 0])

    def _shape_deta(self, i):
        return array([-1, 0, 1])

# Трикутний СЕ пластини
class TFE2D3P(TFEP, TFE2D3):
    def __init__(self):
        super().__init__()

# Трикутний СЕ оболонки
class TFE2D3S(TFES, TFE2D3P):
    def __init__(self):
        super().__init__()
        self.global_x = zeros((3, 3))

    def _create(self):
        self.T = create_transform_matrix(self.x)
        self.global_x = self.x
        self.x = array([self.T.dot(self.x[0, :]), self.T.dot(self.x[1, :]), self.T.dot(self.x[2, :])])
        TFE2D3._create(self)
```