МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет прикладної математики та інформатики
(повне найменування назва факультету)

Кафедра програмування
(повна назва кафедри)

ДИПЛОМНА РОБОТА

Розв’язування спектральних задач модифікованим методом послідовних наближень

Виконав: студент групи ПМІ-41
спеціальності 122 – комп'ютерні науки
(шифр і назва спеціальності)

Дяківський Д.Д.
(підпис) (прізвище та ініціали)

Керівник Ярошко С.А.
(підпис) (прізвище та ініціали)

Рецензент
(підпис) (прізвище та ініціали)

2023

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет прикладної математики та інформатики

Кафедра програмування

Спеціальність 122 Комп'ютерні науки

«ЗАТВЕРДЖУЮ»
Завідувач кафедри Ярошко С.А.

"_ "_____ 2022 року

З А В Д А Н Н Я

НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Дяківському Дмитру Дмитровичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Розв'язування спектральних задач модифікованим методом послідовних наближень»

керівник роботи Ярошко Сергій Адамович, доцент, к. ф.-м. н. ,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені Вченою радою факультету від "13"_вересня 2022 року №_15_.

2. Строк подання студентом роботи 12 червня 2023 року

3. Вихідні дані до роботи літературні джерела, інтернет-ресурси, постановка задачі, наукові статті

4. Зміст дипломної роботи (перелік питань, які потрібно розробити)

1. Дослідити модифікований метод послідовних наближень для пошуку власних значень та власних функцій;

2. Розробити програму, яка реалізовує пошук власних пар;

3. Розробити зручний веб-додаток для відображення результатів обчислень;

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Графіки отриманих власних функцій

6. Консультанти розділів роботи

Розділ	**Прізвище, ініціали та посада консультанта**	**Підпис, дата**	
		завдання видав	завдання прийняв

7. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	**Назва етапів дипломної роботи**	**Строк виконання етапів роботи**	**Примітка**
1	Ознайомлення з предметною областю	05.09.2022 – 10.09.2022	*виконано*
2	Аналіз аналогів	10.09.2022 – 13.09.2022	*виконано*
3	Написання специфікації вимог	13.09.2022 – 15.09.2022	*виконано*
4	Вибір технологій	16.09.2022 – 21.09.2022	*виконано*
5	Побудова архітектури back-end частини веб-додатку	21.09.2022 – 27.09.2022	*виконано*
6	Розробка back-end частини	27.09.2022 – 11.02.2023	*виконано*
7	Побудова архітектури front-end частини веб-додатку	14.02.2023 – 21.02.2023	*виконано*
8	Розробка front-end частини	22.02.2023 – 15.05.2023	*виконано*
9	Оформлення дипломної роботи	16.05.2023 – 08.06.2023	*виконано*
10	Подання дипломної роботи	12.06.2023	*виконано*

Студент _____ Дяківський Д.Д.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Ярошко С.А.
(підпис) (прізвище та ініціали)

ЗМІСТ

АВТОРЕФЕРАТ

Дипломна робота присвячена розробці веб-додатку розв’язування спектральних задач за допомогою модифікованого методу послідовних наближень, а саме для обчислення характеристичних чисел і відповідних їм власних функцій лінійного цілком неперервного оператора, Додаток повинен мати легкий, гнучкий та зручний інтерфейс, зрозумілий формат отриманих результатів та можливість їх збереження.

У першому розділі проаналізовано постановку спектральних задач та математичне обґрунтування їх розв’язку за допомогою модифікованого методу послідовних наближень, розглянуто його алгоритм.

У другому розділі описано пошук характеристичних чисел та побудову відповідних їм власних функцій за допомогою Python-програми та бібліотек для математичних обчислень, уточнення характеристичних чисел методом Ньютона. А також програмну реалізацію веб-додатку з використанням серверної технології ASP.NET Core Web API в поєднанні з Python.NET та технології для створення інтерфейсу користувача React.

Представлено функціональність веб-додатку та приклади його використання.

ВСТУП

При дослідженні теоретичних питань математичної фізики, механіки, хімії та інших наук, їх прикладне використання та застосування, виникають задачі на власні значення. Зокрема, у механіці при дослідженні найрізноманітніших процесів коливань власні значення характеризують періоди коливань, частоти, кутову швидкість та її критичні значення тощо. В архітектурі важливо, щоб власні частоти споруд, що проектуються та будуються, не співпадали з частотами тектонічних коливань земної поверхні, коливань, що виникають під дією вітру, течій. Для отримання інформації про компоненти газової суміші, склад речовини методом спектроскопії використовують власні (використовується в хімічній промисловості, астрономії, біохімічних дослідженнях, криміналістиці та ін.) . В задачах синтезу випромінюючих систем та гідродинамічної стійкості розв'язання проблеми власних значень дає можливість ще на стадії проектування вибирати оптимальні характеристики систем.

Даний перелік можна продовжити. І з розвитком науки та технічної думки цей перелік буде тільки рости, а отже проблема обчислення власних значень і власних функцій є цікавою для дослідників на даний час і не втратить своєї актуальності в майбутньому.

Лінійні задачі на власні значення досить повно вивчені, існують теоретичні напрацювання, розроблено і обґрунтовано ряд методів знаходження власних значень, оцінку їх точності.

Менше вивчені узагальнені лінійні та нелінійні спектральні задачі, зокрема задачі з цілком неперервними операторами. Одним із методів обчислення характеристичних чисел і відповідних їм власних функцій лінійного цілком неперервного оператора, що діє у нормованому функціональному просторі є модифікований метод послідовних наближень (далі — ММПН). Це ітераційний метод, за алгоритмом якого обчислюють послідовні ітерації оператором деякої початкової функції та опрацьовують на кожному кроці всі обчислені функції, що дозволяє, на відміну від інших методів, за невелику кількість кроків отримувати не

тільки перше, але й наступні характеристичні числа оператора і будувати відповідні їм власні функції, особливо ефективний у випадках, коли спектр оператора містить групу характеристичних чисел, близьких за абсолютною величиною до першого.

1. ПОСТАНОВКА ЗАДАЧІ НА ВЛАСНІ ЗНАЧЕННЯ ТА ТЕОРЕТИЧНЕ ОБҐРУНТУВАННЯ МОДИФІКОВАНОГО МЕТОДУ ПОСЛІДОВНИХ НАБЛИЖЕНЬ

Завданням дипломної роботи було дослідити модифікований метод послідовних наближень для пошуку власних значень та власних функцій у деякому нормованому функціональному просторі, на його основі написати програму, яка реалізує пошук власних значень та власних функцій, розробити зручний веб-додаток для введення вхідних даних та інформативного і зрозумілого відображення результатів. Роботу веб-додатку випробувати на конкретних прикладах.

1.1. Математична постановка задачі

Задача на власні значення має вигляд

$$Au = \lambda u, \tag{1.1}$$

де A – лінійний цілком неперервний оператор, що діє у нормованому функціональному просторі E, а його власні функції утворюють базу у цьому просторі. Спектр цього лінійного цілком неперервного оператора дискретний і складається із скінченної або зліченної послідовності власних значень $\lambda_1, \lambda_2, \ldots, \lambda_n, \ldots$, які можуть мати точку скупчення у нулі. Характеристичні числа $\mu_n=1/\lambda_n$ такого оператора становлять скінченну або зліченну послідовність із точкою скупчення у нескінченності, для яких існує ціла функція, яка називається характеристичним рядом задачі (1.1), нулями якої є ці і лише ці числа ([8], стор. 255):

$$F(\mu) = \sum_{j=0}^{\infty} c_j \, \mu^j \tag{1.2}$$

Тоді задачу (1.1) можна переписати у вигляді

$$u - \mu Au = 0 \tag{1.3},$$

де $\mu=1/\lambda$.

Введемо допоміжне рівняння

$$u - \mu Au = v_0 F(\mu) \qquad (1.4),$$

де v_0 — деяка ненульова функція з простору E, вибір якої залежить від конкретної задачі та вирішується окремо. Рівняння (1.4) при довільних значеннях μ та виборі функції v_0 завжди має розв'язок (теорема Фредгольма, [23], стор. 537). Дійсно, якщо μ – регулярне значення оператора A, то $F(\mu) \neq 0$. Тоді права частина рівняння (1.4) відмінна від нуля і дане рівняння має єдиний ненульовий розв'язок. При співпадінні μ з одним із характеристичних чисел оператора A ($\mu = \mu_n$) матимемо $F(\mu) = 0$, рівняння (1.4) вироджується до однорідного рівняння (1.3), ненульовими розв'язками якого є власні функцій u_n нашого оператора A.

Оскільки рівняння (1.4) залежить від μ, то його розв'язок можна шукати у вигляді

$$u = \sum_{m=0}^{\infty} Z_m \, \mu^m \qquad (1.5)$$

Підставивши у рівняння (1.4) його розв'язки (1.5) та прирівнявши коефіцієнти при однакових степенях μ отримаємо рекурентні співвідношення для знаходження функції Z_m:

$$Z_0 = c_0 \, v_0, \;\; Z_m = c_m v_m + A \, Z_{m-1}, \;\; m = 1, 2, 3, \ldots .$$

Якщо використати позначення

$$v_j = A v_{j-1} = A^{j-1} v_0 \, , \;\; j = 1, 2, 3, \ldots ,$$

то функцію Z_m можна записати у вигляді лінійної комбінації послідовних ітерацій v_j:

$$Z_m = \sum_{j=0}^{\infty} c_j \, n^{\;\; m-j} \quad , \;\; m = 0, 1, 2, \ldots . \qquad (1.6)$$

Встановимо умови знаходження коефіцієнтів c_j.

Для оператора A, який діє у скінченовимірному просторі E^N, його спектр складається із N власних значень λ_n і відповідно характеристичних чисел μ_n. Тоді ряд (1.2) вироджується у поліном N-го степеня, а розв'язок (1.5) буде поліномом $(N-1)$-го степеня по степенях μ. Оскільки $Z_N \equiv 0$, то (1.6) набуде вигляду

$$c_0 v_N + c_1 v_{N-1} + \ldots + c_N v_0 \equiv 0. \qquad (1.7)$$

Тотожність (1.7) є функціональним рівнянням для знаходження коефіцієнтів c_j, де $j=0, 1, ..., N$, характеристичного полінома, при чому одне із c_j рівне одиниці (наприклад, $c_N=1$).

Якщо оператор A діє у нескінченовимірному просторі, то умови для коефіцієнтів c_j із формули (1.7) описані у теоремах 1.1, 1.2.

Після обчислення коефіцієнтів c_j знаходимо характеристичні числа μ_n як корені $F(\mu)$ та власні функції u_n із формули (1.5) при $\mu=\mu_n$.

1.2. Теоретичне обґрунтування

Теорема 1.1. *Нехай всі характеристичні числа μ_n лінійного цілком неперервного оператора A: $E \to E$ є простими, μ_1 – найменше за модулем характеристичне число і $|\mu_1| \geq 1$, а для коефіцієнтів b_n розвинення*

$$v_0(x) = \sum_{n=1}^{\infty} b_n u_n(x) \tag{1.8}$$

початкової функції $v_0 \in E$ за його власними функціями u_n, ($\|u_n\|=1$, $n=1, 2, ...$) виконується умова

$$\sum_{n=1}^{\infty} |b_n| = K < \infty \tag{1.9}$$

Якщо коефіцієнти c_j, $j=0, 1, ..., m$, у визначені (1.6) функції Z_m є коефіцієнтами характеристичного ряду (1.2), то виконується умова

$$\lim_{m \to \infty} \|Z_m\| = 0. \tag{1.10}$$

([1], стор. 12)

Доведення цієї і наступної теореми розглядати не будемо, оскільки вони не є предметом дослідження нашої дипломної роботи.

Умова (1.10) для коефіцієнтів c_j лінійної комбінації (1.7) є необхідною. Сформулюємо достатню умову.

Теорема 1.2. *Нехай усі характеристичні числа μ_n лінійного цілком неперервного оператора $A: E \to E$ є простими, і всі коефіцієнти b_n у розвиненні (1.8) початкової функції $v_0 \in E$ за його власними функціями u_n, ($\|u_n\|=1$, n=1, 2, ...) відмінні від нуля. Якщо*

$$\lim_{m\to\infty} \frac{\|Z_m\|}{|c_m|} = 0, \qquad (1.11)$$

то числа , c_j, j=0, 1, ..., m, що входять у визначення (1.7) функцій Z_m, будуть коефіцієнтами деякої цілої функції, яка перетворюється в нуль на μ_n. Якщо при цьому побудована ціла функція $F(\mu)$ не має сторонніх коренів, що співпадають із μ_n, то для $\mu_n = \mu$ ряд (1.5) збігається до $a_n u_n$, де $a_n \neq 0$.

Навпаки, якщо всі характеристичні числа μ_n оператора A дійсні додатні і, починаючи з деякого номера n_0, зростають не повільніше, ніж Cn^p, p>1, для коефіцієнтів ряду (1.8) виконується (1.9), а коефіцієнти c_j, j=1, 2, ..., m, у визначенні (1.7) функцій Z_m є коефіцієнтами характеристичного ряду (1.2), то виконується умова (1.11). ([1], стор. 13-14)

Умови теореми 1.2 не виключають того, що у функції $F(\mu)$ не існують сторонні корені. Для виявлення сторонніх коренів можна розглянути два випадки.

У першому випадку припустимо, що μ^* є стороннім коренем (μ^* може бути кратним), при чому $\mu^* \neq \mu_n$, для довільних n. Тоді ряд (1.5) буде збігатися і для $\mu=\mu^*$, але тоді він буде розв'язком однорідної задачі (1.3) на регулярному значенні μ, тобто розв'язок буде тотожній 0 ([1], стор. 9).

У другому випадку μ^* буде співпадати із одним із характеристичних чисел μ_k. У такому разі сума $a_k u_k$ із формули (1.5) також буде тотожня 0.

Щоб розпізнати ці два випадки (перший — кратне μ^*, та другий — $\mu^*=\mu_k$) та для другого випадку знайти μ_k та u_k, необхідно використати іншу початкову функцію v_0 та розв'язати задачу повторно.

Умова (1.9) є умовою необхідності і її перевіряти не має потреби.

В той же час у теоремі 1.2 вимагається, щоб його характеристичні числа оператора *A* були простими, та існують обмеження на коефіцієнти b_n розкладу

початкової функції $v_0(x)$ за власними функціями оператора A. Якщо ж у розвиненні (1.8) один або декілька коефіцієнтів b_n дорівнюють нулю, то послідовні ітерації (1.6) не містять характеристичного числа μ_n та власної функції u_n і обчислити їх за допомогою ММПН ми їх не зможемо.

Теореми 1.1 та 1.2 можуть також бути використані для знаходження коефіцієнтів c_j і для скінченновимірного випадку, якщо розмірність N оператора A надто велика і розв'язати систему (1.7) на практиці неможливо або складно.

1.3. Схема алгоритму числової реалізації ММПН

Введемо позначення

$$Y_m = \frac{Z_m}{c_m} \quad .$$

Побудуємо ітераційний алгоритм для розв'язування задачі (1.3) на основі теорем 1.1 та 1.2 з використанням послідовних ітерацій v_j, де $j=1, 2, \ldots$. Наш алгоритм за допомогою вибору коефіцієнтів c_j у формулі (1.6) повинен забезпечити збіжність $\| \Psi_m \|$ до нуля при $m \to \infty$. Оскільки для кожної ітерації m значення c_j можна вибирати різні, то позначимо їх $c_j^{(m)}$. Незалежно, чи збіжна послідовність $c_j^{(m)}$, $m=j, j+1, \ldots$ до коефіцієнтів c_j характеристичного ряду заданого формулою (1.2) матимемо, що μ_n є граничною точкою для $\mu_n^{(m)}$, тобто

$\lim_{m\to\infty} \mu_n^{(m)} = \mu_n$, де $\mu_n^{(m)}$ — корінь рівняння $F_m^{(m)}(\mu) = 0,$ при чому

$$F_m^{(m)}(\mu) = \frac{1}{c_m^{(m)}} \sum_{j=0}^{m} c_j^{(m)} \mu^j.$$

Після знаходження $c_j^{(m)}$ та наближення характеристичних чисел $\mu_n^{(m)}$ можна за формулою (1.6) знайти функції Z_j, $j=0, 1, \ldots, m$, замінивши c_j на $c_j^{(m)}$. Тоді функції u_n, наближені до власних, можна визначити за формулами

$$u_{\mathrm{n}}^{(\mathrm{m})} = \sum_{j=0}^{m} Z_j \mu_j^{(m)j} \,, n = 1, 2, \ldots, m. \qquad (1.12)$$

У формулі (1.12) можна перегрупувати доданки і зібрати коефіцієнти при однакових ітераціях ν_j, отримавши формулу

$$u_{\mathrm{n}}^{(\mathrm{m})} = \sum_{j=0}^{m} \mathrm{p}_{n,j}^{(m)} \nu_j \,, n = 1, 2, \dots, m. \tag{1.13}$$

Розглянемо $\mathrm{p}_{n,0}^{(m)}$, де $n=1, 2, \dots, m$. Матимемо

$$\mathrm{p}_{n,0}^{(m)} = \frac{c_0^{(m)}}{c_m^{(m)}} + \frac{c_1^{(m)}}{c_m^{(m)}}\mu_m + \dots + \frac{c_{m-1}^{(m)}}{c_m^{(m)}}\mu_m^{m-1} + \mu_m^m = 0, \quad n = 1, 2, \dots, m.$$

Тоді у формулі (1.13) доданки, що містять ν_0, можна відкинути. Це означає, що для обчислення $u_n^{(m)}$ для початку обчислень можна вибирати функцію ν_0 в залежності від задачі, що розглядаємо (наприклад, може бути відомо, що власні функції u_n є парними або періодичними чи володіють якимись іншими властивостями).

Узагальнивши вище сказане можна сформулювати остаточну схему алгоритму для m-ого кроку так.

1. Обчислюємо $\nu_m = A\nu_{m-1}$.

2. Для забезпечення виконання умови $\| \Psi_m \| \to 0$ при $m \to \infty$, розглянемо задачу $\| \Psi_m^{(m)} \|^2 \to min$, де функцію $\Psi_m^{(m)}$ можна записати у вигляді

$$\Psi_m^{(m)} = \check{c}_0^{(m)} \nu_m + \check{c}_1^{(m)} \nu_{m-1} + \dots + \check{c}_{m-1}^{(m)} \nu_1 + \nu_0,$$

де $\check{c}_j^{(m)} = c_j^{(m)} / c_m^{(m)}$ для $j=0, 1, \dots, m-1$ – невідомі коефіцієнти, спосіб пошуку яких залежить від способу введення у просторі E поняття норми, а також від способу її мінімізації. Також варто сказати, що згідно умов теореми 1.2 послідовність мінімальних значень w_m функціонала $\| \Psi_m^{(m)} \|$ мажорується послідовністю $\{ \| Z_m \| / |c_m| \}$, яка є збіжною до нуля. Звідси маємо

$$\lim_{m \to \infty} w_m = 0.$$

3. Знаходимо корені $\mu_{\mathrm{n}}^{(\mathrm{m})}$, $n=1, 2, \dots, m$ полінома

$$\tilde{F}_m^{(m)}(\mu) = \sum_{j=0}^{m} \tilde{c}_j^{(m)} \mu^j \tag{1.14}$$

Методи пошуку коренів поліномів (1.14) можна знайти, наприклад, у [5]. У веб-додатку для пошуку коренів многочлена використовувалась бібліотека Numpy.

4. Шукаємо функції $u_n^{(m)}$ за формулою (1.12) ,

$$Z_{\mathrm{j}}^{(\mathrm{m})} = \sum_{i=0}^{j-1} \tilde{c}_i^{(m)j} \nu_{j-i} \ .$$

Розглянемо умову припинення обчислень для даного алгоритму.

Нехай характеристичні числа задачі (1.3) відсортовані та пронумеровані у порядку зростання їх модулів, тобто $|\mu_1| \leq |\mu_2| \leq \ldots \leq |\mu_m| \leq \ldots$.

Враховуючи, що $\nu_j = A\nu_{j-1} = A^{j-1}\nu_0$, та представлення ν_0 формулою (1.8) із теореми 1.1 і те, що $Au_n = \frac{1}{\mu_n} u_n,$ отримаємо

$$\nu_j = \frac{1}{\mu_1^j} \left(b_1 u_1 + \sum_{n=2}^{\infty} b_n u_n \left(\frac{\mu_1}{\mu_n} \right)^j \right) \qquad (1.15)$$

Оскільки характеристичні числа відсортовані за зростанням їх модулів матимемо $\left| \frac{\mu_1}{\mu_n} \right| \leq 1$ при $n \geq 2$. Тому при обчисленні кожного наступного ν_j (зростанні ітерації j) значення ряду у правій частині формули (1.15) буде зменшуватись. Цю властивість використовує звичайний степеневий метод обчислення власних значень. Передбачається виконання достатньої кількості ітерацій для знаходження $\nu_j = cu_1$. Після того як обчислювальний процес зупиниться, перше власне значення знаходимо як відношення двох останніх ітерацій.

ММПН використовує усі виконані ітерації. Залучення до обчислення всіх ν_m на кожній стадії обчислювального процесу дозволяє уточнювати відомі наближення характеристичних чисел та отримувати значення нових. Відомості про $\mu_2, \mu_3 \ldots \mu_m \ldots$ міститься у обчисленні ряду (другому доданку) формули (1.15). Отримувати цю інформацію можна лише до тих пір, поки при черговому кроці M

не виконується умова $\nu_{\text{M}} = cu_1$. При кожній наступній ітерації наші значення практично будуть пропорційні ν_{M} і почерпнути нову інформацію буде неможливо.

Із зростанням номера ітерації обумовленість матриці системи рівнянь, з якої визначаємо коефіцієнти полінома (1.14), погіршується, обчислення стають нестійкими, наступні ітерації не дають нові характеристичні числа, а зростання степеня полінома (1.14) дає нулі, які є сторонніми розв'язками. Ці сторонні розв'язки можна розпізнати, оскільки відповідні їм функції практично рівні нулю. У такому випадку ітераційний процес потрібно припинити. Також ітераційний процес припиняють, якщо $\nu_j, j = 0, 1, 2, \dots, m$, стають лінійно залежними.

1.4. Уточнення характеристичних чисел поліному методом Ньютона

При пошуку коренів рівняння (1.14) використовується бібліотека Numpy, яка повертає розв'язки із точністю, що може не відповідати вимогам ММПН. Дані значення можна брати для визначення околу $[a, b]$, на якому шукатимемо уточнені значення характеристичних чисел, і для їх пошуку використаємо метод Ньютона, також відомий як метод дотичних.

Розглянемо нелінійне рівняння вигляду $f(x) = 0$, де $f(x)$ – функція, неперервна на відрізку $[a, b]$, яка має на даному відрізку, відмінні від нуля, похідні першого і другого порядків. У нашому випадку функції $f(x)$ відповідає правій частині рівняння (1.14) і є поліномом степеня m, а отже вона є неперервною на нашому відрізку та має неперервні похідні першого та другого порядків. Тоді, ідея методу Ньютона полягає в тому, що на кожній ітерації графік функції $f(x)$ замінюється дотичною і точку перетину кожної з цих дотичних з віссю абсцис приймають за чергове наближення до шуканого кореня, а кожне наступне наближення обчислюватиметься за формулою:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

За початкове наближення у методі Ньютона слід брати таку точку x_0 з відрізку $[a, b]$, в якій $f''(x_0)f(x_0) > 0$. Процес відбувається допоки не виконається умова:

$$|x_k - \ x_{k+1}| \leq \sqrt{\frac{2m_1}{M_2}}\varepsilon,$$

але якщо виконується умова, що $M_2 < 2m_1$, де M_2 – максимальне значення похідної на проміжку, а m_1 – мінімальне, то ітераційний процес можна завершити при $|x_{k+1} - x_k| < \varepsilon$.

2. ПРАКТИЧНА РЕАЛІЗАЦІЯ ВЕБ-ДОДАТКУ

2.1. Обґрунтування вибору технологій і засобів вирішення поставленого завдання

2.1.1. Back-end

Для створення веб-додатків за допомогою платформи .NET можна використовувати різні технології. ASP.NET підтримує ряд моделей програмування для їх створення:

– ASP.NET Web Forms;
– ASP.NET MVC;
– ASP.NET Core MVC;
– ASP.NET Web API;
– ASP.NET Core Web API;
– Blazor тощо.

Для реалізації веб-додатку було обрано серверну технологію ASP.NET Core Web API. Вона забезпечує сумісність між платформами та пропонує такі функції, як висока продуктивність, масштабованість, інтегрована підтримка Dependency Injection, керування конфігурацією. Вибір ASP.NET Core Web API як серверної технології добре відповідає вимогам обраного застосунку, забезпечуючи надійну основу для обробки HTTP-запитів і керування загальною функціональністю на стороні сервера. При розробці додатку використано найновішу версію платформи .NET – 7.0. Дана версія містить останні оновлення безпеки та продуктивності, усунуто помилки та зауваження, які були виявлені за час використання попередньої версії 6.0. В основному ASP.NET Core Web API, як і вся платформа .NET, використовує C# – об'єктно-орієнтовану мову програмування з безпечною системою типізації для платформи .NET.

Python – інтерпретована об'єктно-орієнтована мова програмування високого рівня зі строгою динамічною типізацією. Структури даних високого рівня разом із

динамічною семантикою та динамічним зв'язуванням роблять її привабливою для швидкої розробки програм, а також як засіб поєднання наявних компонентів. Зазвичай використовується для розробки штучного інтелекту, машинного навчання, аналітики та візуалізації даних, програмування додатків, веб-розробки тощо. Нам вона пригодилася для написання коду при здійсненні обчислень завдяки багатому вибіру математичних фреймворків, бібліотек та патернів, а саме:

1) Numpy – для алгебраїчних перетворень, обчислення кореня числа, знаходження розв’язків системи рівнянь, пошук коренів полінома, пошук оберненої матриці, добуток матриць;

2) Scipy – для знаходження визначеного інтегралу;

3) Sympy – для знаходження невизначеного інтегралу та спрощення функцій;

4) re – для роботи із символьними рядками

Очевидною проблемою стало коректне пов’язання двох різних між собою мов програмування, в нашому випадку C# та Python. Дослідивши це питання з-поміж усіх варіантів виділили два основні:

а) IronPython – одна з основних реалізацій мови Python, призначена для платформи Microsoft .NET або Mono, повністю написана на C# і є транслятором компілюючого типу. На даний момент він є чи не найпопулярнішим рішенням для зв’язки C# та Python, проте, протестувавши його роботу в нашому додатку, ми прийшли до висновку, що водночас він є й переоціненим через поширеність версії на основі Python 2, що не збігається з кодом математичних обчислень зроблених на Python 3. Розробка версії IronPython на основі Python 3 йде повільно, також мало документації з нею, що суттєво ускладнює використання для нас. Звичайно, ми б могли переписати програмний код для використання попередньої версії продукту, однак це не вирішило проблеми через відсутність підтримки для фреймворків та бібліотек, перелічених у описі Python, тому надалі цей варіант не розглядався.

б) Python.NET – пакет, який дає програмістам Python майже безпроблемну інтеграцію з .NET Common Language Runtime (CLR), надає потужний інструмент сценаріїв додатків для розробників .NET. Це дозволяє коду Python взаємодіяти з

CLR, а також може використовуватися для вбудовування Python у програму .NET. Незважаючи на відсутність стабільного релізу цього пакету та невелику популярність, саме згадана останньою можливість вбудовування та інтеграції, порівняно з вищеперерахованими варіантами, повністю задовольнила нашу потребу Python-реалізації та дозволила успішно інтегрувати код математичних обчислень у веб-додаток, зробити його одним цілим.

Також виникла потреба зберігати результати обчислень, для цього було використано бібліотеку OpenXML SDK. Дана бібліотека є офіційною бібліотекою Microsoft для роботи з файлами OpenXML. Вона пропонує повний набір класів і методів для створення, читання та зміни документів OOXML. SDK надає строго типізований API, який дозволяє працювати з елементами, атрибутами та структурами, визначеними в специфікації OOXML. Вона також містить функції для створення звітів, маніпулювання стилями, форматуванням і виконання перевірки документів.

2.1.2. Front-end

React – це широко поширена бібліотека JavaScript для створення інтерфейсу користувача. Вона створена на основі компонентної архітектури та забезпечує декларативний підхід до розробки інтерфейсу. React дозволяє створювати повторно використовувані компоненти інтерфейсу та керувати станом програми, внаслідок чого, оновлюються та відображаються лише необхідні компоненти, що забезпечує покращену продуктивність та взаємодію з користувачем. Крім того React використовує декларативний синтаксис, що означає, що саме розробник описує вигляд інтерфейсу. React слідує односпрямованому потоку даних, де дані протікають в одному напрямку від батьківських до дочірніх компонентів. Це полегшує відстеження та налагодження змін даних, що забезпечує більш передбачувану поведінку програми.

Оскільки потік даних у React прямує від батьківських компонентів до дочірніх, було прийнято рішення дещо спростити процес керування станом

програми, одночасно використовуючи переваги компонентної архітектури React. Для вирішення цієї проблеми було використано бібліотеку React-sweet-state.

React-sweet-state – це бібліотека керування станом для програм React. Дана бібліотека є створеною на основі фреймворку Redux, популярною бібліотекою керування станом. Вона використовує подібну архітектуру, де стан програми зберігається в центральному сховищі, що в свою чергу спрощує взаємодію компонентів та стану додатку. Це дозволяє забезпечити передбачуваний і централізований підхід до управління станом програми. Крім того, React-sweet-state використовує методи запам'ятовування для оптимізації продуктивності. Це гарантує повторну візуалізацію лише тих компонентів, які залежать від конкретних змін стану, зменшуючи непотрібні рендери та покращуючи загальну продуктивність програми.

Далі постало питання розробки інтерфейсу користувача. Оскільки React використовує компонентну архітектуру, доцільним рішенням для пришвидшення процесу розробки було використання UI фреймворку, а саме Ant Design.

Ant Design – є популярною UI бібліотекою для React, яка пропонує набір попередньо розроблених, високо-функціональних компонентів, таких як таблиці, кнопки, поля вводу, меню, картки, сповіщення тощо. Ant Design надає багату колекцію макетів, тем, іконок які забезпечують легкий у розробці та візуально привабливий інтерфейс користувача. Особливо слід звернути увагу на документацію даної бібліотеки, яка включає докладні приклади, посилання на API та рекомендації щодо використання. Бібліотека має активну спільноту, яка сприяє її розвитку, надає підтримку та ділиться додатковими ресурсами, наприклад навчальними посібниками та плагінами сторонніх розробників.

Хоч дана бібліотека забезпечувала широкий спектр компонентів для створення інтерфейсу, проте не надавала можливості візуалізувати результати математичних обчислень графічно. Для реалізації цього було використано бібліотеку React-plotly.

React-plotly надає велику кількість інтерактивних типів діаграм, такі як: лінійні, стовпчасті, секторні, точкові діаграми тощо. Такі компоненти можна легко

налаштувати та покращити за допомогою масштабування, панорамування та інших інтерактивних функцій. Особливістю React-plotly є те, що вона легко інтегрується з станом React програми, дозволяючи динамічно оновлювати дані або конфігурацію компонентів. Це гарантує, що візуалізація залишається чутливою до змін даних або взаємодії користувача.

2.2. Прикладна реалізація алгоритму ММПН

Як вже було сказано, всі обчислення відбувалися з використанням мови програмування Python 3. Після того, як користувач введе початкову функцію v_0, межі інтегрування та оператор $G(x,t)$ першочергово відбувається обчислення послідовних ітерацій функцій v_m, для нашої реалізації ми обмежили $m = 4$.

Для зручності майбутніх алгебраїчних обчислень перш за все видалимо усі порожні рядки із початкової функції n_0, далі замінимо у вхідній функції усі невідомі параметри x на t використовуючи метод sub бібліотеки re, де першим вхідним параметром є символ, що потрібно змінити, другим – символ на який ми змінюємо, третім — стрічка, над якою виконується операція. Крім того, використовуючи бібліотеку Sympy, а саме функцію symbols, оголосимо символи x та t змінними. Дію оператора на функцію в залежності від значення t розглядатимемо на двох проміжках – $[\text{leftSide}, x]$ та $[x, \text{rightSide}]$, після чого інтегруємо утворені функції. Для цього використаємо функцію sympy.integrate, у яку передаємо в якості параметрів підінтегральну функцію, змінну, за якою відбудеться інтегрування, параметр, за яким інтегрується, а також проміжок, на якому відбувається інтегрування. Результати інтегрування на кожному з проміжків додаємо та за допомогою функції sympy.expend спрощуємо утворений вираз. Дана частина програмного коду відображена на рисунку 2.1.

```
v0_x = ''
for i in range(len(funV0)):
    if funV0[i] == '^':
        v0_x += '**'
    else:
        v0_x += funV0[i]

v0_x = v0_x.replace(' ','')

v0_t = re.sub(operatorValues[0], operatorValues[1], v0_x)

x, t = sympy.symbols(f'{operatorValues[0]} {operatorValues[1]}')

if('≥' in scopes[0] or '>' in scopes[0]):
    operators.reverse()

counter = 0
vx_x = [v0_x]
vx_t = [v0_t]
while counter ≠ 4:
    vx_part2 = sympy.integrate('((' + operators[0] + ')*(' + vx_t[counter] + '))', (t, x, rightSide))
    vx_part1 = sympy.integrate('((' + operators[1] + ')*(' + vx_t[counter] + '))', (t, leftSide, x))


    full_vx = str(sympy.expand(str(vx_part1) + '+' + str(vx_part2)))
    vx_x.append(full_vx)
    vx_t.append(re.sub(operatorValues[0], operatorValues[1], full_vx))

    counter = counter + 1
```

Рисунок 2.1. Реалізація дії оператора на функцію v_j.

Наступним кроком переходимо до пошуку коефіцієнтів $\tilde{c}_0, \ldots, \tilde{c}_{m-1}$, які буде використано для пошуку характеристичних чисел. За допомогою функції numpy.zeros створюємо матрицю розміру $m \times m$, заповнену нулями. Елементи матриці обчислюємо як скалярний добуток функцій (n_j, n_k) відповідно до формули (1.18). Таким же чином заповнюємо елементи (n_0, n_k) правої частини формули (1.18). Для обчислення скалярних добутків як значення визначеного інтегралу добутку функцій на [leftSide; rightSide] використаємо функцію scipy.integrate.quad. В результаті обчислення зводяться до розв'язання системи лінійних алгебраїчних рівнянь порядку m. Для пошуку розв'язку системи лінійних неоднорідних рівнянь бібліотека Numpy передбачає функцію numpy.linalg.inv, яка повертає обернену матрицю, та функцію numpy.dot, яка повертає скалярний добуток матриць. Застосування вказаних функцій дозволяє знайти коефіцієнти $\tilde{c}_0, \ldots, \tilde{c}_{m-1}$. Програмний код, що відображає обчислення над матрицями та розв'язання системи рівнянь, зображено на рисунку 2.2.

```
current_function = ''

def f(x):
    return float(eval(current_function,
        {'cos': math.cos, 'sin':math.sin, 'tan': math.tan, 'atan':math.atan, 'x': x, 'e':math.exp}))

x = sympy.Symbol('x')
scalar_matrix = numpy.zeros((len(v_m) - 1, len(v_m) - 1))
result_matrix = numpy.zeros(len(v_m) - 1)
for i1 in range(1, len(v_m)):
    for i2 in range(1, len(v_m)):
        current_function = '((' + v_m[i2] + ')*(' + v_m[i1] + '))'

        result, err = scipy.integrate.quad(f, float(leftSide), float(rightSide))
        scalar_matrix[len(v_m) - 1 - i1][len(v_m) - 1 - i2] = result

    current_function = '((' + v_m[0] + ')*(' + v_m[i1] + '))'

    result, err = scipy.integrate.quad(f, float(leftSide), float(rightSide))
    result_matrix[len(v_m) - 1 - i1] = -result

c_m = []
for i in range(0, len(scalar_matrix)):
    if i == len(scalar_matrix) - 1:
        c_m.append([result_matrix[i] / scalar_matrix[i][i]])
        break
    elif i == 0:
        c_m.append(numpy.linalg.inv(scalar_matrix).dot(result_matrix))
    else:
        new_scalar_matrix = []
        new_result_matrix = []
        for i1 in range(i, len(scalar_matrix)):
            add_scalar = []
            for i2 in range(i, len(scalar_matrix)):
                add_scalar.append(scalar_matrix[i1][i2])
            new_result_matrix.append(result_matrix[i1])
            new_scalar_matrix.append(add_scalar)

        c_m.append(numpy.linalg.inv(new_scalar_matrix).dot(new_result_matrix))
```

Рисунок 2.2. Реалізація формування системи рівнянь та її розв’язання.

Маючи коефіцієнти $\tilde{c}_0^{(m)}, \ldots, \tilde{c}_{m-1}^{(m)}$ знайдемо корені поліному заданого у вигляді (1.14). Для цього функції polynomial.polynomial. Polynomial передаємо коефіцієнти та сформуємо поліном, а за допомогою функції roots знаходимо його корені $\mu_n^{(m)}$. Крім того за допомогою функції isComplex відкидаються результати де корені є комплексними числами. (Рисунок 2.3.)

```
        newCm = []
        result = []
        for i in range(len(cm)):
            allCoef = [coef for coef in cm[i]]
            allCoef.append(1)

            roots = numpy.polynomial.polynomial.Polynomial(allCoef).roots()
            if True in numpy.iscomplex(roots):
                continue
            else:
                result.append(roots)
                newCm.append(cm[i])

        resultForTable = [[float(j) for j in i] for i in result]
        resultForTable = System.Array[System.Array[System.Double]](resultForTable)
```

Рисунок 2.3. Пошук коренів поліномів.

На наступному кроці уточнюємо знайдені корені реалізувавши метод Ньютона. (Рисунки 2.4 та 2.5)

```
def doNewton(function, a, b, e, check):
    add_func = ''
    for i in range(len(function)):
        if function[i] == '^': add_func += '**'
        else: add_func += function[i]
    function = add_func

    def derivative(func): # похідна функції
        x = sympy.Symbol('x')
        y = eval(func,
            {'cos': math.cos, 'sin':math.sin, 'tan': math.tan, 'atan':math.atan, 'x': x})
        res = str(y.diff(x))
        return res

    def f(x): # значення функції в точці
        return float(eval(function))

    def fn(x): # перша похідна
        return float(eval(derivative(function)))

    def fnn(x): # друга похідна
        return float(eval(derivative(derivative(function))))

    def direct(n):  # напрямок з якого відбуватиметься пошук v0
        try:
            if f(n) > 0 and fnn(n) > 0: return True
            elif f(n) < 0 and fnn(n) < 0: return True
            return False
        except:
            return False
```

```
if direct(a): # визначення напрямку
    xs = a
    silence = 1
    step = 0.01
    border = b
else:
    xs = b
    silence = -1
    step = -0.01
    border = a

# пошук мінімального та максимального значення похідної на проміжку [a, b]
min_val = math.fabs(fn(xs))
max_val = math.fabs(fn(xs))
i = xs
while i*silence ≤ border*silence:
    if math.fabs(fn(i)) < min_val: min_val = math.fabs(fn(i))
    elif math.fabs(fn(i)) > max_val: max_val = math.fabs(fn(i))
    i += step

xs = (a+b)/2
temp = b

if(max_val == 0): return xs

while math.fabs(xs - temp) ≥ math.sqrt((2*min_val*e)/max_val):
    temp = xs
    xs = xs - (f(xs)/fn(xs)) # пошук Xi+1

return xs
```

Рисунок 2.4. Метод Ньютона.

```
result = []
for i in range(0, len(cm)):
    if(numpy.iscomplex(mu[i]).any()):
        continue

    polinom = ''
    for j in range(0, len(cm[i])):
        if(j ≠ len(cm[i]) - 1): polinom = polinom + str(cm[i][j])+ '*x**'+ str(j)+'+'
        else: polinom = polinom + str(cm[i][j])+ '*x**'+ str(j)

    concreteRoots = []
    for item in mu[i]:
        concreteRoots.append(doNewton(polinom, item-0.001, item+0.001, 0.0001, False))

    result.append(concreteRoots)
result.append([])

result = System.Array[System.Array[System.Double]](result)
```

Рисунок 2.5. Уточнення коренів використовуючи метод Ньютона.

Крім обчислення власних значень, було побудовано функції $u_n^{(m)}$. Перш за все було знайдено функції $Z_n^{(m)}$, які ми зберегли у вигляді лінійних комбінацій послідовних ітерацій функцій v_j, $(j = 0, \dots, m)$.

В кінцевому результаті, маючи характеристичні числа $\mu_n^{(m)}$ та функції $Z_n^{(m)}$, знаходимо власні функцій $u_n^{(m)}$ за формулою (1.12). Графік власних функції $u_n^{(m)}$ будемо відображати.

```
def getZn(zm, zj):
    zjm = ''
    for i in range(zj):
        zjm += '((' + str(cm[zm][i]) + str(')*(') + vm[zj-i] + '))'
        if(i != zj-1): zjm += '+'
    return zjm

unh = []
for um in range(len(mu)):
    unm = []
    for un in range(len(mu[um])):
        uij = ''
        for j in range(len(mu[um])):
            uij += '(('+getZn(um, j+1)+')*('+str(mu[um][un])+'**'+str(j+1)+'))'
            if(j != len(mu[um])-1): uij += '+'
        unm.append(uij)
    unh.append(unm)
unh.reverse()
```

Рисунок 2.6. Програмний код пошуку наближень власних функцій.

Як кінцевий результат є пошук власних функцій $u_n^{(m)}$ на кожному кроці ММПН. Далі знайдені власні функції $u_n^{(m)}$ передаються в метод GetPlot, де проміжок [leftSide; rightSide] розбивається на 100 рівновіддалених точок, після чого, знаходимо значення $u_n^{(m)}$ у цих точка (рисунок 2.7). Отримані результати передаємо на Front-end частину та відображаємо їх у вигляді графів використовуючи бібліотеку React-plotly.

```
end = len(un)
x = sympy.Symbol('x')
xi = numpy.linspace(float(leftSide), float(rightSide), 100)
allGraphs = []
for uni in range(0, end):
    uniGraphs = []
    for ui in range(0, len(un[uni])):
        def norma_f(x):
            return float(eval('('+str(un[uni][ui])+')**2',
                {'cos': math.cos, 'sin':math.sin, 'tan': math.tan, 'atan':math.atan, 'x': x, 'e':math.exp}))

        def f(x):
            return float(eval(un[uni][ui],
                {'cos': math.cos, 'sin':math.sin, 'tan': math.tan, 'atan':math.atan, 'x': x, 'e':math.exp}))

        norma, err = scipy.integrate.quad(norma_f, float(leftSide), float(rightSide))

        f_ui = [f(i)/math.sqrt(norma) for i in xi]
        uniGraphs.append(f_ui)
    allGraphs.append(uniGraphs)

parsedXi = System.Array[System.Double](xi)
parsedAllGraphs = System.Array[System.Array[System.Array[System.Double]]](allGraphs)
```

Рисунок 2.7. Обчислення значень власних функцій.

2.3. Структура та реалізація Back-end

У Visual Studio Community 2022 версії 17.2.1 створено проект з шаблону ASP.NET Core Web API з використанням .NET 7.0.

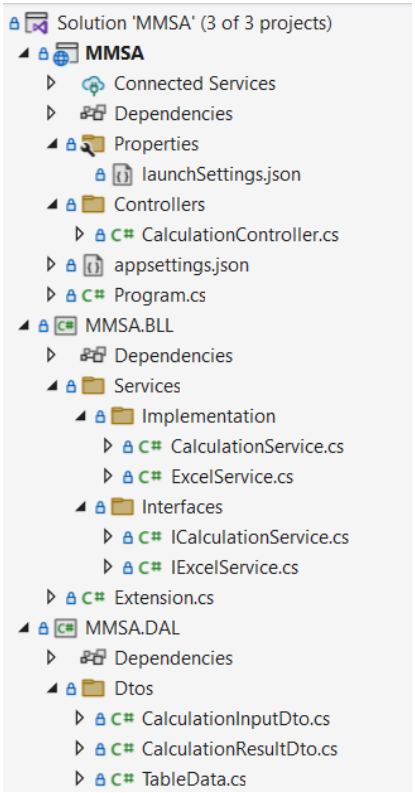


Рисунок 2.8. Структура Back-end частини

Архітектура проекту поділяється на три частини:

1) MMSA (Web application layer);
2) BLL (Business layer);
3) DAL (Data access layer).

Розглянемо розміщення папок і файлів нашого проекту, їх призначення:

launchSettings.json – використовується для налаштування різних аспектів параметрів запуску програми. Він забезпечує зручний спосіб налаштування поведінки програми під час розробки.

4)


```
{
  "$schema": "https://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:20269",
      "sslPort": 44300
    }
  },
  "profiles": {
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "launchUrl": "swagger",
      "applicationUrl": "http://localhost:5228",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "launchUrl": "swagger",
      "applicationUrl": "https://localhost:7131;http://localhost:5228",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "swagger",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```


Рисунок 2.9. launchSettings.json

appsettings.json – файл, який використовується для зберігання конфігураційних рядків, поділений на групи за типами налаштувань. Представлений у двох варіантах: звичайному та розробника.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Рисунок 2.10. Розробницька версія appsettings.json

Controllers – папка для зберігання класів контролерів. Контролери відіграють вирішальну роль у обробці вхідних запитів HTTP та створенні відповідних відповідей HTTP. У нашому вона містить один клас контролер CalculationController.cs.

```
using Microsoft.AspNetCore.Mvc;
using MMSA.BLL.Services.Interfaces;
using MMSA.DAL.Dtos;

namespace MMSA.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    1 reference
    public class CalculationController : ControllerBase
    {
        private readonly ICalculationService _calculationService;
        private readonly IExcelService _excelService;

        0 references
        public CalculationController(ICalculationService calculationService, IExcelService excelService)
        {
            _calculationService = calculationService;
            _excelService = excelService;
        }

        [HttpGet("MakeCalculation")]
        0 references
        public IActionResult MakeCalculation([FromQuery] CalculationInputDto calculationInput)
        {
            var product = _calculationService.MakeCalculation(calculationInput);
            if (product == null)
            {
                return NotFound();
            }
            return Ok(product);
        }

        [HttpGet("GetFile")]
        0 references
        public IActionResult GenerateExcelFile([FromQuery] TableData tableResults)
        {
            var parsedTableResults = _excelService.ParseTableResults(tableResults.CalculationResults);

            return Ok(_excelService.GetFile(parsedTableResults));
        }
    }
}
```

Рисунок 2.11. CalculationController.cs

Program.cs – файл-точка входу в програму. Тут відбувається налаштування ASP.NET Core Web API, зокрема налаштовування Dependency Injection, middleware компонентів для маршрутизації, CORS, Swagger, автентифікації, авторизації та запуск програми для обробки вхідних запитів.

```
using MMSA.BLL.Services.Implementation;
using MMSA.BLL.Services.Interfaces;


var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
builder.Services.AddScoped<ICalculationService, CalculationService>();
builder.Services.AddScoped<IExcelService, ExcelService>();

var app = builder.Build();

app.UseRouting();
app.UseCors(x => x.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader());

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
app.UseHttpsRedirection();
app.UseAuthentication();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

Рисунок 2.12. Program.cs

Services – папка, що містить папки Implementation та Interfaces, класи якої реалізують підхід Dependency Injection, а також здійснюють логіку обчислень та збереження результатів.

```
public interface ICalculationService
{
    public CalculationResultDto MakeCalculation(CalculationInputDto calculationInput);
    2 references
    public object GetVm(object funV0, object operatorValues, object operators, object scopes, object leftSide, object rightSide);
    2 references
    public object GetCm(object vm, object leftSide, object rightSide);
    2 references
    public object[] GetMu(object cm);
    2 references
    public object GetUn(object vm, object cm, object mu);
    2 references
    public object[] GetPlot(object un, object leftSide, object rightSide);
    2 references
    public object GetMainResult(object cm, object mu);
}
```

Рисунок 2.14. ICalculationService.cs

```
namespace MMSA.BLL.Services.Interfaces
{
    4 references
    public interface IExcelService
    {
        2 references
        public List<List<double>> ParseTableResults(List<string> stringTableResults);
        2 references
        public string GetFile(List<List<double>> tableResults);
    }
}
```

Рисунок 2.15. IExcelInterface.cs

```
public class CalculationService : ICalculationService
{
    private readonly ILogger _logger;
    private IntPtr _threadState;


    0 references
    public CalculationService(ILogger<CalculationService> logger)
    {
        Initialize();
        _logger = logger;
    }

    1 reference
    private static void Initialize()
    {
        string pythonDLL = @"C:\Users\Dmytro\AppData\Local\Programs\Python\Python311\python311.dll";
        Environment.SetEnvironmentVariable("PYTHONNET_PYDLL", pythonDLL);
        PythonEngine.Initialize();
    }


    2 references
    public CalculationResultDto MakeCalculation(CalculationInputDto calculationInput)
    {
        try
        {
            var vm = GetVm(calculationInput.InputFunction, calculationInput.OperatorValues,
                calculationInput.Operators, calculationInput.Scopes,
                calculationInput.LeftSide, calculationInput.RightSide);

            var cm = GetCm(vm, calculationInput.LeftSide, calculationInput.RightSide);

            var muNewCm = GetMu(cm);

            var concreteRoots = GetMainResult(muNewCm[2], muNewCm[0]);

            var un = GetUn(vm, cm, concreteRoots);

            var plot = GetPlot(un, calculationInput.LeftSide, calculationInput.RightSide);

            return new CalculationResultDto { MU = (double[][])concreteRoots, PlotXi = (double[])plot[0], PlotFXi = (double[][][])plot[1] };
        }
        catch (Exception exception)
        {
            _logger.LogInformation($"Something gone wrong ... : {exception.Message}");
            return null;
        }
    }
```

Рисунок 2.16. CalculationService.cs

Клас CalculationService реалізує інтерфейс ICalculationService. Конструктор CalculationService ініціалізує середовище виконання Python, викликаючи метод Initialize, який встановлює шлях до DLL Python та ініціалізує Python Engine. Метод MakeCalculation є основною точкою входу для процесу обчислення. Він приймає об’єкт CalculationInputDto як вхідні дані та повертає об’єкт CalculationResultDto.

Методи GetVm, GetCm, GetMu, GetUn і GetPlot є допоміжними методами, описаними вище.

```
public class ExcelService: IExcelService
{
    private readonly ILogger _logger;
    0 references
    public ExcelService(ILogger<CalculationService> logger)
    {
        _logger = logger;
    }

    2 references
    public List<List<double>> ParseTableResults(List<string> stringTableResults)
    {
        _logger.LogInformation("Start parcing ... ");

        var data = new List<List<double>>();
        foreach (var row in stringTableResults)
        {
            var stringNumbers = row.Split(",");
            var rowList = new List<double>();

            foreach (var number in stringNumbers)
                rowList.Add(Double.Parse(number, NumberStyles.Float, CultureInfo.InvariantCulture));

            data.Add(rowList);
        }

        _logger.LogInformation("return results ... ");
        return data;
    }

    2 references
    public string GetFile(List<List<double>> tableResults)
    {
        byte[] excelBytes;
        using (var memoryStream = new MemoryStream())
        {
            using (var spreadsheetDocument = SpreadsheetDocument.Create(memoryStream, SpreadsheetDocumentType.Workbook))
            {
                var workbookPart = spreadsheetDocument.AddWorkbookPart();
                workbookPart.Workbook = new Workbook();

                var worksheetPart = workbookPart.AddNewPart<WorksheetPart>();
                worksheetPart.Worksheet = new Worksheet(new SheetData());
                var sheets = spreadsheetDocument.WorkbookPart.Workbook.AppendChild(new Sheets());

                var sheet = new Sheet
                {
                    Id = spreadsheetDocument.WorkbookPart.GetIdOfPart(worksheetPart),
                    SheetId = 1,
                    Name = "Sheet1"
                };
                sheets.Append(sheet);

                var sheetData = worksheetPart.Worksheet.GetFirstChild<SheetData>();

                for (var row = 0; row < tableResults.Count; row++)
                {
                    var rowData = tableResults[row];
                    var rowElement = new Row();

                    for (var col = 0; col < rowData.Count; col++)
                    {
                        var cell = new Cell
                        {
                            DataType = CellValues.Number,
                            CellValue = new CellValue(rowData[col])
                        };
                        rowElement.Append(cell);
                    }
                    sheetData.Append(rowElement);
                }
            }
            excelBytes = memoryStream.ToArray();
        }
        return (Convert.ToBase64String(excelBytes));
    }
}
```

Рисунок 2.17. ExcelService.cs

Клас ExcelService реалізує інтерфейс IExcelService і забезпечує функціональні можливості для аналізу результатів таблиці, представлених у вигляді рядків та створення файлу Excel на основі списку результатів таблиці. Метод ExcelService приймає параметр типу List<string>, що представляє результати таблиці як рядки. Він повторює кожен рядок у списку, розділяє його комами (','), щоб отримати окремі числа, і перетворює кожне число у тип Double за допомогою методу Double.Parse. Проаналізовані числа зберігаються в об’єкті типу List<List<double>>, що представляє двовимірний список. Нарешті, метод повертає список даних.

Метод GetFile приймає параметр List<List<double>> tableResults, який представляє дані таблиці. Він створює файл Excel за допомогою бібліотеки DocumentFormat.OpenXml. Метод створює об’єкт типу SpreadsheetDocument у пам’яті за допомогою MemoryStream і встановлює його тип на SpreadsheetDocumentType.Workbook. Тобто створює аркуш із назвою «Sheet1» у Excel. Потім він повторює кожен елемент рядку і стовпця списку tableResults, створює клітинку для кожного числа, встановлює тип даних CellValues.Number і додає клітинку до рядка, що додається до даних аркуша. Нарешті, метод перетворює потік пам’яті в масив байтів і повертає файл Excel як рядок у кодуванні base64.

Dto – папка, де знаходять Dto-моделі, що створені для зручності передавання даних між Back-end та Front-end частинами.

2.4. Структура Front-end

Далі розглянемо структуру Front-end частини веб-додатку. Front-end частину було реалізовано за допомого TypeScript React. Для роботи із фреймворками та бібліотеками використовувався npm (Node Package Manager).

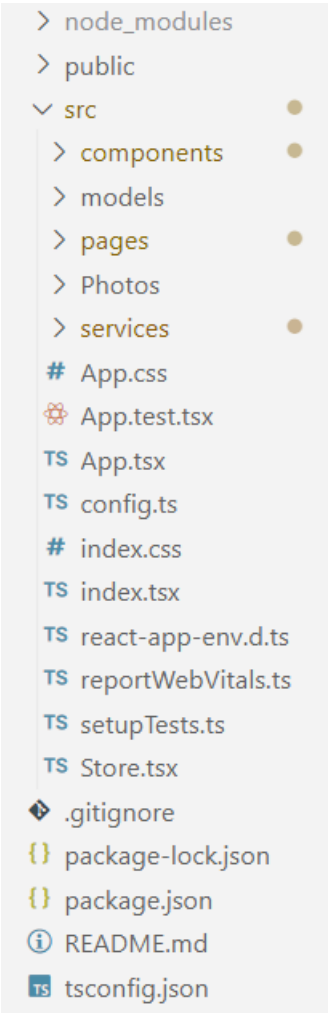


Рисунок 2.18. Структура Front-end частини

1) node_modules містить усі залежності проекту, встановлені за допомогою менеджера пакетів, у даному випадку npm.

2) public папка містить статичні ресурси, як-от шаблони HTML або файли зображень, які обслуговуються безпосередньо веб-сервером.

3) .gitignore визначає файли та папки, які слід ігнорувати контролем версій.

4) package.json містить метадані проекту та перераховує залежності, необхідні для програми.

5) tsconfig.json містить параметри конфігурації компілятора TypeScript.

Папка src – це місце, де знаходиться фактичний вихідний код програми. У папці components відповідно складові, або компоненти інтерфейсу користувача, у папці models містяться моделі даних для звязку Front-end та Back-end частин.

Photos - папка де зберігаються зображення, потрібні для веб-додатку. Далі йде папка services, що зберігає в собі файли, які діють як рівень абстракції для виконання обчислень і завантаження даних за допомогою служби API.

Інші файли, такі як App.css, App.test.tsx, App.tsx, config.ts, index.css, react-app-env.d.ts, reportWebVitals.ts і setupTests.ts, пов'язані з конфігурацією, тестуванням та налаштуванням запуску програми.

2.4.1. Axios та API-services

Перш за все було реалізовано взаємодію Front-end та Back-end частин з використанням бібліотеки Axios, яка надає функції для створення HTTP-запитів.

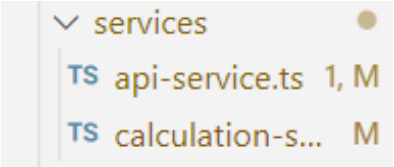


Рисунок 2.19. Axios та сервіс для зв'язку з Back-end частиною.

```
import axios from "axios";
import BASE_URL from "../config";

interface HttpResponse {
  status: number;
  headers: any;
  data: any;
}

const get = async (url: string, data?: any, paramsSerializer?: any): Promise<HttpResponse> => {
  const response = await axios.get(BASE_URL + url, {
    params: data,
    paramsSerializer: paramsSerializer
  });
  return response;
};

const post = async (url: string, data?: any) => {
  const response = await axios.post(BASE_URL + url, data, {
    headers: {
      "Accept": "application/json",
      "Content-Type": 'application/json',
    },
  });
  return response;
};
```

```
const put = async (url: string, data?: any): Promise<HttpResponse> => {
  const response = await axios.put(BASE_URL + url, data, {
    headers: {
      "Accept": "application/json",
      "Content-Type": "application/json",
    },
  });
  return response;
};

const remove = async (url: string, data?: any, options: any = { }): Promise<HttpResponse> => {
  const response = await axios.delete(BASE_URL + url, {
    ...options,
    params: data,
  });
  return response;

};

export default { get, post, put, remove };
```

Рисунок 2.20. api-services.ts (Axios).

Код починається з імпорту бібліотеки axios, яка використовується для створення запитів HTTP, і константи BASE_URL з файлу "../config".

```
1   const BASE_URL = 'https://localhost:7131/api/';
2   export default  BASE_URL;
```

Рисунок 2.21. config.ts

Інтерфейс HttpResponse визначено для опису структури об’єкта відповіді, що повертається запитами HTTP.

Код визначає чотири асинхронні функції: get, post, put і remove, які відповідають різним методам HTTP.

get – асинхронна функція, яка виконує HTTP-запит GET.

post – асинхронна функція, яка виконує запит HTTP POST.

put – асинхронна функція, яка виконує запит HTTP PUT.

remove – асинхронна функція, яка виконує запит HTTP DELETE.

Об’єкт, що експортується, містить вище описані чотири функції. Це дозволяє іншим модулям імпортувати та використовувати ці функції для виконання HTTP-

запитів. Підсумовуючи, цей код надає набір функцій, які обертаються навколо методів Axios (get, post, put, delete), щоб спростити створення HTTP-запитів із базовою URL-адресою. Він обробляє різні методи HTTP та забезпечує послідовний спосіб виконання запитів і обробки відповідей. Даний модуль використовує клас CalculationService, який надає методи для виконання обчислень і завантаження отриманих результатів.

```
import { CalculationSettings } from "../models/CalculationSettings";
import { TableData } from "../models/TableData";
import Api from "./api-service";

export default class CalculationService {
    makeClaculation = async (NewTableSettings: CalculationSettings) => {
        return (
          await Api
            .get(
              `Calculation/MakeCalculation`,
              NewTableSettings,
              (params: any) => {
                return Object.entries(params)
                  .map(([key, value]) => {
                    if (Array.isArray(value) && value) {
                      return value.map((it) => `${key}=${it}`).join("&");
                    }
                    return `${key}=${value}`;
                  })
                  .join("&");
              }
            )
            .catch((error) => {
              throw new Error(error);
            })
        ).data;
      };
```

Рисунок 2.22. Метод makeCalculation.cs

```
    downloadTable = async (tableData: TableData) => {
      return (
        await Api
          .get(
            `Calculation/GetFile`,
            tableData,
            (params: any) => {
              return Object.entries(params)
                .map(([key, value]) => {
                  if (Array.isArray(value) && value) {
                    return value.map((it) => `${key}=${it}`).join("&");
                  }
                  return `${key}=${value}`;
                })
                .join("&");
            }
          )
          .catch((error) => {
            throw new Error(error);
          })
      ).data;
    };
}
```

Рисунок 2.23. Метод downloadTable.cs

Першочергово імпортується Api з "./api-service". Передбачається, що модуль API надає методи для створення HTTP-запитів.

Методи makeCalculation та downloadTable — це асинхронні функції, які приймають відповідні їм параметри. Вони виконують запит HTTP GET до кінцевої точки ‘Calculation/MakeCalculation’ та ‘Calculation/GetFile’ відповідно за допомогою методу Api.get. Обидва методи обробляють будь-які помилки, які виникають під час виклику API, перехоплюючи їх і створюючи новий об’єкт Error із повідомленням про помилку. Властивість .data відповіді повертається за умови, що вона містить потрібні дані з відповіді API.

2.4.2. Сховище стану програми (Store)

Наступним кроком було створення єдиного сховища станів програми, для уникнення деревовидних зв'язків компонентів. Для цього було використано бібліотеку «react-sweet-state» для визначення Store (сховища) та Actions для керування станом програми.

```
import { Action, createHook, createStore } from "react-sweet-state";

type State = {
    tableValues: number[][],
    graphXis: any,
    functionValues: number[][][],
    dataTable: any[],
    columns: any[]
    calculationError: boolean
};

const initialState: State = {
    tableValues: [[]],
    graphXis: [],
    functionValues: [[[]]],
    dataTable: [],
    columns: [],
    calculationError: false
};
```

Рисунок 2.24. State сховища.

У сховищі визначає тип State, який представляє форму стану програми. Він містить такі властивості, як tableValues, graphXis, functionValues, showMenu, dataTable та columns. Ініціалізація початкового стану програми відбувається за допомогою константи initialState, яка призначає значення за замовчуванням.

Далі визначається набір дій за допомогою об'єкта Actions. Кожна дія — це функція, яка приймає корисне навантаження та повертає функцію, яка змінює стан (рисунок 2.25). Ці дії відповідають за оновлення конкретних властивостей стану.

```
const actions = {
  setTableValues:
    (tableValue: [[]]): Action<State> =>
    ({ setState }) => {
      setState({
        tableValues: tableValue
      });
    },

  setTableColumns:
    (newColumns: any[]): Action<State> =>
    ({ setState }) => {
      setState({
        columns: newColumns
      });
    },

  setGraphXis:
    (graphXi: []): Action<State> =>
    ({ setState }) => {
      setState({
        graphXis: graphXi
      });
    },


  setFunctionValues:
    (functionValue: [[[]]]): Action<State> =>
    ({ setState }) => {
      setState({
        functionValues: functionValue
      });
    },

  setDataTable:
    (newData: any[]): Action<State> =>
    ({ setState }) => {
      setState({
        dataTable: newData
      });
    },

  setCalculationError:
    (isError: boolean): Action<State> =>
    ({ setState }) => {
      setState({
        calculationError: isError
      });
    }
};
```

Рисунок 2.25. Actions сховища.

Методи:

1) setTableValues оновлює стан tableValues.
2) setTableColumns оновлює стан columns.

3) setGraphXis оновлює стан graphXis.
4) setFunctionValues оновлює стан functionValues.
5) setDataTable оновлює стан dataTable.

```
const Store = createStore({ initialState, actions });
export const useTable = createHook(Store);
```

Рисунок 2.26. Створення сховища стану програми.

На рисунку 2.26 створюється сховище за допомогою функції createStore з "react-sweet-state". Вона приймає об’єкт, який містить початковий стан (State) і дії (Actions) як параметри. Далі таке сховище можна експортувати до будь якої компоненти програми.

2.4.3. Головна сторінка

Головну сторінку інтерфейсу користувача реалізовує файл MainPage.tsx. Код починається з імпорту необхідних залежностей і компонентів із різних бібліотек.

```
export const MainPage = () => {
    let api = new CalculationService();
    const [state, actions] = useTable();
    const [form] = Form.useForm();
```

Рисунок 2.27. Створення об’єкту api для роботи з API запитами та об’єкт сховища взаємодії з станом програми.

Компонент MainPage визначається як функціональний компонент, що виконує роль головної сторінки програми. Усередині компонента MainPage екземпляр класу CalculationService ініціалізується шляхом створення нового об’єкта за допомогою let api = new CalculationService();. Цей екземпляр використовуватиметься для викликів API для обчислень. Хук useTable використовується для доступу до стану програми та дій над ним. Він повертає

масив із двома елементами: стан (State) і дії (Actions). State містить поточний стан програми, а Actions містить функції для оновлення стану програми.

```
const makeCalculations = async (values: any) => {
    let operatorValues = values.g.split(',')

    const newCalculationSettings = {
      InputFunction: encodeURIComponent(values.inputFunction),
      OperatorValues: [encodeURIComponent(operatorValues[0]), encodeURIComponent(operatorValues[1])],
      Operators: [encodeURIComponent(values.Operator1), encodeURIComponent(values.Operator2)],
      Scopes: [encodeURIComponent(values.Scope1), encodeURIComponent(values.Scope2)],
      LeftSide: encodeURIComponent(values.leftSide),
      RightSide: encodeURIComponent(values.rigthSide)
      };

      await Api.makeClaculation(newCalculationSettings)
      .then((res) => {
        if(res.error === true){
          actions.setCalculationError(res.error)
          notificationLogic("error", "Помилка. Перевірте введені дані!!!!")
        }
        else{
          notificationLogic("success", "Успіх!")
          actions.setCalculationError(res.error)
          let mu = res.mu.reverse()
          mu.shift()
          const filteredArray = res.plotFXi.filter((item: any) => Array.isArray(item) && item.length > 0);
          MakeColumns(mu);
          actions.setTableValues(mu)
          actions.setFunctionValues(filteredArray)
          actions.setGraphXis(res.plotXi)
        }
      });
  };
```

Рисунок 2.28. Метод makeCalculations.

makeCalculations — це асинхронна функція, яка обробляє надсилання форми для виконання обчислень. Вона приймає значення форми як вхідні дані. Спочатку вона витягує значення змінних оператора G типу (1.27) і розбиває його на масив за допомогою метода Split. Потім він створює новий об’єкт CalculationSettings, із введених користувачем значень, та робить асинхронний виклик API для виконання обчислення. (Рисунок 2.28)

Функція використовує await для очікування завершення виклику API. Якщо виклик API успішний, вона оновлює значення станів сховища.

```
const MakeColumns = async (tableValues: any[][]) => {
  const newColumns: any = [
    {
      title: `Номер ітерації n`,
      dataIndex: `iteration`,
      key: `iteration`,
      align: 'center'
    }
  ];

  (tableValues).map((value: any[], index: number) => {
    newColumns.push(
      {
        title: `U${index+1}`,
        dataIndex: `U${index+1}`,
        key: `U${index+1}`,
        align: 'center'
      })
  })

  const dataSource = tableValues.map((row, i) => {
    let rowData: any = { key: i };
    rowData[`iteration`] = i+1
    row.forEach((cell, j) => {
      rowData[`U${j + 1}`] = cell;
    });
    return rowData;
  });

  actions.setDataTable(dataSource);
  actions.setTableColumns(newColumns);
}
```

Рисунок 2.29. Метод makeColumns.

Метод MakeColumns – створює масив стовпців таблиці, перебираючи значення та додаючи об’єкти стовпців до масиву newColumns. Він також створює масив dataSource шляхом зіставлення значень tableValues та створення об’єктів даних рядка. Отриманий масив dataSource і масив newColumns потім використовуються для оновлення стану таблиці за допомогою дій хука useTable. (Рисунок 2.29)

```
const scrollToTop = () => window.scrollTo({top: 0, left: 0, behavior: 'smooth'});
```

Рисунок 2.30. Метод ScrollToTop.

ScrollToTop – це допоміжна функція, яка прокручує вікно вгору після натискання кнопки.

```
<Form
  name="basic"
  onFinish={makeCalculations}
  form={form}
  id="area"
  style={{display:"flex", justifyContent:"center"}}
>
  <div style={{display: "flex", alignItems:"center", flexDirection:"column"}}>
    <Row justify={'center'} style={{height: "75px", fontWeight: '600', fontSize: '30px'  }}>
      Вигляд оператора
    </Row>
    <Row justify={'center'}>
      <img src={operator} alt="" className="curly-brace-left"/>
    </Row>
    <Row>
      <Form.Item
        label="a"
        name="leftSide"
        rules={[
          {
            required: true,
            message: "Поле є пустим!",
          }]}
      >
      <Input
        className='inputFunction'
        placeholder="Введіть межу..."
        maxLength={100}
      />
      </Form.Item>
```

Рисунок 2.31. Загальний вигляд компоненти головної сторінки.

В результаті компонент MainPage повертає код JSX, який визначає макет і форму для виконання обчислень. Він використовує компоненти Layout, Header, Content, Form, Input, Row і Button із Ant Design для створення структури сторінки. Для зручності введення даних було використано компоненту Form, а поля створено за допомогою компонентів Form.Item, що дозволило швидко та гнучко налаштувати такі поля вводу під власні вимоги. Фрагмен коду з використання Form можна побачити на рисунку 2.31.

2.5. Функціональність веб-додатка

Рисунок 2.32. Головна сторінка веб-додатка.

Для того, щоб здійснити обчислення, необхідно вказати межі для обчислень, наприклад від 0 до 1. Після цього потрібно вказати змінні оператора, наприклад *x*, *t*, та ввести сам оператор та правило його обчислення. На останньому кроці вводимо початкову функцію v_0. Далі потрібно підтвердити ввід, натиснувши кнопку «Обчислити». На це натиснення реагує функція MakeCalculation, та отримує в якості параметрів введені значення у Form (рисунок 2.32.). Після чого, надсилає запит на Back-end частину програми для подальших обчислень.

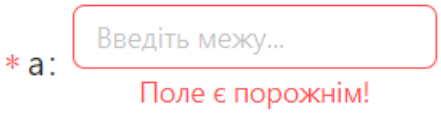


Рисунок 2.33. Порожнє поле вводу.

Якщо ж поле даних залишиться порожнім, з’явиться повідомлення. (Рисунок 2.33)

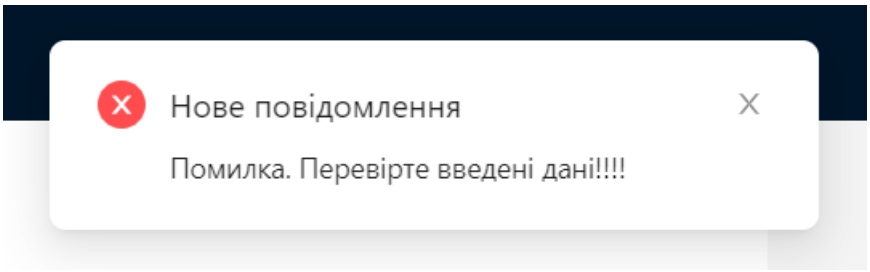


Рисунок 2.34. Повідомлення про помилку обчислень.

Або ж якщо дані введено неправильно та виникли проблеми в обчисленнях, з’являється повідомлення у верхній правій частині екрана про помилку. (Рисунок 2.34)

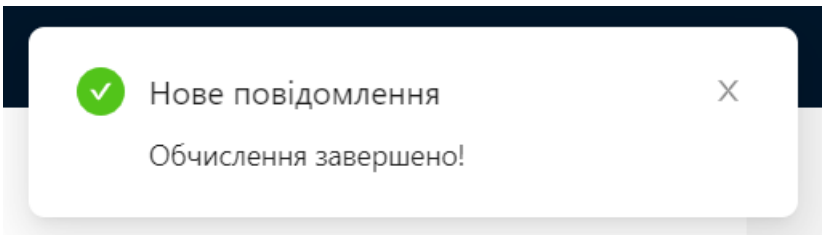


Рисунок 2.35. Успішне виконання обчислень.

В разі успішної валідації введених даних та успішного надсилання API запиту програма розпочне обчислення, спочатку порахувавши v_m, викликавши функцію GetVM. Слід додати, що запуск PythonEngine (Рисунок 2.36), рушія для обчислень на основі Python, відбувся при створені об’єкту сервісу CalculationService. Для його роботи ми підключаємо необхідні Python-коду модулі. Потім у рушій передаються змінні, які використовуються Python-скриптом. В кінці роботи Python-коду ми отримуємо з рушія потрібну змінну.

```
public CalculationService(ILogger<CalculationService> logger)
{
    Initialize();
    _logger = logger;
}

1 reference
private static void Initialize()
{
    string pythonDLL = @"C:\Users\Dmytro\AppData\Local\Programs\Python\Python311\python311.dll";
    Environment.SetEnvironmentVariable("PYTHONNET_PYDLL", pythonDLL);
    PythonEngine.Initialize();
}
```

Рисунок 2.36. Ініціалізація PythonEngine.

Далі відбуваються обчислення c_m за допомогою функції GetCM, де параметром слугує v_m та межі крайової задачі. Схожим чином буде здійснене обчислення μ_m через функцію GetMU, де параметром є c_m , де відбудеться перетворення mu з object на double[][]. Викликавши метод MainResults та передавши йому знайдені власні значення, уточнюємо їх використовуючи метод дотичних, для цього застосовуємо його у околі кожного з характеристичних чисел полінома. Далі ми обчислюємо значення u_m, викликавши функцію GetUN, передавши аргументом μ_m. На останньому кроці передаємо уточнені характеристичні полінома для формування власних функцій у метод GetPlot. Побудувавши їх, розбиваємо проміжок, що розглядається на 100 точок, та обчислюємо значення власних функцій в кожному з них. Формуємо об’єкт типу double[][][] із значеннями власних функцій для побудови графіків та відправляємо їх разом із двовимірним масивом власних значень для побудови таблиці на Front-end частину.

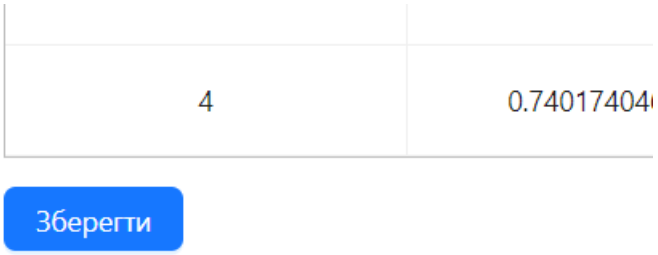
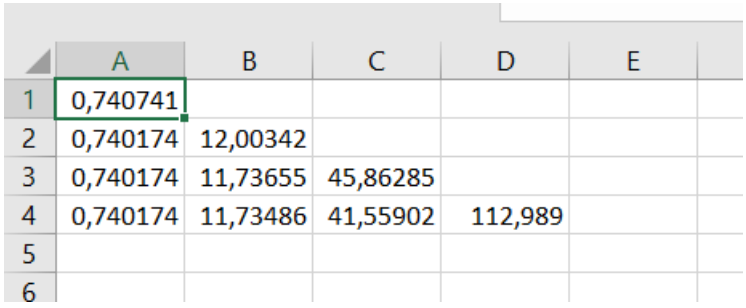


Рисунок 2.37. Кнопка збереження власних значень.

Знайдені власні значення можна зберегти у вигляді Excel-файла (Рисунок 2.38) натиснувши на кнопку “Зберегти під таблицею” (Рисунок 2.37).



	A	B	C	D	E
1	0,740741				
2	0,740174	12,00342			
3	0,740174	11,73655	45,86285		
4	0,740174	11,73486	41,55902	112,989	
5					
6					

Рисунок 2.38. Excel-файл із збереженими власними значеннями.

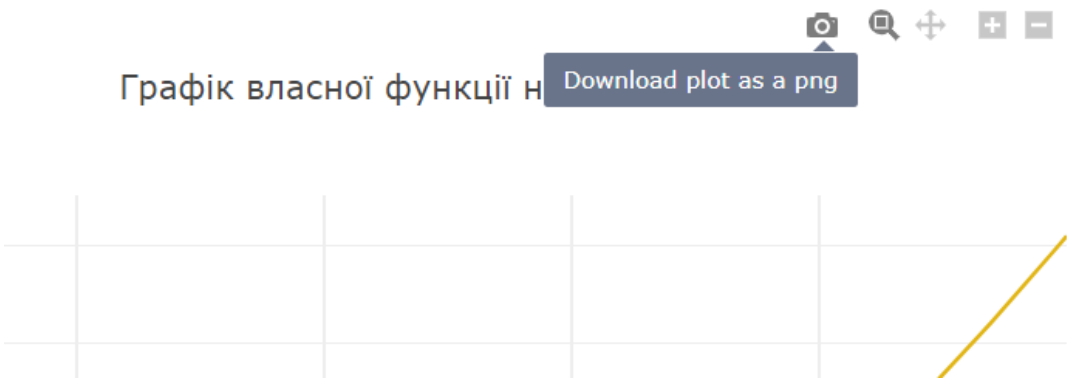


Рисунок 2.39. Кнопка для збереження графіків.

Крім того, завдяки багатим можливостям бібліотеки React-plotly, отримані графічні результати можна зберегти у PNG-форматі, натиснувши на зображення фотокамери у верхньому правому куті графа (Рисунок 2.39).

2.6. Приклад застосування веб-додатку

Рисунок 2.40. Вхідні параметри.

Як приклад, з допомогою нашого веб-додатку знайшли власні значення та побудували графіки власних функцій для розв'язку крайової задачі заданої функцією Гріна (1.27) із початковою функцією $\nu_0 = 1$. Результати пошуку власних значень веб-додаток надав у вигляді таблиці, яку зображено на рисунку 2.41 та для кожної ітерації відобразив графіки наближених власних функцій (рисунки 2.43-2.46). Для перевірки коректності роботи, вхідні параметри були використані одного із прикладів книги Ярошка С.А. "Модифікований метод послідовних наближень для спектральних задач".

Власні значення

Номер ітерації n	U1	U2	U3	U4
1	0.7407407407407408			
2	0.7401739236469439	12.003415819910465		
3	0.7401738844225363	11.736546538729298	45.86285259102849	
4	0.7401740469487463	11.734860972431697	41.55902283047085	112.98895240405491

Рисунок 2.41. Результати пошуку власних значень.

З рисунку 2.5 видно, що власне після п'ятої ітерації перше власне значення є близьким до точного, оскільки його різниця з попереднім відрізняється на шостому знаку після коми тобто менша 0,00001, для другого — 0,001. Для наближення третього та четвертого власного значення похибки уже більші.

номер ітерації n	Метод Коха		ММПН		
	$\mu_1^{(n)}$	$\mu_2^{(n)}$	$\mu_1^{(n)}$	$\mu_2^{(n)}$	$\mu_3^{(n)}$
1	0,75		0,740740741		
2	0,740741		0.740173924	12.00341582	
3	0,7402089	12,89	0.740173884	11.73654657	45.86286679
4	0,74017608	12,00	0.740173884	11.73486499	41.60080731
5	0,740174023	11,80	0.740173884	11.73486183	41.44108171
точні значення	0.740173884	11.73486183	0.740173884	11.73486183	41.43880785

Рисунок 2.42. Результат із книги.

Можемо помітити що результати власних значень є дуже близькими. Тож можна зробити висновок, що графіки достатньо точно відображають власні функції (Рисунки 2.43-2.46).

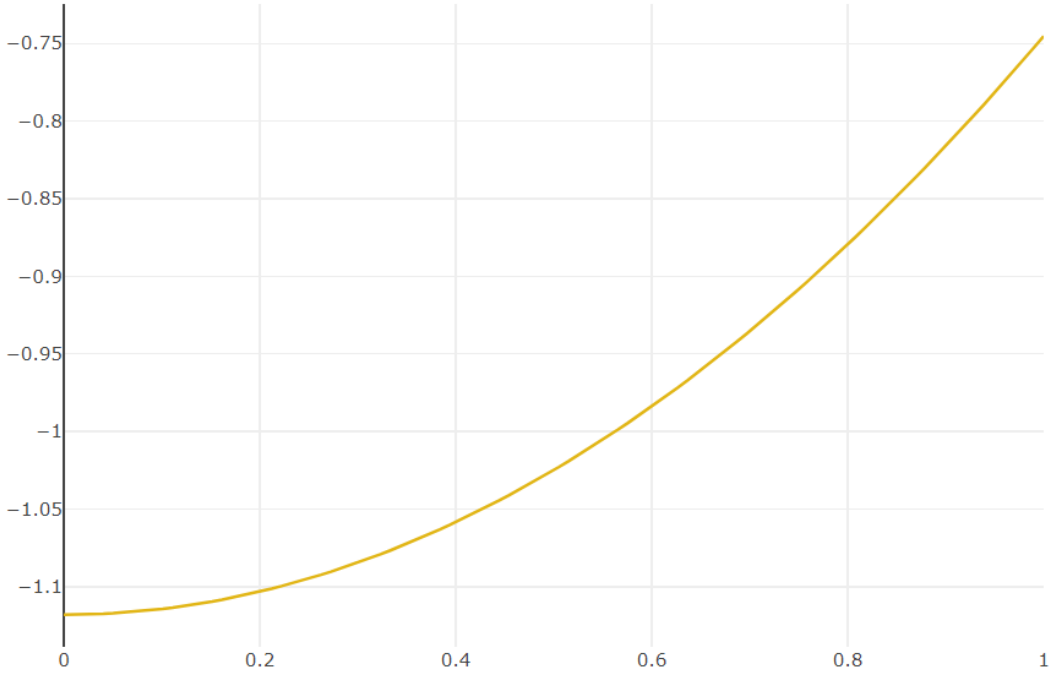


Рисунок 2.43. Графік власної функції на 1-ій ітерації

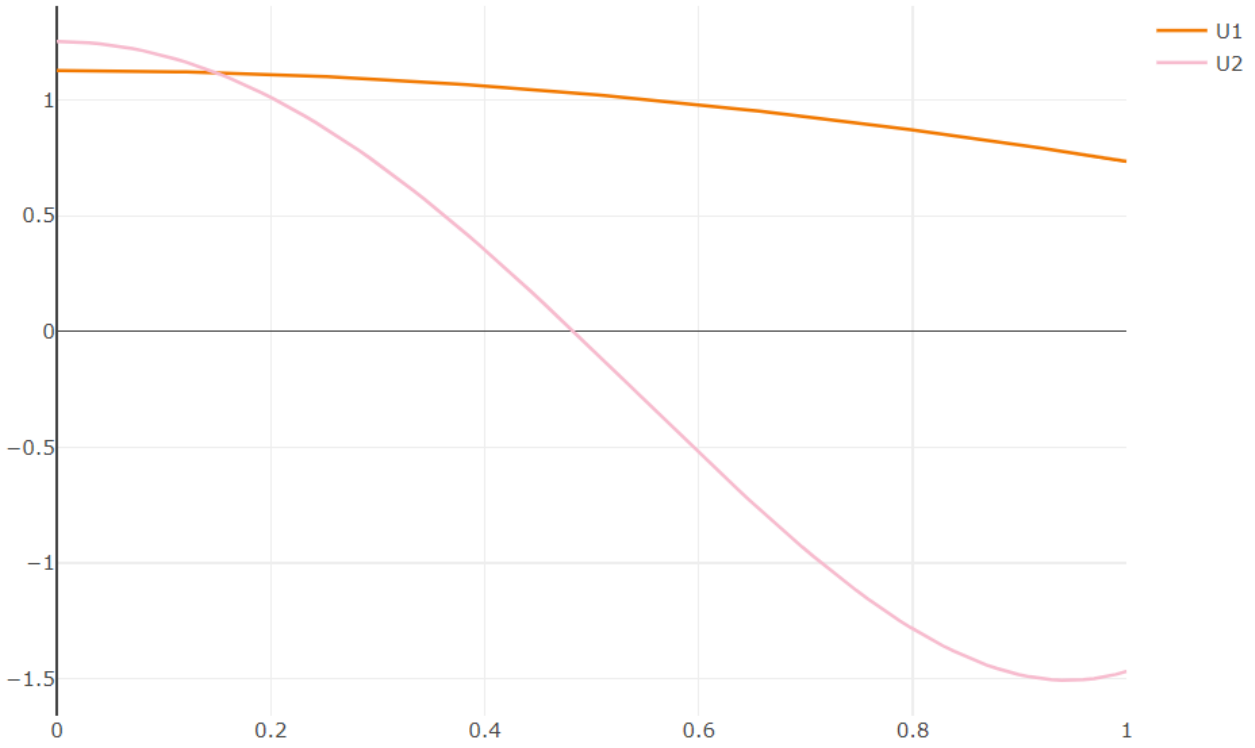


Рисунок 2.44. Графік власних функцій на 2-ій ітерації.

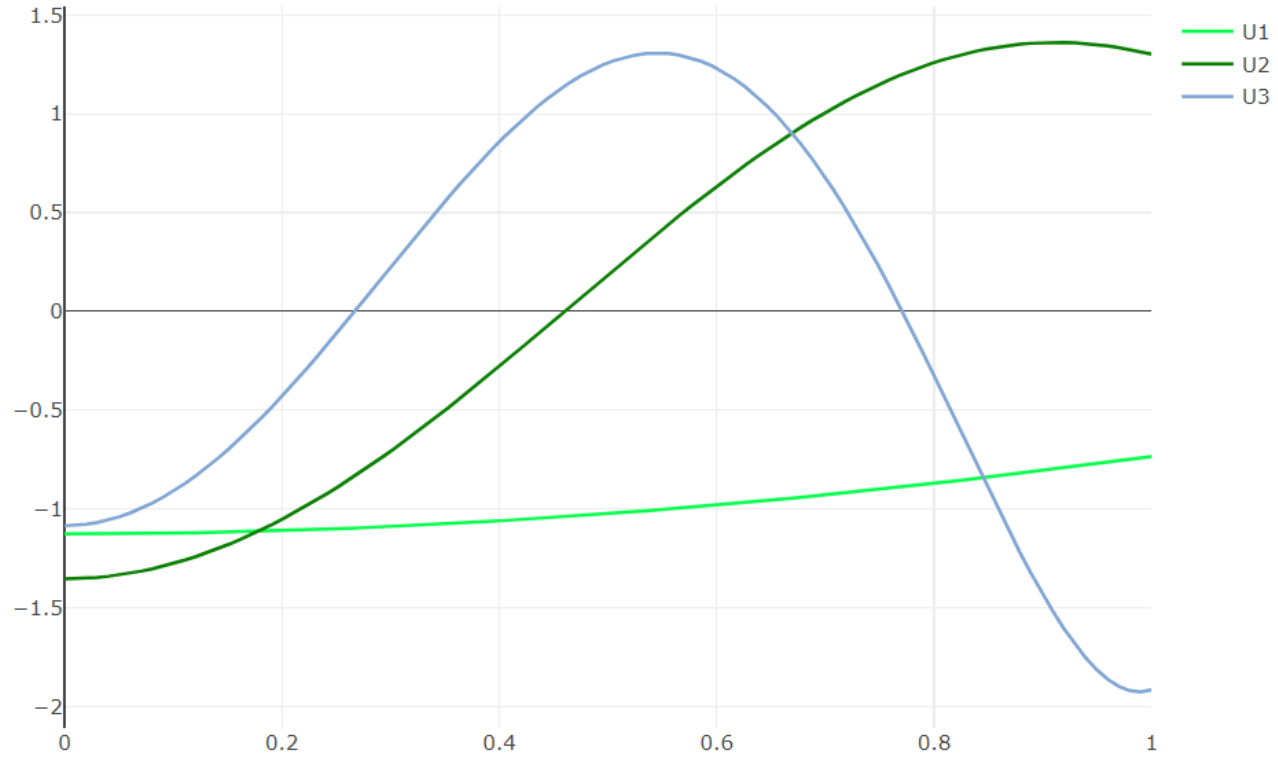


Рисунок 2.45. Графік власних функцій на 3-ій ітерації.

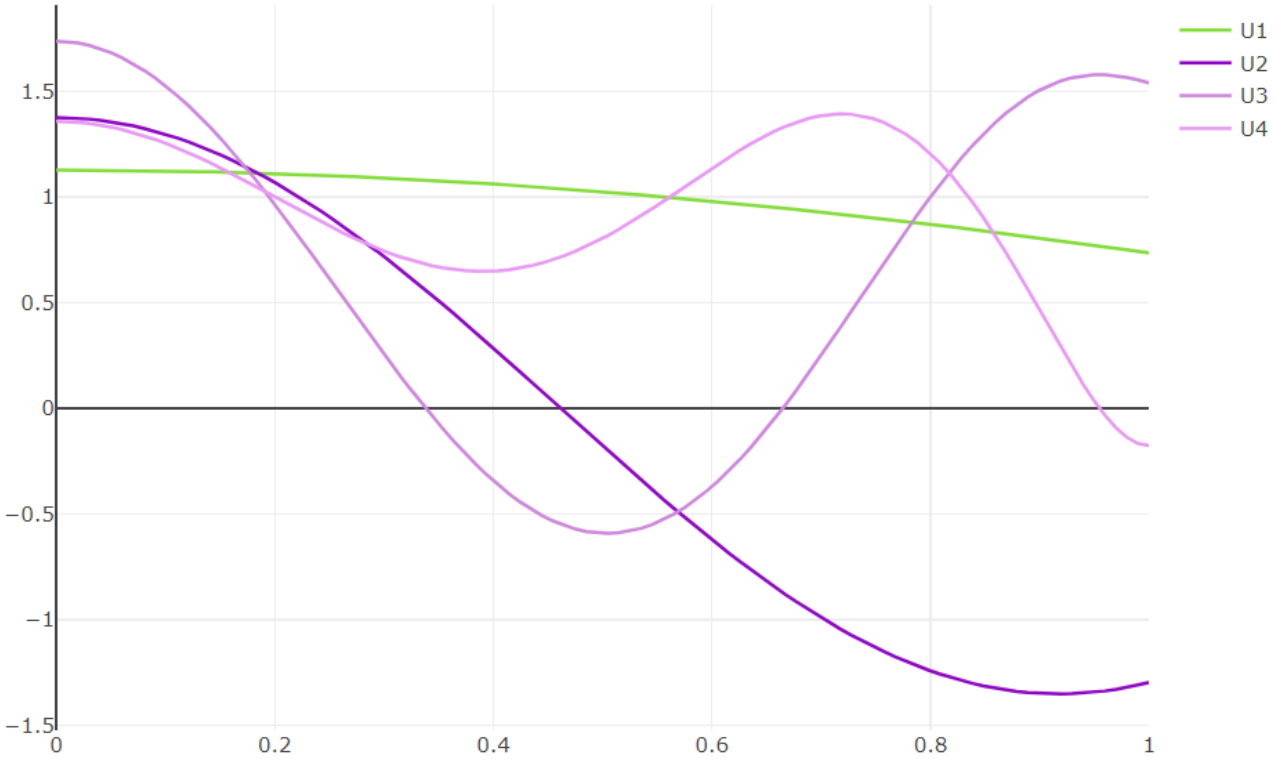


Рисунок 2.46. Графік власних функцій на 4-ій ітерації.

Протестуємо додаток застосувавши ММПН до задачі вигляду:

$$y = \mu \int_0^{\pi} G(x,t)y(t)dt$$

де

$$G(x,t) = \begin{cases} \dfrac{(2-t)x}{2}, x \le t \\ \dfrac{t(2-x)}{2}, x \ge t, \end{cases}$$

Обравши вхідну функцію $v_0(x) = 1$ отримаємо результати:

Власні значення				
Номер ітерації n	U1	U2	U3	U4
1	4.166666666666667			
2	4.116062476558736	31.4341862796574		
3	4.115858528291232	24.37850637694352	87.85765292618152	
4	4.115858366193294	24.141422386789365	65.26022449659526	235.40168552975024

Рисунок 2.47. Власні значення для чотирьох ітерацій.

Результати характеристичних чисел обчислюються з досить гарною точністю, при досить малій кількості кроків що зображено на рисунку 2.47. При цьому ми також маємо наближено обчислені інші характеристичні значення.

номер ітерації n	Степеневий метод, $\mu_1^{(n)}$	ММПН			
		$\mu_1^{(n)}$	$\mu_2^{(n)}$	$\mu_3^{(n)}$	$\mu_4^{(n)}$
1	4,8	4,166666667			
2	4,1667	4,116062477	31,43418628		
3	4,12270	4,115858528	24,37850638	87,85765293	
4	4,116932	4,115858366	24,14142238	65,26022330	235,4016604
5	4,1160399	4,115858366	24,13935078	63,73463220	139,3090487
6	4,11588829	4,115858366	24,13934205	63,66050780	124,5039154
точні значення	4,115858366	4,115858366	24,13934203	63,65910653	122,8891618

Рисунок 2.48 Власні значення із книжки.

У порівнянні з результатами із книжки бачимо, що знайдені власні значення є дуже близькими. Зобразимо також графіки власних функцій (Рисунок 2.49-2.52).

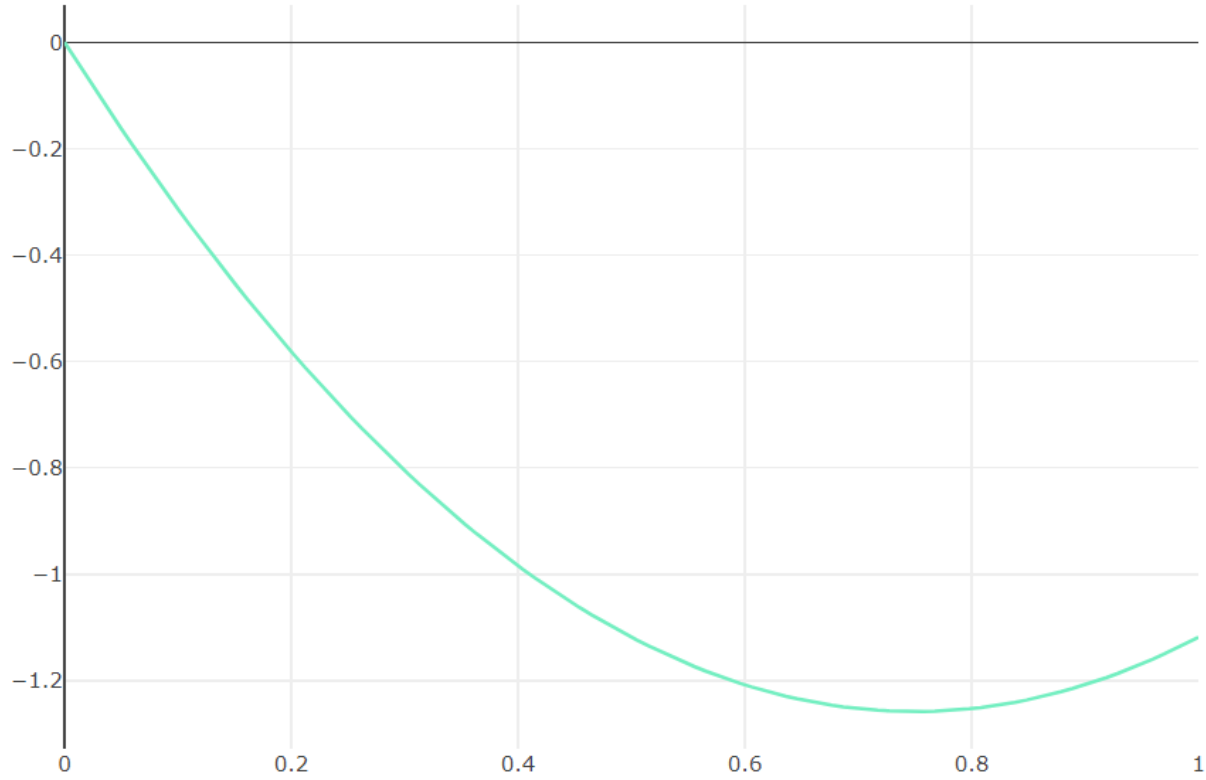


Рисунок 2.49. Графік власних функцій на 1-ій ітерації.

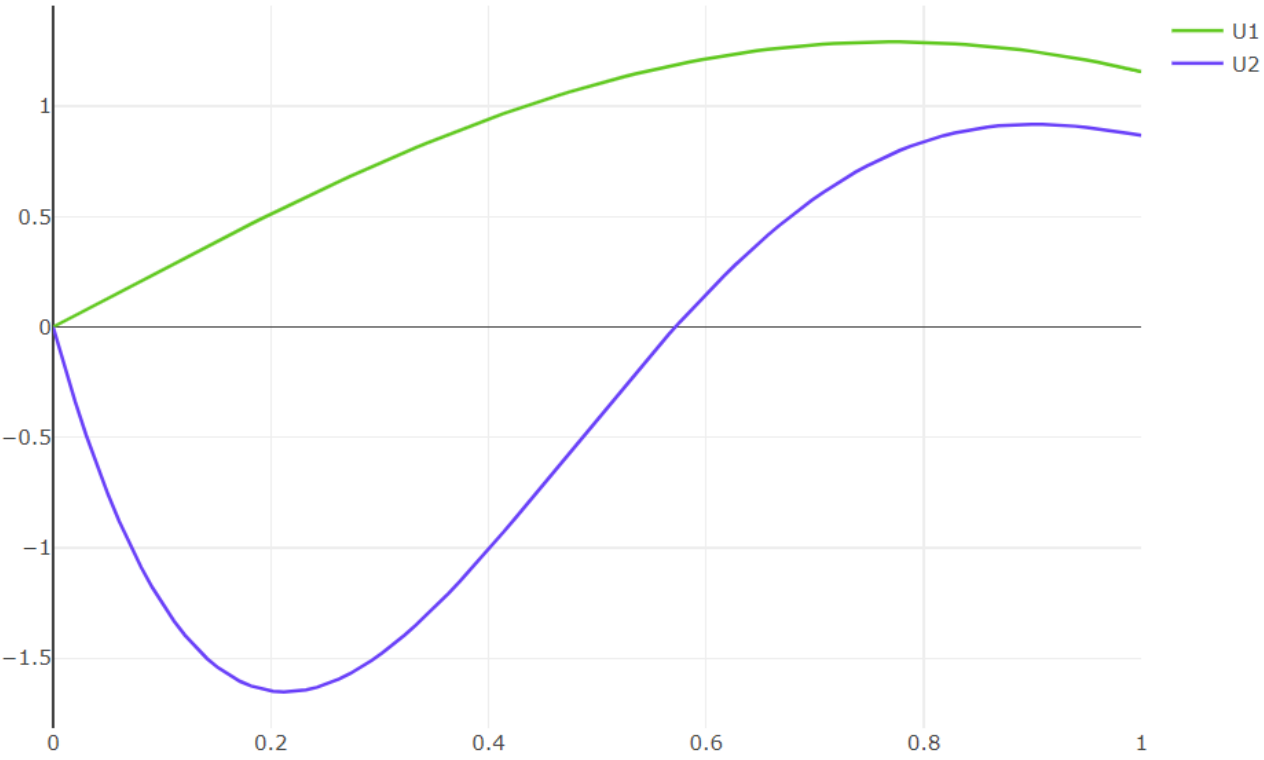


Рисунок 2.50. Графік власних функцій на 2-ій ітерації.

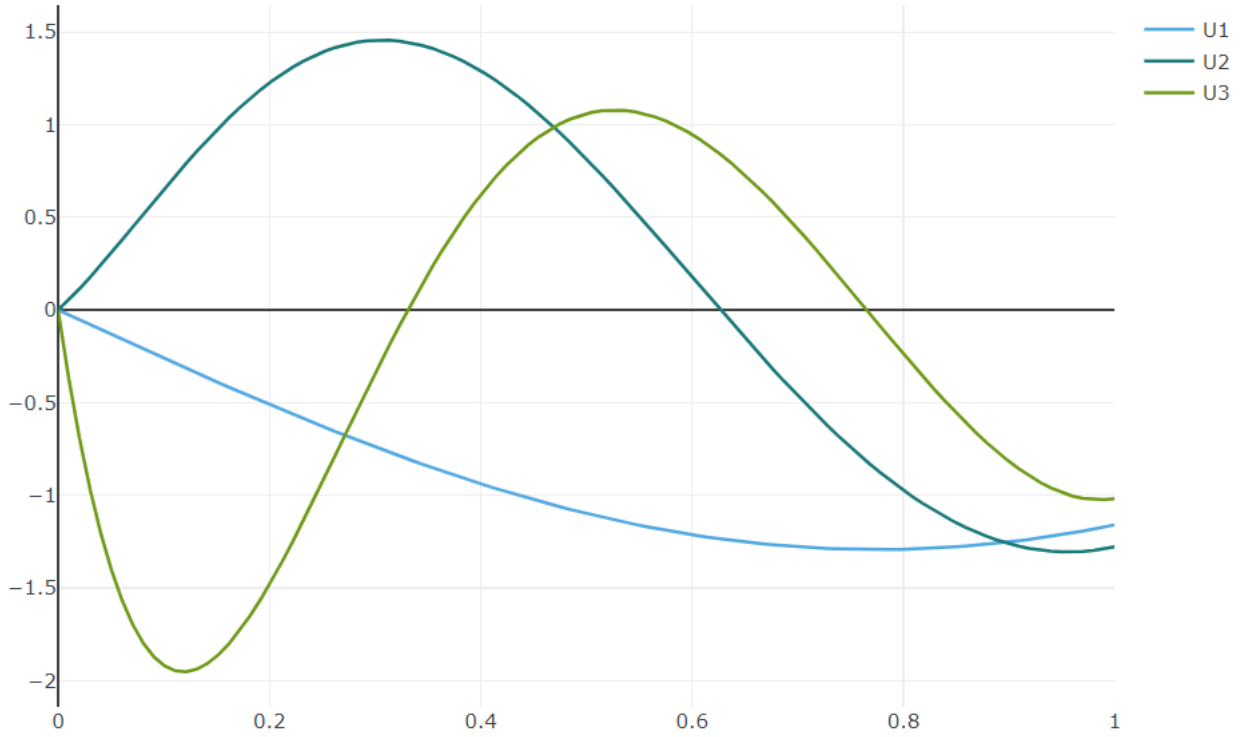


Рисунок 2.51. Графік власних функцій на 3-ій ітерації.

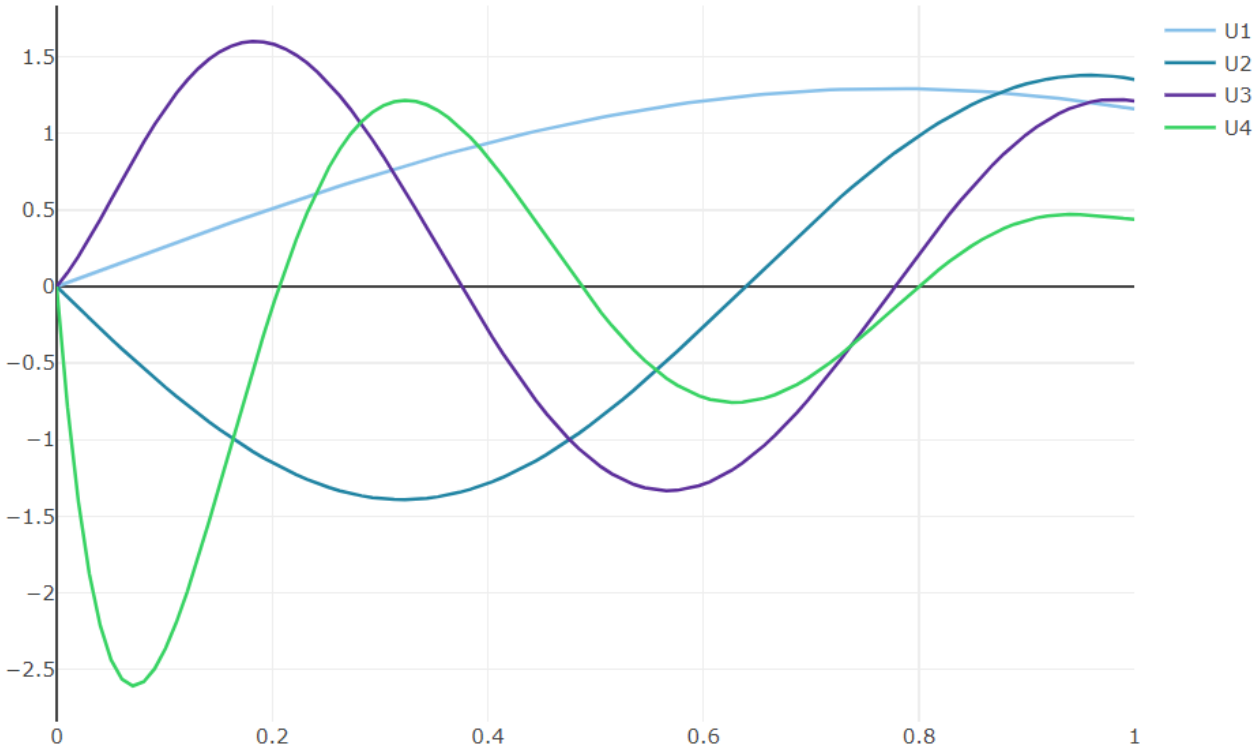


Рисунок 2.52. Графік власних функцій на 4-ій ітерації.

ВИСНОВОК

Під час роботи над дипломною роботою:

1) дослідив теоретичну частину задачі:

а) математичну постановку задачі про власні значення;

б) необхідні та достатні умови існування розв’язку;

в)схему алгоритму модифікованого методу послідовних наближень;

г) збіжність результатів ММПН до розв’язків задачі;

2) визначив методи і засоби реалізації алгоритму ММПН:

а) послідовність виконання обчислень проміжних значень;

б) необхідні програмні засоби та бібліотеки для реалізації ММПН;

3) розробив Python-програму з використанням :

а) Бібліотек для математичних обчислень, таких як: Numpy, Sciру, Sympy, re.

б) конвертації типів CLR;

4) розробив back-end (серверну частину) веб-додатку на основі ASP.NET Core Web API з використанням:

а) платформи .NET версії 7.0;

б) бібліотеки Python.NET;

5) розробив front-end (інтерфейс користувача) веб-додатку на основі React з використанням:

а) React-sweet-state для керуванням станом компонентів програми;

б) Ant Design - використання компонентів для інтерфейсу користувача;

в) React-plotly для відображення графіків;

За результатами дипломної роботи, було розроблено гнучкий та легкий у користуванні веб-додаток для виконання обчислень та отримання результатів модифікованого методу послідовних наближень в реальному часі з вікна веб-

браузера. Обчислення та взаємодія з користувачем забезпечується пов’язанням back-end та front-end частин програми.

При проведенні обчислень, за їх результатами встановлено, що перше та друге власні значення обчислюються із задовільною точністю при невеликій кількості ітерацій ММПН, графіки власних функцій на кожному наступному кроці наближаються до шуканих. Модифікований метод послідовних наближень при вдалому виборі початкової функції дозволяє порівняно швидко знайти наближений розв’язок спектральної задачі із хорошою точністю.

Під час виконання дипломної роботи набув нових вмінь у використанні математичного апарату для виконання поставленого завдання, поглибив навички роботи із ASP.NET Core Web API та одночасному застосуванні Python і .NET можливостями пакету Python.NET. Відкрив для себе нову Front-end бібліотеку React-plotly та значно краще зрозумів керування станом React додатку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ярошко С. Модифікований метод послідовних наближень для спектральних задач / Ярошко С., Ярошко С. — Львів: Lambert Academic Publishing, 2018. — 103 с.
2. Войтович Н. Н. Модификация метода последовательных приближений для однородных задач / Войтович Н. Н., Ровенчак А.И. // Ж. вычисл. матем. и матем. физики. — 1982. — Т.22, №22. — С.348-357.
3. Воробьев Ю. В. Метод моментов в прикладной математике — М.: Гос. издательство физ.-мат. Литературы, 1958. — 186 с.
4. Гончаров В. Л. Теория интерполирования и приближения функций — М.: Гос. узд-во техн.-теор. лит.-ры. 1954. — 328 с.
5. Краскевич В. Е. Численные методы в инженерных вычислениях / Краскевич В. Е., Зеленский К. Х., Гречко В. Н. — К.: Вища школа, 1986. — 263 с.
6. Коллати Л. Теория приближений . Чебышевские приближения и их приложения / Коллати Л., Крабс В. — М.: Наука, 1978. — 272 с.
7. Парлетт Б. Симметричная проблема собственных значений. — М.: Мир, 1983. — 382с.
8. Тимчмарш Е. Теория функций. — М.: Наука, 1980. — 464 с.
9. https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-7.0
10. https://pythonnet.github.io/pythonnet/
11. https://legacy.reactjs.org/
12. https://atlassian.github.io/react-sweet-state/#/
13. https://ant.design/
14. https://plotly.com/javascript/react/