Міністерство освіти та науки України
Національний університет “Києво-Могилянська академія”
Факультет природничих наук
Кафедра фізико-математичних наук

Кваліфікаційна робота

освітній ступень - бакалавр

на тему: «ЧИСЛОВЕ ДОСЛІДЖЕННЯ СТРУКТУРИ ШВИДКИХ МАГНІТОЗВУКОВИХ ВЛАСНИХ МОД У ТОКАМАКАХ»

Виконав студент 4 року навчання
спеціальності
104 Фізика та астрономія
Машинник Назар Сергійович

Керівник Яковенко Юрій Володимирович
доктор фізико-математичних наук,
доцент
Рецензент _Порицький П.В._
(прізвище та ініціали)

Кваліфікаційна робота захищена
з оцінкою _____
Секретар ЕК _____
« 03 » _червня__ 2025 р.

Київ – 2025

ЗМІСТ

Вступ

1) Актуальність проблеми

Фізика плазми змогла досягти результатів, які знайшли активне використання у багатьох областях. Це створює підвищений інтерес до більш точного розуміння процесів у плазмі і можливостей їх контролю. З численних видів хвиль, які виникають в плазмі, особливо важливими для розуміння динаміки плазми є швидкі магнітозвукові хвилі, також відомі як «стисні альвенові хвилі» (compressional Alfven waves)[1]. Швидкі магнітозвукові моди[2] – власні моди ШМХ, стаціонарні конфігурації магнітного поля, що відповідають внутрішнім резонансним структурам плазми. Порожнинні моди (cavity modes) спостерігаються в магнітосфері Землі та інших планет[3, 4]. У сонячному вітрі та магнітосфері планет ШММ відіграють роль у процесах перенесення енергії та збудженні магнітних бур. У сонячній короні вони можуть сприяти нагріванню плазми.

Властивість ШММ до перенесення енергії є однією з основних причин інтересу до них[5]. Саме з ними пов’язують йонно-циклотронну емісію з плазми токамаків[6]. Дослідження показали, що одним з нелінійних ефектів є аномальні втрати енергії у плазмі токамаків, що є критичним для стабільної роботи термоядерних реакторів. Інший варіант застосування ШММ - для діагностики стану плазми (МГД-спектроскопія). При збудженні мод швидкими йонами, їх можна сприймати зовнішніми антенами і використовувати їх для діагностики стану внутрішніх областей плазми, якщо є можливість надійно розраховувати їх частотний спектр.

Незважаючи на те, що ШММ відомі доволі давно, багато їх особливостей ще потребують вивчення. У складній тороїдальній геометрії, характерній для токамаків, прості аналітичні підходи часто виявляються недостатніми для точного опису поведінки ШММ. У зв’язку з цим важливо розробляти та вдосконалювати числові методи моделювання, що дозволяють точно враховувати вплив геометричних факторів, таких як тороїдальність і еліптичність поперечного перерізу плазми. Актуальність цієї теми обумовлена необхідністю забезпечення надійного теоретичного та числового підґрунтя для подальших експериментальних досліджень і практичного використання токамаків.

2) Мета роботи

Метою роботи є дослідження впливу геометрії плазми на просторову структуру ШММ.

3) Постановка задачі

У дипломній роботі досліджується вплив еліптичності перерізу на просторову структуру розв’язків.

Огляд літератури

Швидкі магнітозвукові моди (ШММ), також відомі як компресійні альвенівські моди (Compressional Alfvén Eigenmodes, CAE), виникають як власні коливання плазми в частотному діапазоні між частотою альвенівських хвиль та іонною циклотронною частотою. Їх існування теоретично передбачено в рамках ідеальної магнітогідродинаміки. ШММ експериментально виявлені в низці токамаків. Зокрема, в сферичному токамаку NSTX спостерігали нестійкості, що ідентифікуються як компресійні альвенівські моди. У роботі [1] було показано, що ці моди збуджуються пучком нейтральних частинок і демонструють типову структуру та частотний діапазон для ШММ.

Дослідження [2] показало, що високочастотні компоненти колективних коливань у термоядерній плазмі, зокрема ШММ, можуть збуджуватися взаємодією з високоефективними нагрівальними пучками частинок або іонним циклотронним резонансом. Модифікація властивостей цих мод в тороїдальній геометрії відкрила можливість їх локалізації як власних мод у певних просторових зонах плазми.

У токамаку DIII-D також реєструвались моди, характеристики яких відповідають ШММ [7]. Було встановлено, що ці моди можуть взаємодіяти з іонами та впливати на їх транспорт, що має значення для енергетичного балансу плазми.

У токамаку MAST зафіксовано явища, які підтверджують наявність ШММ у сферичних токамаках [8]. Ці результати стали додатковим підтвердженням універсальності існування ШММ у різних конфігураціях тороїдальної плазми.

ШММ у токамаках мають фізичну аналогію з порожнинними модами (cavity modes) в магнітосфері Землі. В роботі [3] було представлено спостереження таких мод, які виникають в обмежених магнітосферних областях. В роботі [4] було узагальнено результати досліджень для Землі і Юпітера, демонструючи, що порожнинні резонанси є універсальним явищем в магнітно утримуваних плазмах великого масштабу, що розширило розуміння механізмів існування ШММ у лабораторних умовах.

Рівняння для ШММ, частота яких є значно нижчою від йонно-циклотронної, можуть бути виведені з лінеаризованої системи рівнянь ідеальної МГД з урахуванням тороїдальності та неоднорідності плазми. У випадку частот, що наближаються до йонно-циклотронної або перевищують її, потрібно використовувати кінетичний опис плазми, в якому відгук частинок плазми на хвилю враховується тензором діелектричної проникності. У роботі [2] були описані базові підходи до аналізу власних мод у плазмі. Подальші теоретичні дослідження, зокрема [9], розширили ці підходи на випадки з тороїдальною геометрією, показавши можливість формування спектру ШММ через резонансну взаємодію з частинками. У пізніших роботах [1, 10] вивчались умови існування локалізованих компресійних мод, включно з ефектами геометрії, тиску та градієнтів щільності.

Для моделювання ШММ використовуються спеціалізовані числові коди. Один із класичних прикладів — NOVA [11], який базується на рівняннях ідеальної МГД і дозволяє досліджувати власні моди при частотах, що значно нижчі від іонної циклотронної. Проте його застосування обмежене через відсутність кінетичних ефектів і обмеження на діапазон частот. Новішим є код CAE3B, який був застосований для вивчення структури ШММ у сферичних

токамаках [12]. Цей код дозволяє враховувати складну геометрію, включно з еліптичністю та тороїдальністю, і є придатним для аналізу високочастотних компресійних мод.

ШММ мають здатність ефективно переносити енергію в плазмі, що робить їх важливими з погляду контролю над енергетичними потоками. У роботі [5] було проведено нелінійне моделювання збудження ШММ в NSTX, що показало, як моди можуть змінювати профілі енергії та впливати на динаміку швидких іонів. Це підтверджує важливість ШММ у процесах нагрівання та втрат енергії в сучасних експериментах.

Виведення рівнянь

Для розгляду плазми з концентричними циліндричними поверхнями потоку зручно використати циліндричну систему координат: $(r, \vartheta, y = R\varphi)$. Усі величини, що описують хвилі, будуть задовольняти умову періодичності:

$$X(r, \vartheta, \varphi = 0) = X(r, \vartheta, \varphi = 2\pi).$$

Лінії магнітного поля на поверхнях потоку задовольняють наступне рівняння:

$$\vartheta - \iota(r)\varphi = 0,$$

де ι – магнітне число обертів (magnetic winding number, $\iota = \frac{1}{q}$, q – safety factor).

Далі буде використано два локальних базиси для кожної точки: $(e_r, e_\vartheta, e_\varphi)$, пов'язаний з циліндричними координатами; (e_r, e_ϑ, b), пов'язаний з магнітною лінією, де $b = \frac{B}{B}$ – дотична компонента, e_r - нормальна, e_ϑ – бінормальна. Можна показати, що:

$$b = b_\varphi e_\varphi + b_\vartheta e_\vartheta = \frac{1}{\sqrt{1+\epsilon^2\iota^2}} e_\varphi + \frac{1}{\sqrt{1+\epsilon^2\iota^2}} e_\vartheta,$$

$$e_b = b_\vartheta e_\vartheta - b_\vartheta e_\varphi,$$

де $\epsilon = \frac{r}{R}$.

Проекції оператора ∇ мають наступний вигляд:

$$\nabla_\vartheta = e_\vartheta \cdot \nabla = \frac{1}{r}\frac{\partial}{\partial\vartheta}, \nabla_\varphi = e_\varphi \cdot \nabla = \frac{1}{R}\frac{\partial}{\partial\varphi},$$

$$\nabla_\parallel = b \cdot \nabla = \frac{b_\varphi}{R}\left(\frac{\partial}{\partial\varphi} + \iota\frac{\partial}{\partial\vartheta}\right), \nabla_b = e_b \cdot \nabla = \frac{b_\varphi}{R}\left(\frac{\partial}{\partial\vartheta} - \iota\epsilon^2\frac{\partial}{\partial\varphi}\right).$$

Також далі буде використано ротор:

$$\nabla \times e_r = \nabla \times e_\varphi = 0, \nabla \times e_\vartheta = \frac{1}{r} e_\varphi.$$

З рівнянь для локальних базисів тепер можна отримати:

$$\nabla \times e_b = b_\varphi \nabla \times e_\vartheta - e_\vartheta \times \nabla b_\varphi + e_\varphi \times \nabla b_\vartheta = \alpha_\parallel b_b + \alpha_b e_b,$$

$$\nabla \times b = b_\vartheta \nabla \times e_\vartheta - e_\varphi \times \nabla b_\varphi + e_\vartheta \times \nabla b_\vartheta = \beta_\parallel b_b + \beta_b e_b,$$

де:

$$\alpha_{||} = \frac{b_\varphi^2}{r} = \frac{1}{r} - \bar{\alpha}, \bar{\alpha} = \frac{b_\theta^2}{r} \sim \frac{\epsilon^2}{r},$$

$$\alpha_b = -\frac{b_\varphi^2}{rR}\frac{d}{dr}(r^2\iota) \sim \frac{\epsilon}{r}, \beta_\parallel = -\alpha_b, \beta_b = -\bar{\alpha}.$$

Можна знехтувати повздовжньою компонентою електричного поля:

$$E = E_r e_r + E_b e_b.$$

$$\nabla \times E = E_b \nabla \times e_b - e_b \nabla \times E_b - e_r \nabla \times E_r$$
$$= -e_r \nabla_\parallel E_b + e_b(\nabla_\parallel E_r + \alpha_b \mathrm{E}_b) + b(\hat{k}_r \mathrm{E}_b - \nabla_b \mathrm{E}_r)$$

$\hat{k}_r$ – оператор, який буде використовуватись у наступних формах, в залежності від ситуації:

$$\hat{k}_r X = \frac{\partial X}{\partial r} + \alpha_\parallel X = \frac{1}{f}\frac{\partial}{\partial r}(fX) = \frac{1}{r}\frac{\partial}{\partial r}(rX) - \bar{\alpha}X, f = \exp(\int dr \alpha_\parallel).$$

Тоді модуль ротора електричного поля:

$$|\nabla \times E|^2 = |\nabla_\parallel E_b|^2 + |\nabla_\parallel E_r + \alpha_b \mathrm{E}_b|^2 + |\hat{k}_r \mathrm{E}_b - \nabla_b \mathrm{E}_r|^2$$

З рівнянь Максвелла

$$\frac{i\omega}{c} B = \nabla \times E, \frac{i\omega}{c} \varepsilon E = -\nabla \times B,$$

де ε – тензор діелектричної проникності, з використанням проекцій базисних векторів можна отримати:

$$B_r = \frac{i\omega}{c} \nabla_\parallel E_b, B_b = -\frac{i\omega}{c}(\hat{k}_r \mathrm{E}_b - \nabla_b \mathrm{E}_r),$$

$$\nabla \times B = B_b \nabla \times e_b - e_b \times \nabla B_b + B_\parallel \nabla \times b - b \times \nabla B_\parallel - e_r \times \nabla B_r.$$

Використавши $r-$ і $b-$ проекції, можна отримати:

$$\varepsilon_1 E_r + i\varepsilon_2 E_b = \frac{ic}{\omega}(\nabla_b B_\parallel - \nabla_\parallel B_b),$$

$$-i\varepsilon_2 E_r + \varepsilon_1 E_b = \frac{ic}{\omega}(\hat{k}_r \mathrm{B}_b - \nabla_b \mathrm{B}_r + \beta_\parallel B_\parallel).$$

Далі буде використано припущення, що хвильові величини у цьому випадку мають форму $X(r,\vartheta,\varphi) = Y(r)\exp(im\vartheta - in\varphi - i\omega t)$, і наступні заміни:

$$\nabla_{\|} = ik_{\|}, \nabla_b = ik_{b,}$$

$$k_{\|} = b_\varphi \frac{m\iota - n}{R} \sim \frac{\epsilon}{r}, k_{\|} = b_\varphi \frac{m - \iota\epsilon^2 n}{r} \sim \frac{1}{r}$$

В результаті можна отримати систему рівнянь:

$$\bar{\varepsilon}_1 E_r + i\bar{\varepsilon}_2 E_b = -\frac{ck_b}{\omega} B_{\|},$$

$$-i\bar{\varepsilon}_2 E_r + (\bar{\varepsilon}_1 - \frac{c^2}{\omega^2}\alpha_b^2)E_b = -i\frac{c}{\omega}(\nabla_r B_{\|} - \beta_b B_{\|}),$$

де $\bar{\varepsilon}_1 = \varepsilon_1 - \frac{c^2}{\omega^2}k_{\|}^2, \bar{\varepsilon}_2 = \varepsilon_2 + \frac{c^2}{\omega^2}k_{\|}\alpha_b.$

Розв'язавши її, можна знайти E_r і E_b:

$$E_r = -\frac{c}{\omega D}[\left(\bar{\varepsilon}_1 - \frac{c^2}{\omega^2}\alpha_b^2\right)k_b B_{\|} + \bar{\varepsilon}_2(\nabla_r B_{\|} - \beta_b B_{\|})],$$

$$E_b = -\frac{ic}{\omega D}[\bar{\varepsilon}_1(\nabla_r B_{\|} - \beta_b B_{\|}) + \bar{\varepsilon}_2 k_b B_{\|}].$$

$$D = \bar{\varepsilon}_1^2 - \bar{\varepsilon}_2^2 - \bar{\varepsilon}_1\frac{c^2}{\omega^2}\alpha_b^2.$$

Підставивши ці розв'язки у b −проекцію, можна отримати рівняння для $B_{\|}$:

$$\frac{\omega^2}{c^2}B_{\|} = -\frac{1}{r}\frac{\partial}{\partial r}\left(rQ_1(r,\omega)\frac{\partial B_{\|}}{\partial r}\right) + S(r,\omega)B_{\|},$$

де:

$$S = Q_1 k_b^2 - \frac{1}{r}\frac{\partial}{\partial r}(r\bar{\alpha}Q_1) - \frac{1}{r}\frac{\partial}{\partial r}(rk_b Q_2) - \frac{c^2}{\omega^2}k_b^2\frac{\alpha_b^2}{D} + \bar{\alpha}^2 + 2Q_2 k_b\bar{\alpha},$$

$$Q_1 = \frac{\bar{\varepsilon}_1}{D}, Q_2 = \frac{\bar{\varepsilon}_2}{D}.$$

Головним недоліком цього рівняння є вплив знаменника, який значно погіршує точність розв'язку біля границі внутрішнього відбиття. Цю проблему можна вирішити, якщо перейти до двох поперечних компонент E і після знаходження їх отримати $B_{\|}$ числовим диференціюванням. Компоненти електричного поля в

полярних координатах є розривними біля полюса, якщо поле там ненульове. Тому було вибрано вивести рівняння для проєкцій E на дві квазіортогональні координати, ξ та η, які майже збігаються з парою R, z, але трохи нахилені, щоб бути ортогональними до магнітної силової лінії:

$$e_\xi = \left[1 - \left(1 - b_\varphi\right)\frac{b_x^2}{b_\parallel^2}\right] e_x + \frac{\left(b_\varphi - 1\right) b_x b_z}{b_\parallel^2} e_z - b_x e_\varphi,$$

$$e_\eta = \left[1 - \left(1 - b_\varphi\right)\frac{b_x^2}{b_\parallel^2}\right] e_z + \frac{\left(b_\varphi - 1\right) b_x b_z}{b_\parallel^2} e_y - b_z e_\varphi,$$

$$E = E_\xi e_\xi + E_\eta e_\eta.$$

Після громіздких, але простих обчислень можна отримати лагранжіан для варіаційної задачі: $L = \text{К} - \text{П} = \frac{1}{2}\frac{\omega^2}{c^2} E^T \cdot \varepsilon \cdot E - \frac{1}{2} |\nabla \times E|^2$.

Аналітичний розв’язок для циліндричної конфігурації

У випадку при $\iota = const, \rho(r) = const, \varepsilon_1 = \varepsilon_1(\omega), \varepsilon_2 = \varepsilon_2(\omega)$, залишивши доданки до порядку $O(\epsilon^2)$, рівняння для $B_\parallel$ спрощується до рівняння Бесселя першого роду:

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial B_\parallel}{\partial r}\right) + G(\omega)B_\parallel - \frac{\bar{m}^2(\omega)}{r^2}B_\parallel = 0,$$

де:

$$\bar{m}^2 = m^2\left(1 - \frac{c^2}{\omega^2\bar{\varepsilon}_1}\frac{4\iota}{R^2}\right),$$

$$G = \frac{\omega^2 D}{c^2\bar{\varepsilon}_1} - m\iota\frac{2n - m\iota}{R^2} + \frac{\bar{\varepsilon}_2}{\bar{\varepsilon}_1}\iota\frac{2n - m\iota}{R^2} - \frac{\bar{\varepsilon}_2}{\bar{\varepsilon}_1}\frac{2m\iota^2}{R^2} + \frac{2\iota^2}{R^2}$$

$B_\parallel(r = a) = 0$, тому:

$$B_\parallel(r) = J_{\bar{m}}(\sqrt{G}r),$$

де $J_{\bar{m}}$ – функція Бесселя першого роду порядку $\bar{m}$,

$$J_{\bar{m}}\left(\sqrt{G(\omega)}a\right) = 0,$$

$$j^2_{(\bar{m},l)} = G(\omega)a^2,$$

де $j_{(\bar{m},l)}$ – нуль $J_{\bar{m}}$ під номером l. Це можна використати для пошуку власних частот і порівняння точності числових розв’язків.

Алгоритм для числового знаходження розв’язку

Лагранжіан для варіаційної задачі має наступний вигляд: $L = \text{К} - \text{П}$, де

$$\text{К} = E^T \cdot B(\lambda) \cdot E$$

$$\text{П} = \frac{1}{2}\left(\left|\frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z}\right|^2 + \left|\frac{\partial E_\varphi}{\partial z} + \frac{in}{R}E_z\right|^2 + \left|\frac{\partial E_\varphi}{\partial x} + \frac{in}{R}E_x + \frac{E_\varphi}{R}\right|^2\right)$$

Для отримання розв’язку використовується комбінація методу оберненої векторної ітерації і методу відношення Релея. Метод оберненої векторної ітерації гарантовано знаходить найближче власне значення. Метод відношення Релея сходиться швидше, бо на кожному кроці коригує наближення для власного значення, використовуючи відношення Релея, але він може отримати не найближче власне значення, якщо початковий вектор дуже несхожий на розв'язок. Програма робить кілька обернених векторних ітерацій, щоб наблизити вектор до справжнього розв’язку, а потім використовує відношення Релея. Залежність $B(\lambda)$ є нелінійною, тому необхідно використовувати лінеаризацію. Кожна зовнішня ітерація починається з побудови нових матриць, лінеаризованих відносно отриманого наближення. Після цього запускаються внутрішні ітерації для розв'язання стандартної задачі на власні значення для побудованих розріджених матриць. Оскільки матриці є дуже великими, але розрідженими, для них нераціонально застосовувати стандартні методи пошуку власних значень. Для побудови матриці лінеаризованої задачі і знаходження її LU-розкладення використовується бібліотека для роботи з розрідженими матрицями (scipy.sparse).

В результаті розв’язком є електричне поле у вигляді $E_\xi = f_\xi\varphi(x,z) + g_\xi\psi(x,z), E_\eta = f_\eta\varphi(x,z) + g_\eta\psi(x,z)$, і магнітне поле

$B_{\|}$, отримане з електричного за допомогою числового диференціювання. Спеціальні масиви-вказівники класу Matrix_Builder забезпечують правильну відповідність між індексами в об'єднаному векторі та окремих векторах $f_\xi, g_\xi, f_\eta, g_\eta$.

Залежність власної частоти від магнітного числа обертів

Було проведено розрахунки при наступних умовах: $B_0 = 2.6, a = 30, ellipticity = 1, R_0 = 300, n_0 = 2 * 10^{-12}$. За допомогою одновимірного коду було знайдено 9 власних частот: для $m = 0,1,2$ і радіальних квантових чисел $l = 0,1,2$ при $\iota = 0$. Після цього за допомогою двовимірного коду було знайдено власні частоти для $\iota = 0, 0.1,0.2,0.3,0.4,0.5$, з використанням на кожному кроці попередньо отриманого значення власної частоти як початкового наближення.

Результати:

```
l  init  1d       2d iota=.0   iota=.1   iota=.2   iota=.3   iota=.4   iota=.5
m=0
0  4   -> 4.043  ->    3.895  -> 4.151  -> 3.986  -> 4.381  -> 4.083  -> 3.965
1  19  -> 19.428 ->    17.746 -> 17.753 -> 17.679 -> 17.772 -> 17.788 -> 17.806
2  46  -> 46.37  ->    44.89  -> 44.708 -> 44.655 -> 44.733 -> 44.942 -> 45.279
m=1
0  8   -> 8.513  ->    8.552  -> 8.545  -> 8.541  -> 8.537  -> 8.534  -> 8.532
1  28  -> 28.524 ->    28.671 -> 28.651 -> 28.624 -> 28.623 -> 28.622 -> 28.626
2  59  -> 59.977 ->    60.484 -> 60.55  -> 60.405 -> 60.405 -> 60.392 -> 60.423
m=2
0  15  -> 15.288 ->    15.358 -> 15.331 -> 15.312 -> 15.301 -> 15.297 -> 15.3
1  41  -> 41.059 ->    41.283 -> 41.226 -> 41.183 -> 41.160 -> 41.162 -> 41.187
2  78  -> 78.242 ->    78.597 -> 79.03  -> 78.986 -> 78.919 -> 78.969 -> 79.044
```

У вигляді графіків:

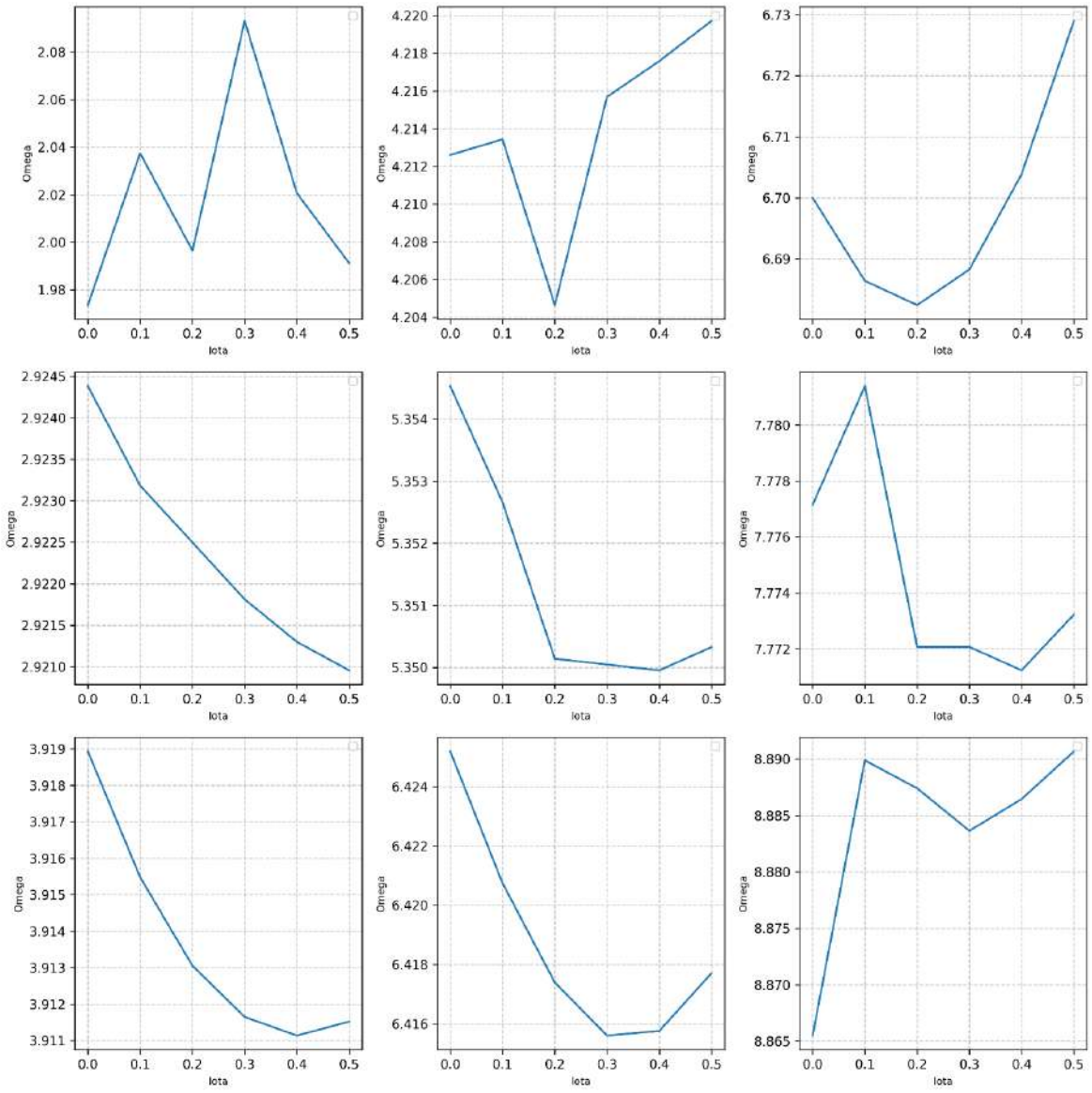

Рисунок 1: Залежність власних частот від магнітного числа обертів.

Зміна власної частоти дуже мала. Це є достатньою причиною зробити висновок про відсутність залежності від магнітного числа обертів. У аналітично знайдених залежностях власних частот[12] показано, що доданки, залежні від ι, є дуже малими порівняно з іншими (порядку $\frac{a^2}{R^2}$), що є одним з підтверджень коректності отриманих результатів.

Дослідження впливу еліптичності циліндра на структуру мод

Було проведено розрахунки при наступних параметрах: n = 30, iota0 = 0, delta = 0, lambda0 = [60, 100, 150, 200]. Отримано графіки електричного і магнітного полів при зміні параметра ellipticity = [1, 1.5 ,2]:

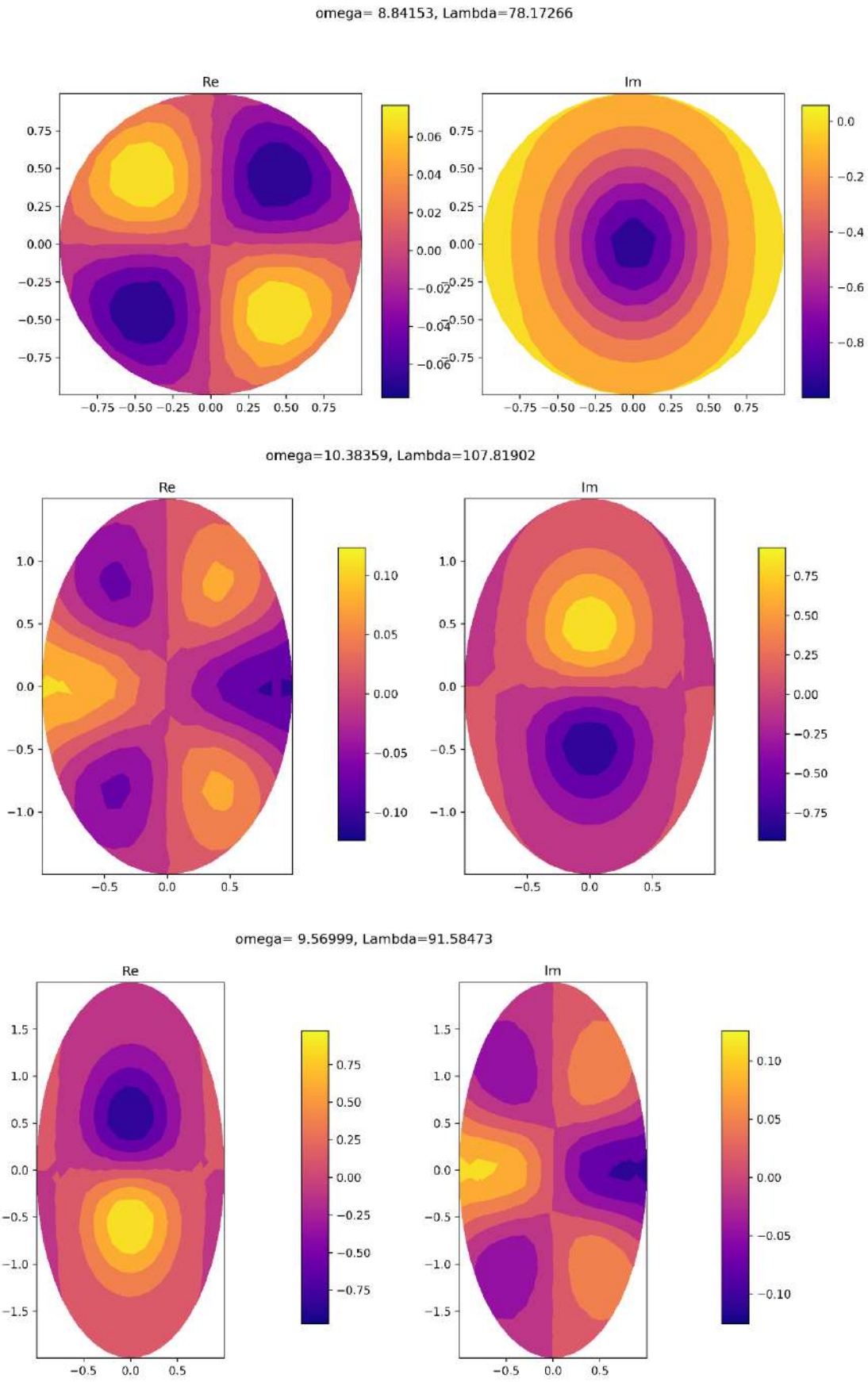

Рисунок 2: Графіки електричного поля, lambda0=60.

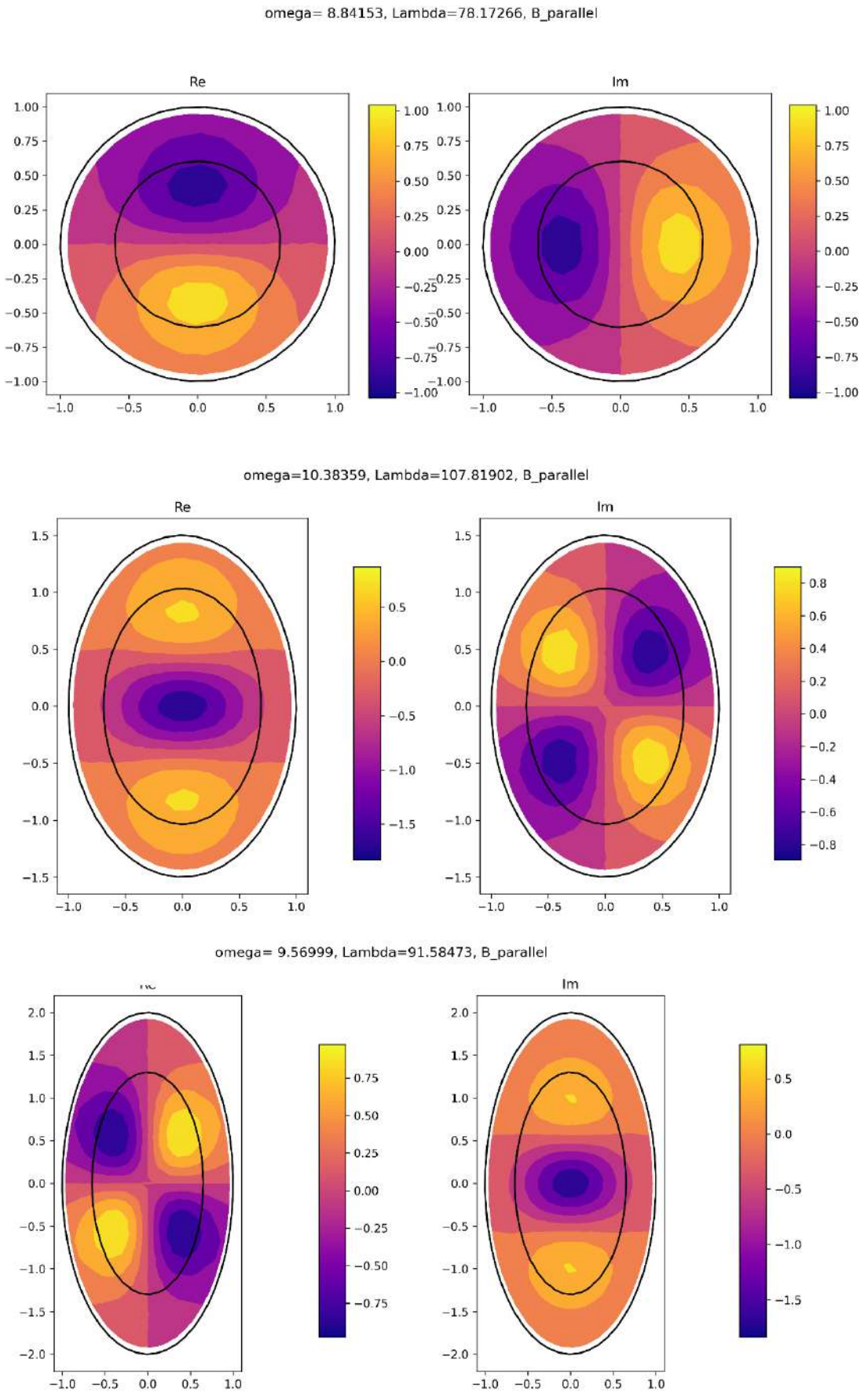

Рисунок 3: Графіки магнітного поля, lambda0=60.

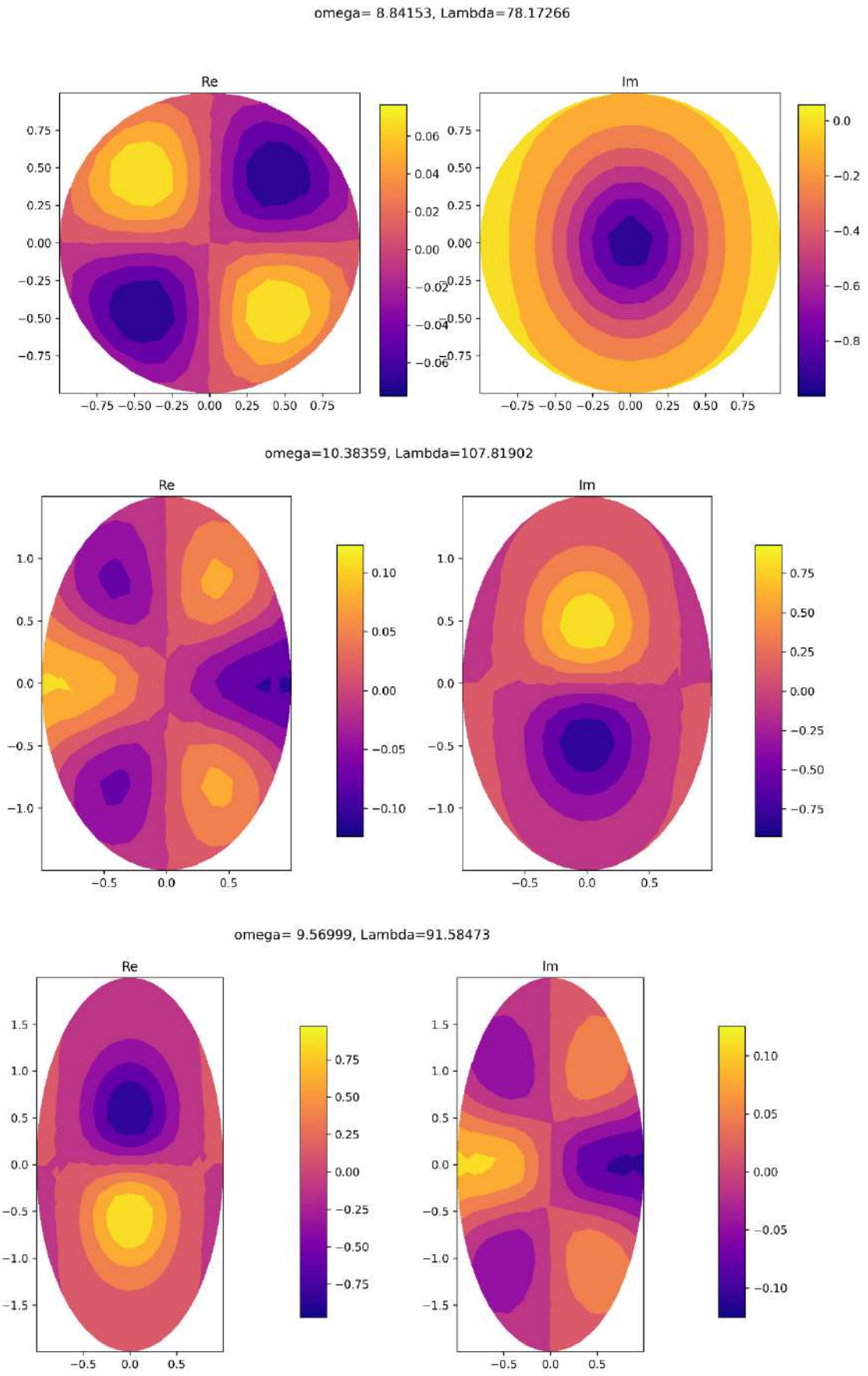

Рисунок 4: Графіки електричного поля, lambda0=100.

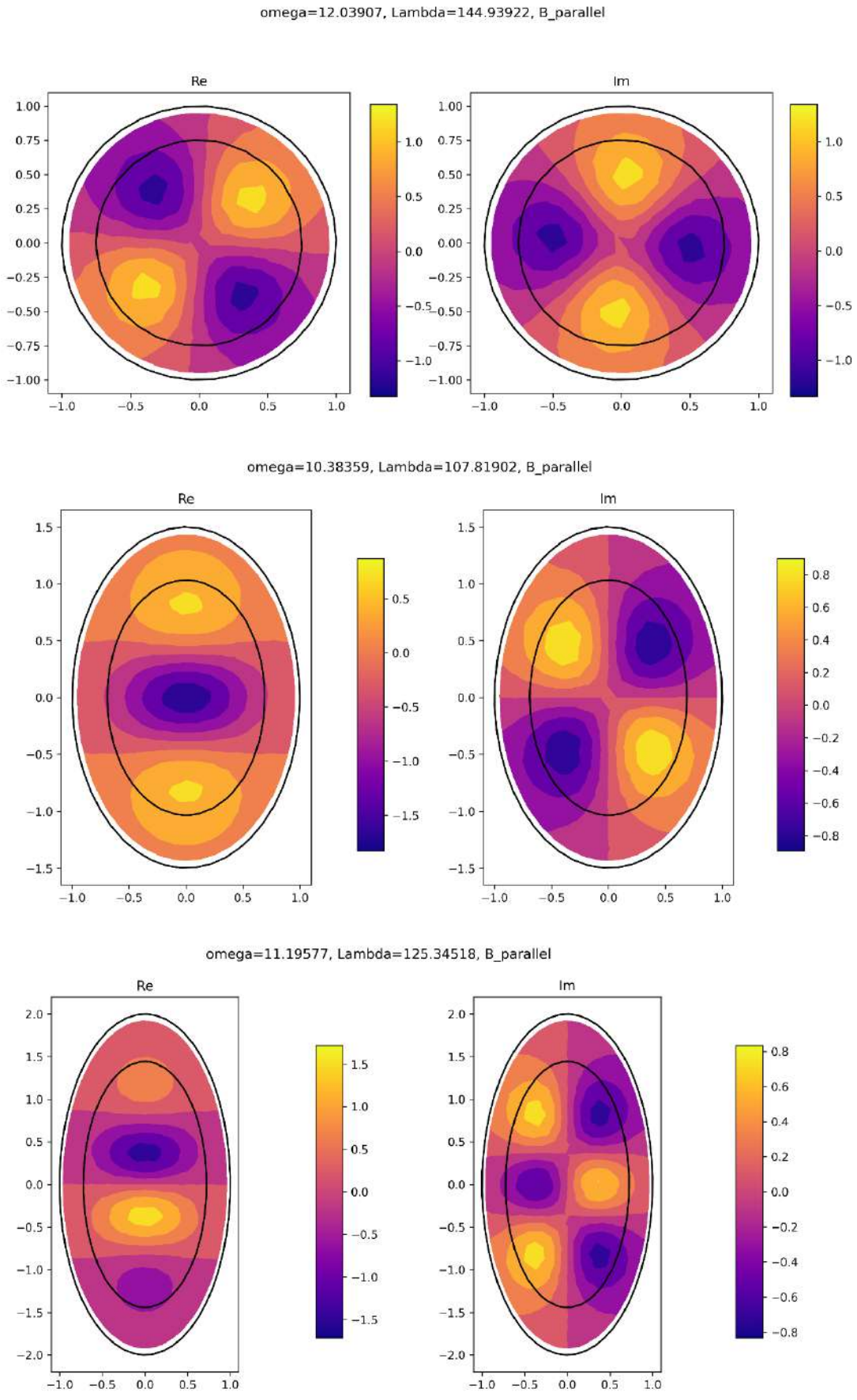

Рисунок 5: Графіки магнітного поля, lambda0=100.

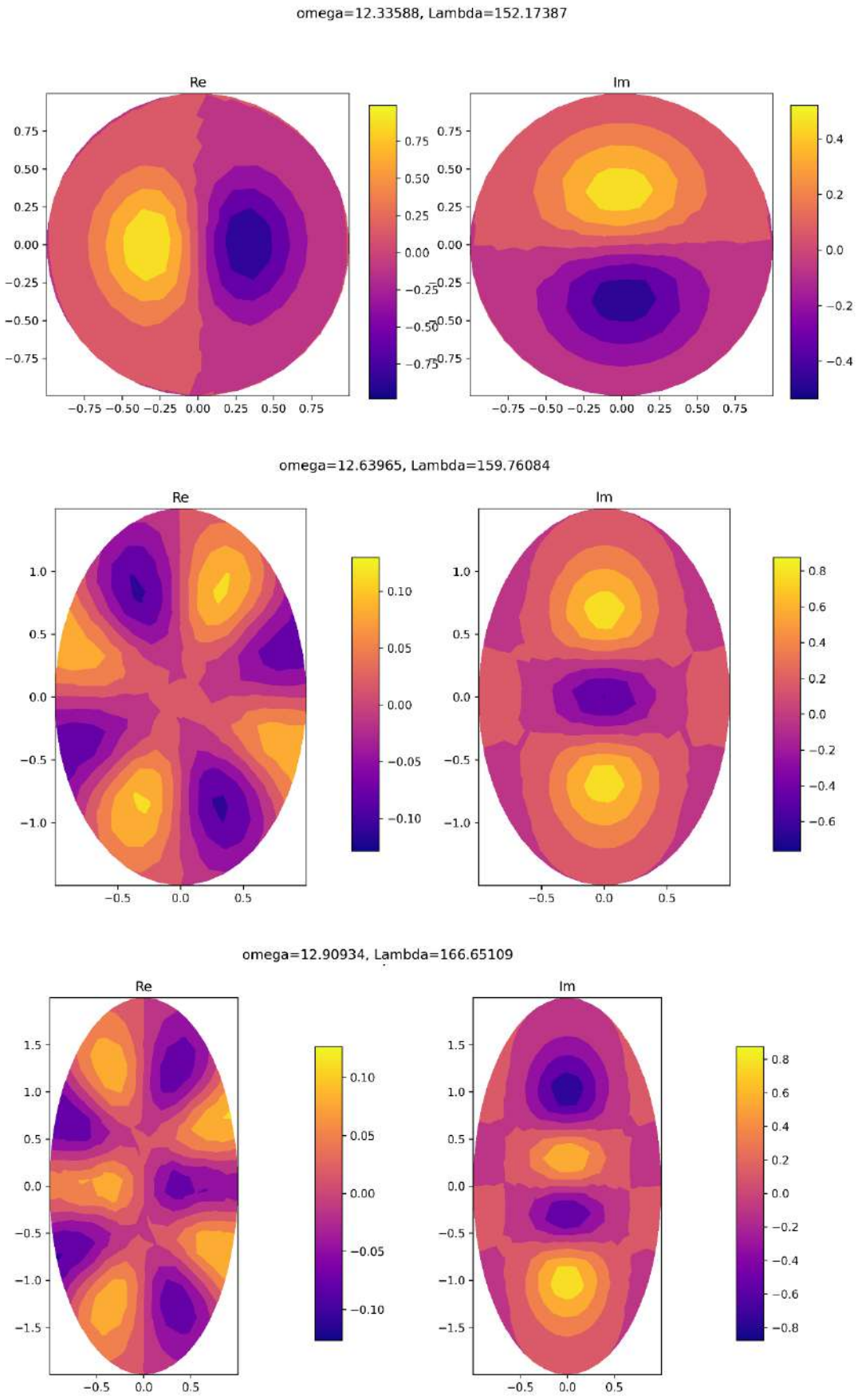

Рисунок 6: Графіки електричного поля, lambda0=150.

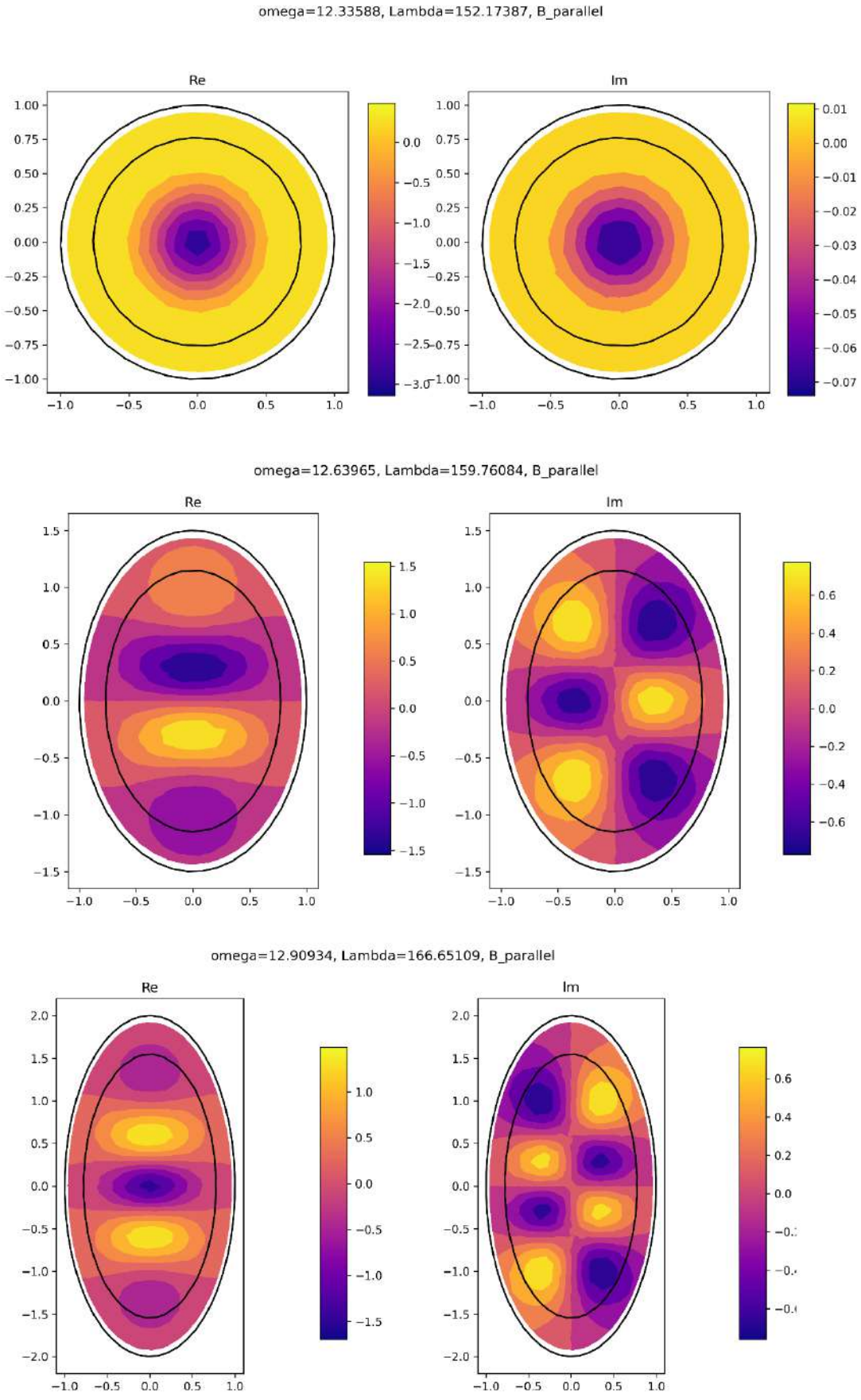

Рисунок 7: Графіки магнітного поля, lambda0=150.

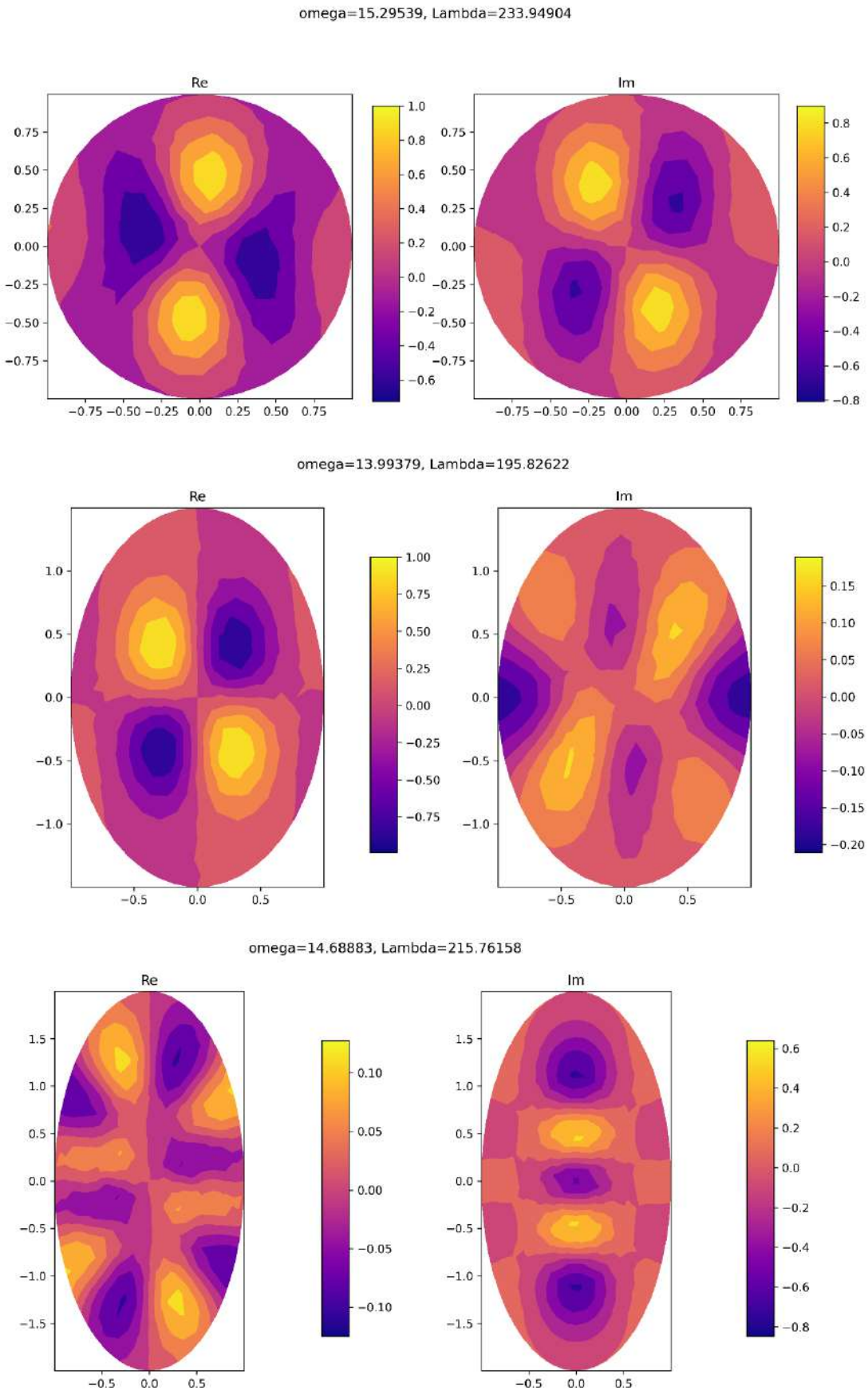

Рисунок 8: Графіки електричного поля, lambda0=200.

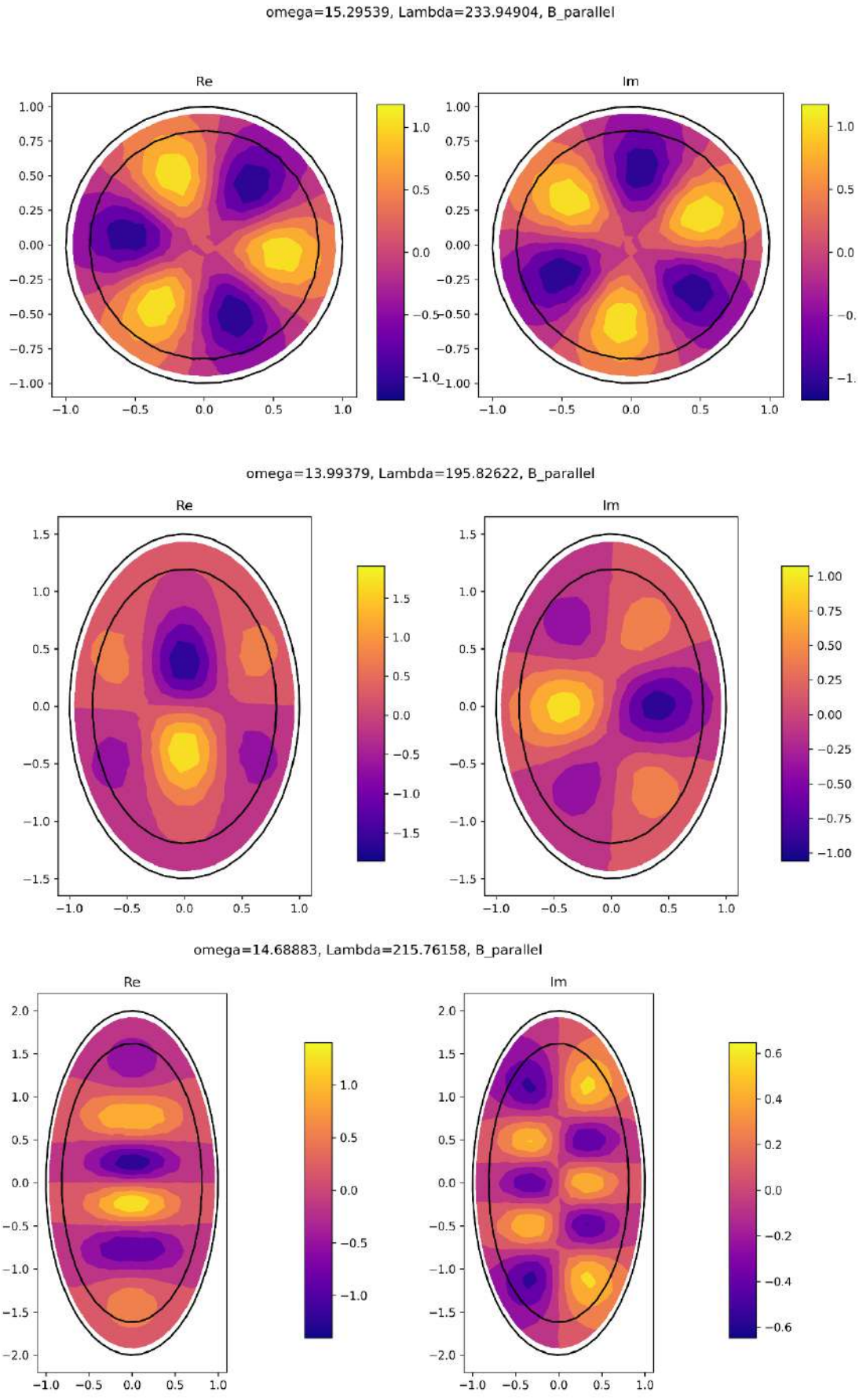

Рисунок 9: Графіки магнітного поля, lambda0=200.

Добре видно, що структура розв’язків є дуже схожою на розв’язки для двовимірного квантового гармонічного осцилятора, при цьому зростання еліптичності має вплив, аналогічний до зростання квантових чисел.

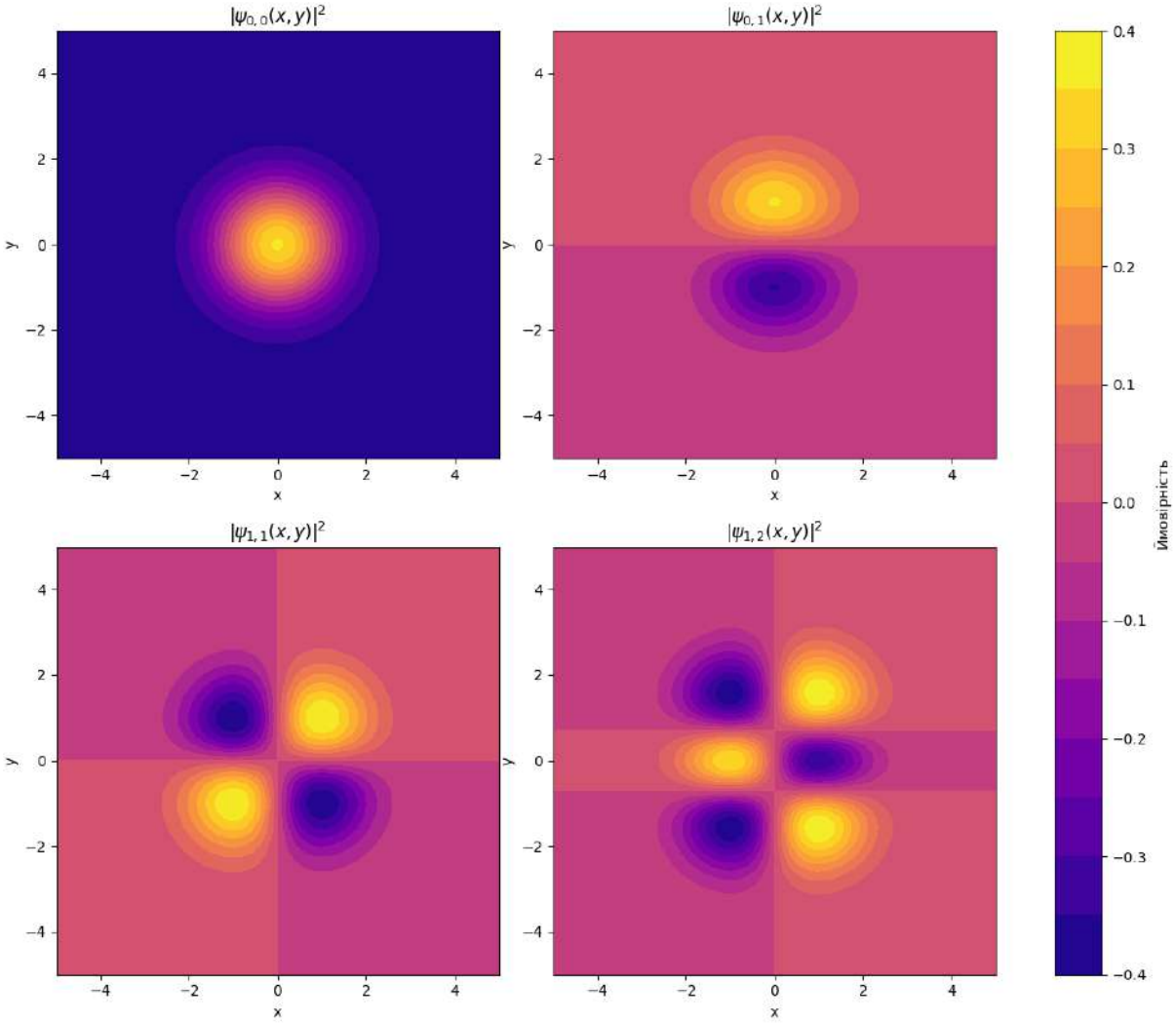

Рисунок 10: Графіки розв’язків двовимірного квантового гармонічного осцилятора для комбінацій квантових чисел (0,0), (0,1), (1,1), (1,2).

Двовимірний квантовий гармонічний осцилятор допускає власні функції, в яких квантовими числами є номери нулів розв'язку на осях x та y. У такій системі квантовими числами виступають n_x та n_y — кількості вузлів вздовж відповідних осей. Це можливо завдяки тому, що координати x та y в рівнянні розділяються. Енергія такого осцилятора має вигляд:

$$E = h\nu(a_x n_x + a_y n_y + 1),$$

де коефіцієнти a_x та a_y визначаються через коефіцієнти перед відповідними квадратичними доданками в виразі для потенціальної енергії.

У осесиметричному випадку, коли $a_x = a_y$, рівняння можна переписати в полярних координатах. Завдяки симетрії, хвильові функції можна представити у вигляді:

$$\psi(r, \theta) = \psi_m(r)\exp(im\theta),$$

де m — обертальне квантове число. Повна енергія в цьому випадку виражається як $E = h\nu a(n + 1)$, де $n = n_x + n_y$, тобто вона залежить тільки від суми квантових чисел, а не від їх розподілу. Це створює виродженість рівнів — наприклад, для $n = 1$ існує два незалежні стани з $(n_x, n_y) = (0,1),\ (n_x, n_y) = (1,0)$, які можна комбінувати в обертальні моди з певним m. Отже, в симетричному випадку можливі два варіанти поведінки системи, з квантуванням руху за кожною координатою окремо та з квантуванням за обертальним моментом.

Рівняння ШММ є складнішими. Якщо переріз еліптичний, строго розділити не вдається ані полярні, ані декартові координати. Однак результати розрахунків свідчать про те, що на якісному рівні можна спостерігати перехід від розв’язків обертального типу у круглому перерізі до розділення хвильових рухів за декартовими координатами з зростанням еліптичності.

Висновки

Було розроблено числовий метод для знаходження швидких магнітозвукових мод у плазмі з довільною формою поперечного перерізу. Метод базується на лінеаризованих рівняннях МГД і використовує скінченноелементну апроксимацію.

Проведено перевірку точності методу на круглій геометрії шляхом порівняння з розв'язком у циліндричній конфігурації. Показано відсутність суттєвої залежності власної частоти від магнітного числа обертів у розглянутому діапазоні параметрів. Це узгоджується з аналітичними оцінками, в яких внески, пов'язані з тороїдальністю, є малими.

Виявлено якісний перехід у структурі мод при зміні еліптичності поперечного перерізу:

У випадку кругової симетрії мода має обертальну структуру, що відповідає розв'язкам у полярних координатах. Зі зростанням еліптичності моди стають більш схожими на власні функції гармонічного осцилятора в декартовій системі координат.

Запропоновано фізичну інтерпретацію спостереженого переходу: еліптичність перерізу змінює виродженість рівнів за обертальним моментом, зміщуючи систему до опису в координатному представленні.

Встановлено, що еліптичність значно ускладнює гармонічний склад мод. У той час як у симетричних випадках моди складаються переважно з однієї гармоніки з незначними сателітами, у випадку еліптичного перерізу моди мають широкий спектр полоїдних гармонік. Це має важливе фізичне значення, оскільки гармонічний склад визначає резонансні умови взаємодії

моди зі швидкими йонами — основним механізмом їх збудження в токамаках.

Практична цінність результатів полягає в можливості точніше передбачати та аналізувати локалізацію і збудження ШММ у сучасних експериментах, а також у потенційному покращенні методів діагностики стану плазми через врахування геометричних ефектів.

Література

[1] N.N. Gorelenkov et al., "Compressional Alfven eigenmode instability in NSTX." Nuclear Fusion 42 (2002) 977.

[2] B. Coppi, "High-energy components and collective modes in thermonuclear plasmas." Phys. Fluids 29 (1986) 4060

[3] G. Crowley, W. J. Hughes, T. B. Jones, “Observational evidence of cavity modes in the Earth’s magnetosphere”, J. Geophys. Res., 92 (1987) 12233–12240.

[4] R. L. Lysak “Field Line Resonances and Cavity Modes at Earth and Jupiter”. Front. Astron. Space Sci. 9 (2022) 913554.

[5] E. V. Belova, N. N. Gorelenkov, "Nonlinear simulations of beam-driven compressional Alfvén eigenmodes in NSTX", Phys. Plasmas 24 (2017) 042505

[6] JET Team, "Fusion energy production from a deuterium-tritium plasma in the JET tokamak", Nucl. Fusion 32 (1992)

[7] W. W. Heidbrink et al., Nucl. Fusion 46 (2006) 324.

[8] L. C. Appel et al., Plasma Phys. Control. Fusion 50 (2008) 115011

[9] N.N. Gorelenkov, C.Z. Cheng, "Alfven cyclotron instability and ion cyclotron emission." Nucl. Fusion 35 (1995) 1743

[10] O. S. Burdo, Ya. I. Kolesnichenko, Phys. Lett. A 384 (2020) 126825

[11] C. Z. Cheng, M. S. Chance, “NOVA: A nonvariational code for solving the MHD stability of axisymmetric toroidal plasmas”, J. Comput. Phys. 71 (1987) 124

[12] H.M. Smith, E. Verwichte. Compressional Alfven eigenmode structure in spherical tokamaks, Plasma Phys. Control. Fusion 51 (2009) 075001

Додаток

Вміст файлу `plasma.py`:

```
from nep import *
from basis import *
from device import *
from grid import *
from grqi import *
from gauss import *
from iota import *
from matrix import *
from coefficients import *
import numpy as np
import matplotlib.pyplot as plt

def T():
    values["region_a"]=0.
    values["region_b"]=1.
    values['nPoints']=41
    values['device_B0']=2.6
    values['device_a']=30.
    values['device_R0']=300.
    values['device_n0']=2.e12
    values['device_ellipticity']=1.
    values['m']=0
    values['n']=30
    values['iota0'] = 0.
    values['delta'] = 0.
    values['right_boundary'] = 'Et0'
    values['step'] = .15
    values['spread'] = 1.1
    values['theta1'] = np.pi/2
    values['theta_shift'] = .9
    values['cylindrical'] = 1
    values['tolerance'] = 1.e-4
    values['max_iteration'] = 20
    values['gamma'] = 1.
    values['pedestal'] = .05
    return

def replacer(k,v):
    try:
        if isinstance(values[k],int):values[k]=int(v)
        elif isinstance(values[k],float):values[k]=float(v)
        elif isinstance(values[k],str):values[k]=v
        else:print("Wrong input, try again")
    except:print("Wrong input, try again")
    return

def M():
    print("The list of replacable values")
```

```
    for (k,v) in zip(values.keys(),values.values()):print(f"{k}:{v}")
    while True:
        print("""Enter your command:
          Q to exit
          <value name> <space> <value> to replace a selected value""")
        s=input()
        if s=='Q':return
        s=s.split()
        if len(s)==2:replacer(s[0],s[1])
        else:print("Wrong input, try again")

def F():
    print("Enter the name of file to  get the values from.\nEvery line
should look like this:\n<value name> <space> <value>")
    name=input()
    try:
        with open(name,'r') as file:
            s=file.readlines()
            for s_ in s:
                s__=s_.split()
                if len(s__)==2:replacer(s__[0],s__[1])
    except:print("Wrong input, try again")
    return

def S(save=False,lambda0=0):
    device=Device("D",values["device_B0"],values["device_a"],values["devi
ce_R0"],values["device_n0"],values["device_ellipticity"])
    GRQI_parameters = {'nIterations': 10,
                   'tolerance': 1.e-3,
                   'II_iter': [7,1],
                   'II_tol': None,
                   'factorize': False,
                   'seed': None}
    grid = Triangular_Grid_in_Ellipse (device.a1, device.a2,
values['step']*device.a1, values['spread'], values['theta1'],
values['theta_shift'])
    iota = Parabolic_q_iota (iota0=values['iota0'],
delta=values['delta'])
    density =
Quasiparabolic_Density_Profile(values['gamma'],values['pedestal'],device)
    coeff = FMW_coefficients (device, density, values['n'],
values['right_boundary'], iota,cylinder=(values['cylindrical']==1))
    basis = Quadratic_Basis2D (grid,
fixed_boundary=(coeff.boundary_type=='fixed'), quad_order=5)
    matrix = Matrix_Builder (coeff, basis, sparse=True)
    problem = Nonlinear_EigenProblem (basis,matrix, values['tolerance'],
values['max_iteration'], GRQI_parameters=GRQI_parameters)
    problem.run(save,lambda0)

def test_ellipticity():
    for l in [60, 100, 150, 200]:
        for e in [1,1.5,2]:
            replacer('device_ellipticity',e)
            S(save=True,lambda0=l)
```

```
    return

def test_toroidness():

    return

def run():
    print("""Enter your command:
          Q to exit
          T to use the template values
          M to update the values manually
          F to read the values from file
          S to solve the problem described by present values
          te to run a test of eliipticity influence
          tt to run a test of toroidness influence""")
    s=input()
    if s=='Q':return
    if s=='T':T()
    if s=='M':M()
    if s=='F':F()
    if s=='S':S()
    if s=='te':test_ellipticity()
    if s=='tt':test_toroidness()
    return run()

values={}
T()
run()
```

Вміст файлу `nep.py`:

```
import numpy as np
from grqi import GRQI, sparse_GRQI
import matplotlib.pyplot as plt
import matplotlib.cm as mcm
import matplotlib.colors as col
import matplotlib

class Nonlinear_EigenProblem:
    def __init__(self, basis, matrix, tolerance, maxIteration,
GRQI_parameters=None):
        self.matrix = matrix
        if not matrix.self_conj: raise AssertionError('Not ready for non-
Hermitian problems')
        self.cmplx = matrix.get_cmplx_indicator()
        self.tolerance = tolerance
        self.basis=basis
        self.GRQI_parameters = GRQI_parameters
        self.maxIteration = maxIteration
        self.sparse = matrix.get_sparse_indicator()
        self.cmplx = matrix.get_cmplx_indicator()
        self.prepare_matrix_A()
        self.solutions = []
```

```
    def prepare_matrix_A(self):
        self.A = self.matrix.build_A()
    def build_matrices(self, Lambda):
        self.B = self.matrix.build_B(Lambda)
        self.modB = self.matrix.build_dB(Lambda)
        self.modA = self.A - self.B
    def build_solution(self, Lambda0):
        Lambda = Lambda0
        for i in range(self.maxIteration):
            self.build_matrices(Lambda)
            dLambda, eivector = self.GRQI_linsolver()
            Lambda += dLambda
            if abs(dLambda) <= self.tolerance:break
        else:print(f'!!! {self.maxIteration} steps, iterations did not
converge\nLambda0 = {Lambda0}, last step = {dLambda}, Lambda = {Lambda}')
        print(f'final Lambda {Lambda}')
        eivector = self.normalize(eivector)
        self.solutions.append({'init': Lambda0, 'eivalue': Lambda,
'eivector': eivector})
    def GRQI_linsolver(self):
        if self.sparse:Lambda, eivector = sparse_GRQI(self.modA,
self.modB, x=None, mu0 = complex(0.,0.) if self.cmplx else 0., is_complex
= self.cmplx, **self.GRQI_parameters)
        else:Lambda, eivector = GRQI(self.modA, self.modB, x=None, mu0 =
complex(0.,0.) if self.cmplx else 0., is_complex = self.cmplx,
**self.GRQI_parameters)
        if self.cmplx: Lambda = Lambda.real
        return Lambda, eivector
    def make_solutions(self, Lambda_list):
        for Lambda0 in Lambda_list:self.build_solution(Lambda0)
    def normalize(self, vector):
        max_element = max(vector, key=abs)
        if self.cmplx:new_vector = vector *(np.conjugate(max_element)/
abs(max_element)**2)
        else:new_vector = vector / max_element
        return new_vector
    def show_radius (self, solution_index, theta, nSteps=40,save=False):
        try:
            eivector = self.solutions[solution_index]['eivector']
            eivalue = self.solutions[solution_index]['eivalue']
        except IndexError:
            print ("Wrong index")
            return
        variable0 = self.matrix.get_variable (0, eivector)
        variable1 = self.matrix.get_variable (1, eivector)
        #name0 = self.variable_names[0]
        #name1 = self.variable_names[1]
        grid = self.basis.grid
        center = grid.get_center()
        ray_plus = grid.build_ray (theta, center, nSteps)
        ray_minus = np.flip (grid.build_ray (theta+np.pi, center,
nSteps), axis=0)[:-1,:]
        profile0 = np.empty (nSteps)
        profile1 = np.empty (nSteps)
```

```
        rho_plus = np.sqrt (ray_plus[:,0]**2 + ray_plus[:,1]**2)
        rho_minus = -np.sqrt (ray_minus[:,0]**2 + ray_minus[:,1]**2)
        ray = np.concatenate ((ray_minus, ray_plus))
        rho = np.concatenate ((rho_minus, rho_plus))
        profile0 = np.array ([self.basis.get_value (ray[i,0], ray[i,1],
variable0) for i in range(len(rho))])
        profile1 = np.array ([self.basis.get_value (ray[i,0], ray[i,1],
variable1) for i in range(len(rho))])
        ct = np.cos (theta)
        st = np.sin (theta)
        Er = (profile0 * ct + profile1 * st) * np.sign(rho)
        Eb = (-profile0 * st + profile1 * ct) * np.sign(rho)
        kx_Ez = np.array ([self.basis.get_value (ray[i,0], ray[i,1],
variable1, 'x') for i in range(len(rho))]) * (-1j)
        kz_Ex = np.array ([self.basis.get_value (ray[i,0], ray[i,1],
variable0, 'y') for i in range(len(rho))]) * (-1j)
        Bpar = kx_Ez - kz_Ex
        fig, ax = plt.subplots (3, sharex=True, figsize=(8,6))
        fig.subplots_adjust(left=0.1, right=0.95, top=0.95, hspace=0)
        fig.suptitle ('omega={:8.5f}, Lambda={:8.5f}'.format (
            np.sqrt(eivalue), eivalue))
        if self.cmplx:
            ax[0].plot (rho, np.real(profile0), color='blue', marker='.',
label='Re ')#+name0)
            ax[0].plot (rho, np.imag(profile0), color='blue', ls=':',
marker='.', label='Im ')#+name0)
            ax[0].plot (rho, np.real(profile1), color='red', marker='.',
label='Re ')#+name1)
            ax[0].plot (rho, np.imag(profile1), color='red', ls=':',
marker='.', label='Im ')#+name1)
            ax[1].plot (rho, np.real(Er), color='blue', label='Re Er',
marker='.')
            ax[1].plot (rho, np.imag(Er), color='blue', ls=':', label='Im
Er', marker='.')
            ax[1].plot (rho, np.real(Eb), color='red', label='Re Eb',
marker='.')
            ax[1].plot (rho, np.imag(Eb), color='red', ls=':', label='Im
Eb', marker='.')
            ax[2].plot (rho, np.real(Bpar), color='green', label='Re
Bpar', marker='.')
            ax[2].plot (rho, np.imag(Bpar), color='green', ls=':',
label='Im Bpar', marker='.')
        else:
            ax[0].plot (rho, profile0, color='blue', label='name0',
marker='.')
            ax[0].plot (rho, profile1, color='red', label='name1',
marker='.')
            ax[1].plot (rho, Er, color='blue', label='Er', marker='.')
            ax[1].plot (rho, Eb, color='red', label='Eb', marker='.')
            ax[2].plot (rho, Bpar, color='green', label='Re Bpar',
marker='.')
        ax[0].legend()
        ax[1].legend()
        ax[2].legend()
        if save:
```

```
            plt.savefig(f'Lambda={self.solutions[0]['init']:.3f}/Radial
profile, Ellipticity={self.matrix.coeff.device.ellipticity}.png',dpi=300)
        else:
            plt.show()
        matplotlib.pyplot.close()

    def show_solution (self, var_no=0, index=None, value=None,
cmap='plasma',save=False):
        if index is None:
            if value is None:raise AssertionError ('index and value are
both None')
            solution, index = min (enumerate(self.solutions), key =
lambda sol: abs(sol[1]['eivalue'] - value))
            print ('Drawing eigenfunction No {}'.format(index))
        else:solution = self.solutions[index]
        variable = self.matrix.get_variable (var_no,
solution['eivector'])
        eivalue = solution['eivalue']
        omega = np.sqrt (eivalue)
        title = 'omega={:8.5f}, Lambda={:8.5f}'.format(omega, eivalue)
        triang = self.basis.grid.refine (1)
        z = np.array ([self.basis.get_value (*point, variable) for point
in zip (triang.x, triang.y)])
        if True:
            fig, ax = plt.subplots(1, 2, figsize=(11,6))
            plt.subplots_adjust(left=0.07, right=0.88, wspace=0.4)
            ax[0].set_aspect('equal')
            ax[1].set_aspect('equal')
            colorbox = [plt.axes ([0.43, 0.18, 0.03, 0.6]),
                        plt.axes ([0.9, 0.18, 0.03, 0.6])]
            colorbar = [fig.colorbar
                              (mcm.ScalarMappable (cmap=cmap,
                                   norm=self.build_color_norm (z.real)),
                               cax=colorbox[0]),
                        fig.colorbar
                              (mcm.ScalarMappable (cmap=cmap,
                                   norm=self.build_color_norm (z.imag)),
                               cax=colorbox[1])]
            ax[0].tricontourf (triang, z.real, cmap=cmap)
            ax[1].tricontourf (triang, z.imag, cmap=cmap)
            fig.suptitle(title)
            ax[0].set_title ('Re')
            ax[1].set_title ('Im')
        if save:
            plt.savefig(f'Lambda={self.solutions[0]['init']:.3f}/Solution
, Ellipticity={self.matrix.coeff.device.ellipticity}.png',dpi=300)
        else:
            plt.show()
        matplotlib.pyplot.close()

    def show_Bparallel (self, index=None, value=None,
cmap='plasma',save=False):
        if index is None:
            if value is None:
```

```
                raise AssertionError ('index and value are both None')
            solution, index = min (enumerate(self.solutions), key =
lambda sol: abs(sol[1]['eivalue'] - value))
        else:
            solution = self.solutions[index]
        Ex = self.matrix.get_variable (0, solution['eivector'])
        Ez = self.matrix.get_variable (1, solution['eivector'])
        eivalue = solution['eivalue']
        omega = np.sqrt (eivalue)
        title = 'omega={:8.5f}, Lambda={:8.5f}, {}'.format (omega,
eivalue, 'B_parallel')
        triang = self.basis.grid.get_cell_centers()
        kx_Ez = np.array ([self.basis.get_value (*point, Ez, 'x') for
point in zip (triang.x, triang.y)])
        kz_Ex = np.array ([self.basis.get_value (*point, Ex, 'y') for
point in zip (triang.x, triang.y)])
        device = self.matrix.coeff.device
        Bpar = (kz_Ex- kx_Ez) * complex (0, 1. / (omega * device.nu *
np.sqrt(device.S)))
        boundary = self.basis.grid.get_boundary()
        boundary = np.concatenate ((boundary, boundary[0:1,:]))
        ktrans2 = self.matrix.coeff.k_trans2 (triang.x, triang.y,
eivalue)
        if True:
            fig, ax = plt.subplots(1, 2, figsize=(11,6))
            plt.subplots_adjust(left=0.07, right=0.88, wspace=0.4)
            ax[0].set_aspect('equal')
            ax[1].set_aspect('equal')
            colorbox = [plt.axes ([0.43, 0.18, 0.03, 0.6]), plt.axes
([0.9, 0.18, 0.03, 0.6])]
            colorbar = [fig.colorbar (mcm.ScalarMappable (cmap=cmap,
norm=self.build_color_norm (Bpar.real)), cax=colorbox[0]),
                        fig.colorbar (mcm.ScalarMappable (cmap=cmap,
norm=self.build_color_norm (Bpar.imag)), cax=colorbox[1])]
            ax[0].tricontourf (triang, Bpar.real, cmap=cmap)
            ax[1].tricontourf (triang, Bpar.imag, cmap=cmap)
            ax[0].plot (boundary[:,0], boundary[:,1], color='black')
            ax[1].plot (boundary[:,0], boundary[:,1], color='black')
            fig.suptitle(title)
            ax[0].set_title ('Re')
            ax[1].set_title ('Im')
            if self.encloses0 (ktrans2):
                ax[0].tricontour (triang, ktrans2, levels=[0.0],
                                  colors='black')
                ax[1].tricontour (triang, ktrans2, levels=[0.0],
                                  colors='black')
        if save:
            plt.savefig(f'Lambda={self.solutions[0]['init']:.3f}/Bparalle
l, Ellipticity={self.matrix.coeff.device.ellipticity}.png',dpi=300)
        else:
            plt.show()
        matplotlib.pyplot.close()

    def run(self,save=False,lambda0=0):
```

```
if save:
    import os
    self.make_solutions([lambda0])
    try:os.mkdir(f'Lambda={self.solutions[0]['init']:.3f}')
    except FileExistsError: pass
    self.show_solution(index=0,save=True)
    self.show_radius(0,0,save=True)
    self.show_Bparallel(0,0,save=True)
    return

while True:
    print("""Enter your command:
  Q to exit
  S to make a solution
  P to plot a solution
  B to show Bparallel
  R to show the radial profile
  """)
    s=input()
    if s=='Q':return
    if s=='S':
        print('Enter the Lambda0:')
        try:Lambda0=float(input())
        except:
            print('Wrong input')
            continue
        self.make_solutions([Lambda0])
    if s=='P':
        print('Enter the index:')
        try:index=int(input())
        except:
            print('Wrong input')
            continue
        self.show_solution(index=index)
    if s=='B':
        print('Enter the index:')
        try:index=int(input())
        except:
            print('Wrong input')
            continue
        self.show_Bparallel (index=index)
    if s=='R':
        print('Enter the index:')
        try:index=int(input())
        except:
            print('Wrong input')
            continue
        print('Enter the angle:')
        try:theta=float(input())*np.pi/180.
        except:
            print('Wrong input')
            continue
```

```
                self.show_radius(index,theta)

    @staticmethod
    def encloses0 (array):
        return array.min()<0. and array.max()>0.

    @staticmethod
    def build_color_norm (array):
        return col.Normalize (vmin=array.min(), vmax=array.max())
```

Вміст файлу `matrix.py`:

```
import numpy as np
import scipy.sparse as sprs

class Matrix_Builder:
    def __init__ (self, coeff, basis, sparse=False,
self_conj=True,cmplx=True):
        self.basis = basis
        self.boundary_type = coeff.boundary_type
        self.coeff = coeff
        self.self_conj = self_conj
        self.sparse = sparse
        self.cmplx = cmplx
        if coeff.is_complex() and not cmplx:
            print ('Setting the "cmplx" flag to true because the
coefficients are complex.')
            self.cmplx = True
        self.nonlinear = coeff.is_nonlinear()
        self.nVar = self.coeff.get_number_of_variables()
        self.basis_size = self.basis.get_size()
        self.allocate_matrix_indices()
        if self.boundary_type == 'En0' or self.boundary_type ==
'Et0':self.prepare_boundary_coefficients()

    def prepare_boundary_coefficients (self):
        self.nxy = np.zeros ((self.basis_size, self.nVar))
        for i in range (self.basis_size):
            if self.basis.is_boundary(i): self.nxy[i,:] = self.coeff.nxy
(*self.basis.get_top_coordinates(i))

    def allocate_matrix_indices (self):
        self.component = np.empty ((self.basis_size, self.nVar), int)
        self.index = []
        for i in range(self.basis_size):
            if self.basis.is_boundary (i):
                if self.boundary_type == 'free':
                    for j in range(self.nVar):
                        self.component[i,j] = len(self.index)
                        self.index.append ([i, j])
                elif (self.boundary_type == 'En0' or self.boundary_type
== 'Et0'):
                    for j in range(self.nVar):self.component[i,j] =
len(self.index)
```

```
                    self.index.append ([i, -1])
                elif self.boundary_type == 'fixed':raise AssertionError
('Internal error: boundary node with fixed-boundary condition')
                else:raise AssertionError ('Unknown boundary condition
"{}"'.format(self.boundary_type))
            else:
                for j in range(self.nVar):
                    self.component[i,j] = len(self.index)
                    self.index.append ([i, j])
        self.index = np.array (self.index)
        self.dim = self.index.shape[0]
    def get_index (self):
        return self.index
    def get_component (self):
        return self.component
    def get_size (self):
        return self.dim
    def get_sparse_indicator (self):
        return self.sparse
    def get_cmplx_indicator (self):
        return self.cmplx
    def build_A (self):
        if self.sparse:
            print (' ')
            print ('Building sparse matrix A')
            self.A = self.build_sparse_matrix (self.A_element)
        else:
            print ('Building dense matrix A')
            self.A = self.build_dense_matrix (self.A_element)
            print ('Matrix A has {} nonzero elements'.format(self.A.nnz))
        return self.A
    def build_B (self, Lambda=0.):
        if self.sparse:
            print (' ')
            print ('Building sparse matrix  B')
            self.B = self.build_sparse_matrix (self.B_element, Lambda)
        else:
            print (' ')
            print ('Building dense matrix  B')
            self.B = self.build_dense_matrix (self.B_element, Lambda)
            print ('Matrix A has {} nonzero elements'.format(self.B.nnz))
        return self.B
    def build_dB (self, Lambda):
        if self.sparse:
            print ('Building sparse matrix dB')
            self.dB = self.build_sparse_matrix (self.dB_element, Lambda)
        else:
            print ('Building dense matrix dB')
            self.dB = self.build_dense_matrix (self.dB_element, Lambda)
        return self.dB
    def build_dense_matrix (self, element_getter, Lambda=0.):
        if self.cmplx:result = np.zeros ((self.dim, self.dim),
dtype=np.complex64)
```

```
        else:result = np.zeros ((self.dim, self.dim))
        for i in range(self.dim):
            for j in range(i, self.dim):
                fi = self.index[i,0]
                fj = self.index[j,0]
                if self.basis.disjoint (fi, fj):continue
                vi = self.index[i,1]
                vj = self.index[j,1]
                if self.cmplx and not self.coeff.is_complex():
                    result[i,j] = complex (self.aggregate(element_getter,
fi, vi, fj, vj, Lambda), 0.)
                else:
                    result[i,j] = self.aggregate (element_getter, fi,
vi,fj, vj, Lambda)
                if i==j: continue
                if self.self_conj:
                    if self.cmplx:result[j,i] = np.conjugate
(result[i,j])
                    else:result[j,i] = result[i,j]
                else:
                    if self.cmplx and not
self.coeff.is_complex():result[j,i] = complex
(self.aggregate(element_getter, fj, vj, fi, vi, Lambda), 0.)
                    else:result[j,i] = self.aggregate (element_getter,
fj, vj,fi, vi, Lambda)
        return result
    def build_sparse_matrix (self, element_getter, Lambda=0.):
        A = self.build_dense_matrix (element_getter, Lambda)
        i_list = []
        j_list = []
        value_list = []
        for i in range(self.dim):
            for j in range(i, self.dim):
                fi = self.index[i,0]
                fj = self.index[j,0]
                if self.basis.disjoint (fi, fj):continue
                i_list.append (i)
                j_list.append (j)
                vi = self.index[i,1]
                vj = self.index[j,1]
                if self.cmplx and not
self.coeff.is_complex():value_list.append (complex
(self.aggregate(element_getter, fi, vi, fj, vj, Lambda), 0.))
                else:value_list.append (self.aggregate (element_getter,
fi, vi,fj, vj, Lambda))
                if i==j: continue
                i_list.append (j)
                j_list.append (i)
                if self.self_conj:
                    if self.cmplx:value_list.append (np.conjugate
(value_list[-1]))
                    else:value_list.append (value_list[-1])
                else:
```

```
                    if self.cmplx and not
self.coeff.is_complex():value_list.append (complex
(self.aggregate(element_getter, fj, vj, fi, vi, Lambda), 0.))
                    else:value_list.append (self.aggregate
(element_getter, fj, vj, fi, vi, Lambda))
        return sprs.csr_matrix ((value_list,
(i_list,j_list)),shape=(self.dim,self.dim))
    def aggregate (self, element_getter, fi, vi, fj, vj, Lambda):
        if vi == -1:
            if vj == -1:return sum ([sum ([element_getter (fi, i, fj, j,
Lambda) * self.nxy[fi,i] for i in range(self.nVar)]) * self.nxy[fj,j] for
j in range(self.nVar)])
            else:return sum ([element_getter (fi, i, fj, vj, Lambda) *
self.nxy[fi,i] for i in range(self.nVar)])
        else:
            if vj == -1:return sum ([element_getter (fi, vi, fj, j,
Lambda) * self.nxy[fj,j] for j in range(self.nVar)])
            else:return element_getter (fi, vi, fj, vj, Lambda)
    def A_element (self, fi, vi, fj, vj, Lambda):
        a = self.coeff.c[vi][vj]
        if a is None:result = 0.
        else:result = self.basis.matrix_element(a, fi, fj)
        a = self.coeff.axx[vi][vj]
        if a is not None:result += self.basis.matrix_element(a, fi, fj,
'x', 'x')
        a = self.coeff.axy[vi][vj]
        if a is not None:result += self.basis.matrix_element(a, fi, fj,
'x', 'y')
        a = self.coeff.ayx[vi][vj]
        if a is not None:result += self.basis.matrix_element(a, fi, fj,
'y', 'x')
        a = self.coeff.ayy[vi][vj]
        if a is not None:result += self.basis.matrix_element(a, fi, fj,
'y', 'y')
        a = self.coeff.px[vi][vj]
        if a is not None:result += self.basis.matrix_element(a, fi, fj,
'no', 'x')
        a = self.coeff.qx[vi][vj]
        if a is not None:result += self.basis.matrix_element(a, fi, fj,
'x', 'no')
        a = self.coeff.py[vi][vj]
        if a is not None:result += self.basis.matrix_element(a, fi, fj,
'no', 'y')
        a = self.coeff.qy[vi][vj]
        if a is not None:result += self.basis.matrix_element(a, fi, fj,
'y', 'no')
        return result
    def B_element (self, fi, vi, fj, vj, Lambda):
        b = self.coeff.b[vi][vj]
        if b is None:result = 0.
        else:
            if self.nonlinear:result = self.basis.matrix_element (lambda
x,y: b(x,y,Lambda),fi, fj)
            else:result = self.basis.matrix_element (b, fi, fj)
        return result
```

```
    def dB_element (self, fi, vi, fj, vj, Lambda):
        db = self.coeff.db[vi][vj]
        if db is None:result = 0.
        else:result = self.basis.matrix_element (lambda x,y:
db(x,y,Lambda),fi, fj)
        return result
    def dump_matrix (self, matrix, file, to_real=False):
        if self.sparse:array = matrix.toarray()
        else:array = matrix
        m, n = array.shape
        file.write('     ')
        for j in range(n):file.write ('{:12d} '.format(j))
        file.write('\n')
        for i in range(m):
            file.write('{:4d} '.format(i))
            for j in range(n):
                if self.cmplx and to_real:file.write ('{:12.3g}
'.format(array[i,j].real))
                else:file.write ('{:12.3g} '.format(array[i,j]))
            file.write('\n')
    def dump_matrices (self, file_name, to_real=False, dump_dB=False):
        with open (file_name, 'w') as file:
            file.write (' A matrix\n')
            self.dump_matrix (self.A, file, to_real=to_real)
            file.write ('\n B matrix\n')
            self.dump_matrix (self.B, file, to_real=to_real)
            if dump_dB:
                file.write ('\n dB matrix\n')
                self.dump_matrix (self.dB, file, to_real=to_real)
    def get_variable (self, var_no, vector):
        if self.cmplx:result = np.empty (self.basis_size, dtype=complex)
        else:result = np.empty (self.basis_size)
        for i in range(self.basis_size):
            comp = self.component[i, var_no]
            if self.index[comp,1] == -1:result[i] = vector[comp] *
self.nxy[i,var_no]
            else:result[i] = vector[comp]
        return result
    def make_vector (self, f):
        vector = np.empty (self.dim)
        for i in range (self.dim):
            ip = self.index[i,0]
            x, z = self.basis.grid.points [self.basis.index[ip],:]
            if self.index[i,1] == -1:vector[i] = (sum ([f[k](x,z) *
self.nxy[ip,k] for k in range(self.nVar)])/ sum (self.nxy[ip,:]**2))
            else:vector[i] = f[self.index[i,1]](x,z)
        return vector
    def get_component_coordinates (self):
        x = np.empty (self.dim)
        z = np.empty (self.dim)
        for i in range(self.dim):
            ip = self.index[i,0]
            x[i], z[i] = self.basis.grid.points [self.basis.index[ip],:]
```

```
        return x, z
```

Вміст файлу `iota.py`:

```
import numpy as np

class Parabolic_q_iota:
    def __init__ (self, iota0, delta):
        self.iota0 = iota0
        self.delta = delta
    def __call__ (self, s, order=0):
        if order==0:return self.iota0 / (1. + self.delta * s)
        elif order==1:return - self.iota0 * self.delta / (1. + self.delta
* s) ** 2
        else:raise AssertionError ('iota derivative of order {}
undefined'.format(order))
class Constant_Density_Profile:
    def __init__ (self):
        print(1)
    def rho (self, x, z):
        if type (x) is float:return 1.
        else:return np.ones_like (x, float)

class Quasiparabolic_Density_Profile:
    def __init__  (self, gamma, pedestal, device, logger=None):
        self.a1 = device.a1
        self.a2 = device.a2
        self.gamma = gamma
        self.pedestal = pedestal
        self.rho_change = 1. - pedestal
        self.alpha = 1. - pedestal ** (1./gamma)
        self.C1 = -2. * self.alpha * self.gamma
        self.C2 = 4. * self.alpha * self.gamma * (self.gamma-1)
    def rho (self, x, z):
        return ((1. - self.alpha * ((x/self.a1)**2 + (z/self.a2)**2))
                ** self.gamma)
```

Вміст файлу `grqi.py`:

```
from scipy.linalg import solve, lu_factor, lu_solve, LinAlgError
from scipy.sparse.linalg import spsolve, factorized, MatrixRankWarning
import warnings
import numpy as np

def inv_iter (mu0, A, B=None, x=None, is_complex=False,
              nIterations=20, tolerance=None, factorize=False,
              seed=None):
    dim = A.shape[0]
    assert A.shape == (dim, dim), 'IVI: wrong shape of A matrix'
    if B is not None:assert B.shape == (dim, dim), 'IVI: wrong shape of B
matrix'
    if x is None:
        rng = np.random.default_rng (seed)
```

```
        if is_complex:x = (np.sqrt (rng.uniform (0., 1., dim)) * np.exp
(2j * rng.uniform (0., np.pi, dim)))
        else:x = rng.uniform (-1., 1., dim)
    else:x /= np.max (np.abs(x))
    if B is None:C = A - mu0 * np.eye (dim)
    else:C = A - mu0 * B
    if factorize:
        lu_and_piv = lu_factor (C)
        solver = lambda x: lu_solve (lu_and_piv, x, overwrite_b=True)
    else:solver = lambda x: solve (C, x)
    solving_count = 0
    for iteration in range(nIterations):
        if B is not None:y = np.dot (B, x)
        y = solver (y)
        peak_place = np.argmax (np.abs(y))
        y /= y[peak_place]
        solving_count += 1
        if tolerance is not None:
            try:x /= x[peak_place]
            except (ZeroDivisionError, OverflowError):
                x = y
                continue
            error = np.max (np.abs (y-x))
            if error < tolerance:break
        x = y
    mu = np.vdot (y, np.dot(A,y))
    if B is not None:mu /= np.vdot (y, np.dot(B,y))
    return mu, y, solving_count
def GRQI (A, B=None, x=None, mu0=None, is_complex=False, nIterations=20,
          tolerance=None, II_iter=1, II_tol=None, factorize=False,
          seed=None):
    dim = A.shape[0]
    assert A.shape == (dim, dim), 'RQI: wrong shape of A matrix'
    if B is not None:assert B.shape == (dim, dim), 'RQI: wrong shape of B
matrix'
    if x is None:
        rng = np.random.default_rng (seed)
        if is_complex:x = (np.sqrt (rng.uniform (0., 1., dim)) * np.exp
(2j * rng.uniform (0., np.pi, dim)))
        else:x = rng.uniform (-1., 1., dim)
    else: x /= np.max (np.abs(x))
    if mu0 is None:
        mu0 = np.vdot (x, np.dot(A,x))
        if B is not None: mu0 /= np.vdot (x, np.dot(B,x))
    solve_count = 0
    for iteration in range(nIterations):
        if type(II_iter) is int:II_iter_limit = II_iter
        else:
            if iteration >= len(II_iter):II_iter_limit = II_iter[-1]
            else:II_iter_limit = II_iter[iteration]
        try:
            mu, x, II_iterations = inv_iter (
                mu0, A, B=B, x=x, is_complex=is_complex,
```

```
                nIterations=II_iter_limit, tolerance=II_tol,
                factorize=factorize)
            except LinAlgError:break
            solve_count += II_iterations
            if tolerance is not None:
                change = np.abs (mu - mu0)
                if change < tolerance:break
            mu0 = mu
        return mu, x
    def sparse_inv_iter (mu0, A, B=None, x=None, is_complex=False,
                         nIterations=20, tolerance=None, factorize=False,
                         seed=None):
        warnings.simplefilter ('error', MatrixRankWarning)
        dim = A.shape[0]
        assert A.shape == (dim, dim), 'IVI: wrong shape of A matrix'
        if B is not None:assert B.shape == (dim, dim), 'IVI: wrong shape of B
    matrix'
        if x is None:
            rng = np.random.default_rng (seed)
            if is_complex:x = (np.sqrt (rng.uniform (0., 1., dim)) * np.exp
    (2j * rng.uniform (0., np.pi, dim)))
            else:x = rng.uniform (-1., 1., dim)
        else: x /= np.max (np.abs(x))
        if B is None:C = A - mu0 * np.eye (dim)
        else:C = A - mu0 * B
        if factorize:solver = factorized (C)
        else:solver = lambda x: spsolve (C, x)
        solving_count = 0
        for iteration in range(nIterations):
            if B is not None:y = B.dot(x)
            y = solver (y)
            peak_place = np.argmax (np.abs(y))
            y /= y[peak_place]
            solving_count += 1
            if tolerance is not None:
                try:x /= x[peak_place]
                except (ZeroDivisionError, OverflowError):
                    x = y
                    continue
                error = np.max (np.abs (y-x))
                if error < tolerance:break
            x = y
        mu = np.vdot (y, A.dot(y))
        if B is not None:mu /= np.vdot (y, B.dot(y))
        return mu, y, solving_count
    def sparse_GRQI (A, B=None, x=None, mu0=None, is_complex=False,
                     nIterations=20, tolerance=None, II_iter=1, II_tol=None,
                     factorize=False, seed=None):
        dim = A.shape[0]
        assert A.shape == (dim, dim), 'RQI: wrong shape of A matrix'
        if B is not None:assert B.shape == (dim, dim), 'RQI: wrong shape of B
    matrix'
        if x is None:
```

```
        rng = np.random.default_rng (seed)
        if is_complex:x = (np.sqrt (rng.uniform (0., 1., dim)) * np.exp
(2j * rng.uniform (0., np.pi, dim)))
        else:x = rng.uniform (-1., 1., dim)
    else:x /= np.max (np.abs(x))
    if mu0 is None:
        mu0 = np.vdot (x, A.dot(x))
        if B is not None:mu0 /= np.vdot (x, B.dot(x))
    solve_count = 0
    for iteration in range(nIterations):
        if type(II_iter) is int:II_iter_limit = II_iter
        else:
            if iteration >= len(II_iter):II_iter_limit = II_iter[-1]
            else:II_iter_limit = II_iter[iteration]
        try:
            mu, x, II_iterations = sparse_inv_iter (
                mu0, A, B=B, x=x, is_complex=is_complex,
                nIterations=II_iter_limit, tolerance=II_tol,
                factorize=factorize)
        except (RuntimeError, MatrixRankWarning):break
        solve_count += II_iterations
        if tolerance is not None:
            change = np.abs (mu - mu0)
            if change < tolerance:break
        mu0 = mu
    return mu, x
```

Вміст файлу `grid.py`:

```
import numpy as np
import matplotlib.tri as tri
import matplotlib.pyplot as plt

class Triangular_Grid_in_Ellipse:
    def __init__ (self, a, b, step, spread=1., theta1=0., theta_shift=1.,
                  center=(0,0)):
        self.a = a
        self.b = b
        self.step = step
        self.spread = spread
        self.center = np.array (center, dtype=float)
        self.theta1 = theta1
        self.theta_shift = theta_shift
        rmax = np.sqrt (a*b)
        self.nRadii = int(round (rmax / step)) + 1
        self.rad_list = np.linspace (0, rmax, self.nRadii)
        x = [0.]
        z = [0.]
        self.boundary_node = [False]
        self.nBoundary = 0
        self.nInternal = 1
        self.boundary_index = []
        theta_start = theta1
```

```
        for j, r in enumerate(self.rad_list[1:]):
            nPoints = int(round (2.*np.pi*r / (spread * step)))
            theta_step = 2.*np.pi / nPoints
            for i in range(nPoints):
                theta = theta_start + theta_step * i
                x.append (r * np.cos(theta))
                z.append (r * np.sin(theta))
                if j == self.nRadii-2:  # Boundary circle
                    self.boundary_node.append (True)
                    self.nBoundary += 1
                    self.boundary_index.append (len(x)-1)
                else:                   # Internal circle
                    self.boundary_node.append (False)
                    self.nInternal += 1
            theta_start = theta_start + theta_shift
        self.points = np.array([x, z]).transpose()
        self.boundary_node = np.array(self.boundary_node)
        elongate = np.sqrt (b/a)
        self.points[:,0] /= elongate
        self.points[:,1] *= elongate
        self.normal, self.tangent = self.get_orths (
            self.points[self.boundary_index,0],
            self.points[self.boundary_index,1])
        self.points += center
        self.delaunay = tri.Triangulation (self.points[:,0],
                                           self.points[:,1])
        self.trifinder = self.delaunay.get_trifinder()
        self.cells = self.delaunay.triangles
        self.cell_neighbors = self.delaunay.neighbors
        self.nNodes = self.points.shape[0]
        self.nCells = self.cells.shape[0]
        self.build_edges()
        self.arrange_node_neighbors()
        print ('node neighbors ready')
        self.cell_square = np.empty (self.nCells)
        for i in range (self.nCells):
            self.cell_square[i] = self.square (i)
        self.cell_centers = tri.Triangulation (
            np.array ([sum (self.points[self.cells[cell_no,:],0]) / 3.
                       for cell_no in range(self.nCells)]),
            np.array ([sum (self.points[self.cells[cell_no,:],1]) / 3.
                       for cell_no in range(self.nCells)]))
    def get_center (self):
        return self.center
    def get_boundary (self):
        return self.points [self.boundary_index, :]
    def get_boundary_index (self):
        return self.boundary_index
    def get_orths (self, x, z):
        xa = x / self.a**2
        zb = z / self.b**2
        norm = np.sqrt (xa**2 + zb**2)
```

```
            xa /= norm
            zb /= norm
            normal = np.empty ((len(x), 2))
            tangent = np.empty ((len(x), 2))
            normal[:,0] = xa
            normal[:,1] = zb
            tangent[:,0] = -zb
            tangent[:,1] = xa
            return normal, tangent
        def get_boundary_orths (self):
            return self.normal, self.tangent
        def build_edges (self):
            self.edges = []
            self.boundary_edge = []
            self.edge_adjacents = []
            for cell in range(self.nCells):
                for node in range(3):
                    neighbor = self.cell_neighbors[cell,node]
                    if neighbor == -1: # Boundary edge
                        node1 = (node+1) % 3
                        self.edges.append ([self.cells[cell,node],
                                            self.cells[cell,node1]])
                        self.boundary_edge.append (True)
                        self.edge_adjacents.append ([cell,-1])
                    elif neighbor > cell: # New internal edge
                        node1 = (node+1) % 3
                        self.edges.append ([self.cells[cell,node],
                                            self.cells[cell,node1]])
                        self.boundary_edge.append (False)
                        self.edge_adjacents.append ([cell, neighbor])
            self.edges = np.array (self.edges)
            self.boundary_edge = np.array (self.boundary_edge)
            self.edge_adjacents = np.array (self.edge_adjacents)
            self.nEdges = self.edges.shape[0]
        def show_grid (self, annotate_nodes=False,
    annotate_cells=False,annotate_edges=False):
            fig, ax = plt.subplots()
            ax.set_aspect ('equal')
            for i in range(self.nEdges):
                if self.boundary_edge[i]:ax.plot
    (self.points[self.edges[i,:],0],self.points[self.edges[i,:],1],
    color='black')
                else:ax.plot
    (self.points[self.edges[i,:],0],self.points[self.edges[i,:],1],
    color='green')
            ax.plot ([self.points[i,0] for i in range(self.nNodes)
                      if not self.boundary_node[i]],
                     [self.points[i,1] for i in range(self.nNodes)
                      if not self.boundary_node[i]],
                     ls='', marker = '.', color='blue')
            ax.plot ([self.points[i,0] for i in range(self.nNodes)
                      if self.boundary_node[i]],
                     [self.points[i,1] for i in range(self.nNodes)
```

```
                    if self.boundary_node[i]],
                   ls='', marker = '.', color='red')
        if annotate_nodes:
            for i in range(self.nNodes):ax.annotate (str(i),
self.points[i,:], color='magenta')
        if annotate_cells:
            for i in range(self.nCells):ax.annotate (str(i),
self.get_cell_middle(i), color='brown')
        if annotate_edges:
            for i in range(self.nEdges):ax.annotate (str(i),
self.get_edge_middle(i), color='orange')
        plt.show()
    def arrange_node_neighbors (self):
        self.node_neighbors = self.nNodes * [set()]
        for c in range(self.nCells):
            for n in self.cells[c,:]:self.node_neighbors[n] =
self.node_neighbors[n] | {c,}
    def get_cell_middle (self, cell):
        return np.array([sum ([self.points[n,0] for n in
self.cells[cell,:]]) / 3.,
                         sum ([self.points[n,1] for n in
self.cells[cell,:]]) / 3.])
    def get_edge_middle (self, edge):
        return np.array([sum ([self.points[n,0] for n in
self.edges[edge,:]]) / 2.,
                         sum ([self.points[n,1] for n in
self.edges[edge,:]]) / 2.])
    def square (self, cell):
        x1, z1 = self.points[self.cells[cell,0],:]
        x2, z2 = self.points[self.cells[cell,1],:]
        x3, z3 = self.points[self.cells[cell,2],:]
        return ((z2-z1) * (x2-x3) - (z2-z3) * (x2-x1)) * 0.5
    def point_in_cell (self, point, cell_index):
        x1, z1 = self.points[self.cells[cell_index,0],:]
        x2, z2 = self.points[self.cells[cell_index,1],:]
        x3, z3 = self.points[self.cells[cell_index,2],:]
        return (((x3-x2) * (point[1]-z2) - (z3-z2) * (point[0]-x2) >= 0.)
                and ((x2-x1) * (point[1]-z1) - (z2-z1) * (point[0]-x1) >=
0.)
                and ((x1-x3) * (point[1]-z3) - (z1-z3) * (point[0]-x3) >=
0.))
    def get_cell_coords (self, cell_index):
        coords = self.points[self.cells[cell_index,:],:]
        return coords[:,0], coords[:,1]
    def get_cell_centers (self):
        return self.cell_centers
    def refine (self, density):
        if density < 1:
            density = 1
            print ('Triangular_Grid :: refine warning: density<1; set to
1')
        if density == 1:return self.delaunay
        new_points = self.points
        additional = []
```

```
        for e in range(self.nEdges):
            edge = self.edges[e,:]
            point1 = self.points[edge[0],:]
            point2 = self.points[edge[1],:]
            for i in range(1, density):additional.append (point1 *
(i/density) + point2 * (1.-i/density))
        if density > 2:
            for c in range(self.nCells):
                cell = self.cells[c,:]
                point0 = self.points[cell[0],:]
                point1 = self.points[cell[1],:]
                point2 = self.points[cell[2],:]
                for i0 in range(1, density-1):
                    for i1 in range (1, density-i0):additional.append
(point0 * (i0/density)+ point1 * (i1/density) + point2 * (1.-
(i0+i1)/density))
        additional = np.array(additional)
        new_points = np.concatenate ((new_points, additional))
        triang = tri.Triangulation (new_points[:,0], new_points[:,1])
        return triang

    @staticmethod
    def show_nodes (triang):
        fig, ax = plt.subplots()
        ax.set_aspect ('equal')
        ax.plot (triang.x, triang.y,ls='', marker = '.', color='blue')
        plt.show()
    def build_ray (self, angle, starting_point=[0.,0.], nPoints=2):
        accuracy = max (0.01, 1./nPoints)
        x0, z0 = starting_point
        start_cell = self.trifinder (x0, z0)
        if start_cell==-1:return None
        a2 = self.a**2
        b2 = self.b**2
        c = np.cos(angle)
        s = np.sin(angle)
        D = c**2 / a2 + s**2 / b2 - (x0*s - z0*c) ** 2 / (a2*b2)
        D = np.sqrt(D)
        if D < 0.0:
            print ("WARNING: ray does not cross the ellipse (D =
{});".format(D))
            print ("starting point ({:8.5f}, {:8.5f}) found in cell
{}".format(x0, z0, start_cell))
            print ("ray azimuth is {:7.4f} = {9.4f}
degrees".format(angle, angle * 180. / np.pi))
            return None
        l = -x0 * c / a2 - z0 * s / b2
        length1 = l - D
        length2 = l + D
        if length2 <= 0. or length1 > 0.:
            print ("WARNING: ray does not cross the ellipse (lengths =
{{:8.4f}}, {{:8.4f}});".format(length1,length2))
            print ("starting point ({:8.5f}, {:8.5f}) found in cell
{}".format(x0, z0, start_cell))
```

```
            print ("ray azimuth is {:7.4f} = {9.4f}
degrees".format(angle, angle * 180. / np.pi))
            return None
        step = accuracy * length2
        length = length2
        while (self.trifinder (x0 + length * c,z0 + length * s) == -
1):length -= step
        mesh = np.linspace (0., length, nPoints)
        ray = np.empty ((nPoints, 2))
        ray[:,0] = x0 + mesh * c
        ray[:,1] = z0 + mesh * s
        return ray
```

Вміст файлу `gauss.py`:

```
import numpy as np

class Gauss2D:
    def __init__ (self, n):
        if (n<1 or n>8):raise AssertionError ("n={} is not supported in
Gauss2D".format(n))
        if n==1:
            s = 1./3.
            self.coord = np.array([s,s,s])
            self.weight = np.array([1.])
        elif n==2:
            w = 1./3.
            s1 = 1. / 6.
            s2 = 2. / 3.
            self.coord = np.array([[s2,s1,s1],[s1,s2,s1],[s1,s1,s2]])
            self.weight = np.array([w,w,w])
        elif n==3 or n==4:
            w1 = 0.223381589678011
            s11 = 0.108103018168070
            s12 = 0.445948490915965
            w2 = 0.109951743655322
            s21 = 0.816847572980459
            s22 = 0.091576213509771
            self.coord = np.array([[s11,s12,s12], [s12,s11,s12],
                                   [s12,s12,s11], [s21,s22,s22],
                                   [s22,s21,s22], [s22,s22,s21]])
            self.weight = np.array([w1,w1,w1,w2,w2,w2])
        elif n==5:
            s0 = 1./3.
            w1 = 0.132394152788506
            s11 = 0.059715871789770
            s12 = 0.470142064105115
            w2 = 0.125939180544827
            s21 = 0.797426985353087
            s22 = 0.101286507323456
            self.coord = np.array([[s0,s0,s0], [s11,s12,s12],
[s12,s11,s12],
                                   [s12,s12,s11], [s21,s22,s22],
                                   [s22,s21,s22], [s22,s22,s21]])
```

```
            self.weight = np.array([9./40, w1, w1, w1, w2, w2, w2])
        elif n==6:
            w1 = 0.116786275726379
            s11 = 0.501426509658179
            s12 = 0.249286745170910
            w2 = 0.050844906370207
            s21 = 0.873821971016996
            s22 = 0.063089014491502
            w3 = 0.082851075618374
            s31 = 0.053145049844817
            s32 = 0.310352451033784
            s33 = 0.636502499121399
            self.coord = np.array([[s11,s12,s12], [s12,s11,s12],
                                   [s12,s12,s11], [s21,s22,s22],
                                   [s22,s21,s22], [s22,s22,s21],
                                   [s31,s32,s33], [s33,s31,s32],
                                   [s32,s33,s31], [s31,s33,s32],
                                   [s32,s31,s33], [s33,s32,s31]])
            self.weight = np.array([w1, w1, w1, w2, w2, w2,
                                    w3, w3, w3, w3, w3, w3])
        else: #n==8 or n==7
            w1 = 0.144315607677787
            s1 = 1./3.
            w2 = 0.095091634267285
            s21 = 0.081414823414554
            s22 = 0.459292588292723
            w3 = 0.103217370534718
            s31 = 0.658861384496480
            s32 = 0.170569307751760
            w4 = 0.032458497623198
            s41 = 0.898905543365938
            s42 = 0.050547228317031
            w5 = 0.027230314174435
            s51 = 0.008394777409958
            s52 = 0.263112829634638
            s53 = 0.728492392955404
            self.coord = np.array([[s1,s1,s1],    [s21,s22,s22],
                                   [s22,s21,s22], [s22,s22,s21],
                                   [s31,s32,s32], [s32,s31,s32],
                                   [s32,s32,s31], [s41,s42,s42],
                                   [s42,s41,s42], [s42,s42,s41],
                                   [s51,s52,s53], [s53,s51,s52],
                                   [s52,s53,s51], [s51,s53,s52],
                                   [s52,s51,s53], [s53,s52,s51]])
            self.weight = np.array([w1, w2, w2, w2,
                                    w3, w3, w3, w4, w4, w4,
                                    w5, w5, w5, w5, w5, w5])
    def __call__ (self, f, vertex_x, vertex_y, square):
        x = np.dot (self.coord, vertex_x)
        y = np.dot (self.coord, vertex_y)
        return np.dot (self.weight, f(x,y)) * square
```

Вміст файлу `device.py`:

```
import numpy as np

class Constants:
    c = 2.9979e10
    Mp = 1.6726e-24    # Proton mass
    Me = 9.1094e-28    # Proton charge
    e = 4.8032e-10     # electron charge
    h = 6.6261e-27     # Planck
    eV = 1.6022e-12

class Device:
    def __init__ (self, name, B0, a, R0, n0, ellipticity=1., mu=2, Z=1):
        const = Constants()
        self.name = name
        self.B0 = B0 * 1.e4 # T to Gs
        self.a1 = 1.
        self.a2 = ellipticity
        self.ellipticity = ellipticity
        self.a = a
        self.R0 = R0
        self.n0 = n0
        self.mu = mu
        self.Z = Z
        self.aspect = R0 / a
        self.inv_aspect = a / R0
        self.charge_i = const.e * Z
        self.Mi = mu * const.Mp
        self.omega_Bi = self.charge_i * self.B0 / (self.Mi * const.c)
        self.omega_Be = const.e * self.B0 / (const.Me * const.c)
        self.v_A0 = self.B0 / np.sqrt (4. * np.pi * self.Mi * n0)
        self.omega_A0 = self.v_A0 / R0
        self.rho_A = self.v_A0 / self.omega_Bi
        self.nu = self.v_A0 / const.c
        self.S = (self.a / self.rho_A) ** 2
```

Вміст файлу `coefficients.py`:

```
import numpy as np
import matplotlib.pyplot as plt

class FMW_coefficients:
    def __init__ (self, device, density, n, boundary_type, iota,
                  cylinder=False):
        self.device = device
        self.density = density
        self.rho = density.rho
        self.device = device
        self.iota = iota
        self.inv_aspect = device.inv_aspect
        self.inv_aspect2 = device.inv_aspect ** 2
        self.omega_A0 = device.omega_A0 / device.omega_Bi
```

```
        self.omega_Apol = self.omega_A0 / self.inv_aspect
        self.Lambda_Apol = self.omega_Apol ** 2
        self.omega_A = self.omega_A0 * n
        self.Lambda_A = self.omega_A ** 2
        self.Lambda_w = (1. + (1. + np.sqrt(1.+4./self.Lambda_A)) * 0.5 *
self.Lambda_A) * self.Lambda_A
        self.a1 = device.a1
        self.a2 = device.a2
        self.R0 = device.R0
        self.kappa = device.ellipticity
        self.J = self.kappa + 1. / self.kappa
        self.cylinder = cylinder
        self.n = n
        self.boundary_type = boundary_type
        self.b = [[self.b11, self.b12],[self.b21, self.b11]]
        self.db = [[self.db11, self.db12],[self.db21, self.db11]]
        self.axx = [[self.axx11, self.a12],[self.a12, self.h]]
        self.ayy = [[self.h, self.a12],[self.a12, self.azz22]]
        self.axy = [[self.a12, None],[self.axz21, self.a12]]
        self.ayx = [[self.a12, self.axz21],[None, self.a12]]
        self.c = [[self.c11, self.c12],[self.c21, self.c22]]
        self.px = [[self.px11, self.px12],[self.px21, self.px22]]
        self.qx = [[self.qx11, self.qx12],[self.qx21, self.qx22]]
        self.py = [[self.pz11, self.pz12],[self.pz21, self.pz22]]
        self.qy = [[self.qz11, self.qz12],[self.qz21, self.qz22]]
    def is_nonlinear (self):
        return True
    def is_complex (self):
        return True
    def nxy_is_defined (self):
        return self.boundary_type == 'En0' or self.boundary_type == 'Et0'
    def get_number_of_variables (self):
        return 2
    def get_variable_names (self):
        return ['Ex', 'Ez']
    def h (self, x, z):
        if self.cylinder:
            if type(x) is float:return complex (1., 0)
            else:return np.ones (len(x), dtype=complex)
        else:
            result = 1. + self.inv_aspect * x
            if type(x) is float:return complex (result, 0.)
            else:return result.astype(complex)
    def anti_h (self, x, z):
        return -self.h (x, z)
    def get_sizes (self):
        return self.a1, self.a2
    def b11 (self, x, z, Lambda):
        h = self.h(x,z)
        h2 = h**2
        return  (h2 * h * self.rho(x,z) / (1. - Lambda * h2) * (Lambda /
self.Lambda_Apol))
    def b12 (self, x, z, Lambda):
```

```
        h2 = self.h(x,z)**2
        return  (h2 * h2 * self.rho(x,z) / (1. - Lambda * h2) *
(Lambda**1.5 / self.Lambda_Apol * 1j))
    def b21 (self, x, z, Lambda):
        h2 = self.h(x,z)**2
        return  (h2 * h2 * self.rho(x,z) / (1. - Lambda * h2) *
(Lambda**1.5 / self.Lambda_Apol * (-1j)))
    def db11 (self, x, z, Lambda):
        h = self.h(x,z)
        h2 = h**2
        return  (h2 * h * self.rho(x,z) / (1. - Lambda * h2) ** 2/
self.Lambda_Apol)
    def db12 (self, x, z, Lambda):
        h2 = self.h(x,z)**2
        return  (h2 * h2 * self.rho(x,z) * (3. - Lambda * h2)/ (1. -
Lambda * h2) ** 2* (np.sqrt(Lambda) / self.Lambda_Apol * 0.5j))
    def db21 (self, x, z, Lambda):
        h2 = self.h(x,z)**2
        return  (h2 * h2 * self.rho(x,z) * (3. - Lambda * h2) / (1. -
Lambda * h2) ** 2 * (np.sqrt(Lambda) / self.Lambda_Apol * (-0.5j)))
    def show_density_profile (self):
        fig, ax = plt.subplots()
        x_list = np.linspace (0., 1., 201)
        ax.plot (x_list, [self.rho(x, 0.) for x in x_list])
        ax.set_title ('normalized density profile')
        plt.show()
    def nxy (self, x, z):
        d1 = x / self.a1**2
        d2 = z / self.a2**2
        norm = np.sqrt (d1**2 + d2**2)
        if self.boundary_type == 'Et0':return d1 / norm, d2 / norm
        elif self.boundary_type == 'En0':return -d2 / norm, d2 / norm
        else:raise AssertionError ('No method to build coefficients for
boundary condition type "{}"'.format(self.boundary_type))
    def show_density_map (self, triang, cmap='plasma'):
        z = np.array ([self.rho (*point) for point in zip (triang.x,
triang.y)])
        fig, ax = plt.subplots(figsize=(11,6))
        ax.set_aspect('equal')
        tcf = ax.tricontourf (triang, z, cmap=cmap)
        fig.colorbar (tcf)
        plt.show()
    def show_h_map (self, triang, cmap='plasma'):
        z = np.array ([self.h (*point).real for point in zip (triang.x,
triang.y)])
        fig, ax = plt.subplots(figsize=(11,6))
        ax.set_aspect('equal')
        tcf = ax.tricontourf (triang, z, cmap=cmap)
        fig.colorbar (tcf)
        plt.show()
    def k_trans2 (self, x, z, omega2):
        h2 = self.h(x,z).real ** 2
        om2 = omega2 * (h2 / self.omega_A0) ** 2 * self.rho(x,z)
        oh = (1. - omega2 * h2)
```

```
        M = om2 - self.n**2 * oh
        return M / (h2 * oh) - omega2 * om2**2 / (M * oh)
    def r2 (self, x, z):
        return np.sqrt (x**2 + z**2)
    def a12 (self, x, z):
        iota = self.iota (self.r2 (x, z))
        return ((-0.5 * self.inv_aspect2) * iota**2* x * z * self.h(x,z))
    def axx11 (self, x, z):
        r2 = self.r2 (x, z)
        return (((self.inv_aspect / self.kappa) * self.iota(r2) * z) **
2* self.h(x,z))
    def azz22 (self, x, z):
        r2 = self.r2 (x, z)
        return (((self.kappa * self.inv_aspect) * self.iota(r2) * x) **
2* self.h(x,z))
    def axz21 (self, x, z):
        iota = self.iota (self.r2 (x, z))
        kappa2 = self.kappa**2
        return (((self.inv_aspect2 * 0.5) * iota**2* ((self.kappa * x) **
2 + (z / self.kappa) ** 2) - 1.)* self.h(x,z))
    def c11 (self, x, z):
        h = self.h (x, z)
        return (((self.iota (self.r2 (x, z)) / self.kappa) ** 2 * h+
self.n**2 / h) * self.inv_aspect2)
    def c22 (self, x, z):
        h = self.h (x, z)
        return (((self.iota (self.r2 (x, z)) * self.kappa) ** 2 * h+
self.n**2 / h) * self.inv_aspect2)
    def c12 (self, x, z):
        return ((1j * self.n * self.inv_aspect2 * self.J)*
self.iota(self.r2(x,z))**2)
    def c21 (self, x, z):
        return ((-1j * self.n * self.inv_aspect2 * self.J)*
self.iota(self.r2(x,z))**2)
    def px11 (self, x, z):
        return ((-1j * self.n / self.kappa * self.inv_aspect2) *
self.iota(self.r2(x,z)) * z)
    def px12 (self, x, z):
        iota = self.iota (self.r2(x,z))
        uxi = ((self.inv_aspect2 * (0.5 + 1./self.kappa**2))* iota**2 * z
* self.h (x, z))
        return uxi + (1j * self.n * self.kappa * self.inv_aspect2) * iota
* x
    def px21 (self, x, z):
        iota = self.iota (self.r2 (x, z))
        return -self.inv_aspect2 * iota**2 * z * self.h(x,z)
    def px22 (self, x, z):
        iota = self.iota (self.r2 (x, z))
        return (-0.5 * self.inv_aspect2) * iota**2 * x * self.h(x,z)
    def qx11 (self, x, z):
        return np.conjugate (self.px11 (x, z))
    def qx12 (self, x, z):
        return self.px21 (x, z)
    def qx21 (self, x, z):
```

```
        return np.conjugate (self.px12 (x, z))
    def qx22 (self, x, z):
        return self.px22 (x, z)
    def pz11 (self, x, z):
        iota = self.iota (self.r2 (x, z))
        return (-0.5 * self.inv_aspect2) * iota**2 * z * self.h(x,z)
    def pz12 (self, x, z):
        iota = self.iota (self.r2 (x, z))
        return self.inv_aspect2 * iota**2 * x * self.h(x,z)
    def pz21 (self, x, z):
        iota = self.iota (self.r2(x,z))
        ueta = ((-self.inv_aspect2 * (0.5 + 1./self.kappa**2))* iota**2 *
x * self.h (x, z))
        return (ueta + (-1j * self.n * self.kappa * self.inv_aspect2)*
iota * z)
    def pz22 (self, x, z):
        return ((1j * self.n * self.kappa * self.inv_aspect2)*
self.iota(self.r2(x,z)) * x)
    def qz11 (self, x, z):
        return self.pz11 (x, z)
    def qz12 (self, x, z):
        return np.conjugate (self.pz21 (x, z))
    def qz21 (self, x, z):
        return self.pz12 (x, z)
    def qz22 (self, x, z):
        return np.conjugate (self.pz22 (x, z))
```

Вміст файлу `basis.py`:

```
import numpy as np
import matplotlib.pyplot as plt
from gauss import *
class Quadratic_Basis2D:
    def __init__ (self, grid, fixed_boundary=True, quad_order=5):
        self.grid = grid
        self.index = []
        self.support = []
        self.boundary = []
        self.top_coordinates = []
        for n in range(grid.nNodes):
            if not (fixed_boundary and grid.boundary_node[n]):
                self.index.append ([n,0])
                self.support.append (grid.node_neighbors[n])
                self.boundary.append (grid.boundary_node[n])
                self.top_coordinates.append (grid.points[n,:])
        for n in range(grid.nEdges):
            if not (fixed_boundary and grid.boundary_edge[n]):
                self.index.append ([n,1])
                self.support.append (set(grid.edge_adjacents[n,:]) - {-
1})
                is_boundary = grid.boundary_edge[n]
                self.boundary.append (is_boundary)
                self.top_coordinates.append(0.5 *
(grid.points[grid.edges[n,0],:] + grid.points[grid.edges[n,1],:]))
```

```
        self.index = np.array (self.index)
        self.top_coordinates = np.array (self.top_coordinates)
        self.size = self.index.shape[0]
        self.build_pointers_to_supported_functions()
        self.quad = Gauss2D (quad_order)
        self.nInternal = self.boundary.count(False)
        self.nBoundary = self.boundary.count(True)
        print ('Basis is ready')
        print ('Basis size: {}'.format(self.size))
        print ('Basis internal elements: {}'.format(self.nInternal))
        print ('Basis boundary elements: {}'.format(self.nBoundary))
    def get_size (self):
        return self.size
    def get_number_of_internal_elements (self):
        return self.nInternal
    def get_number_of_boundary_elements (self):
        return self.nBoundary
    def is_boundary (self, n):
        return self.boundary[n]
    def get_top_coordinates (self, element):
        return self.top_coordinates[element,:]
    def build_pointers_to_supported_functions (self):
        self.supported = []
        for cell_no in range (self.grid.nCells):
            func_set = set()
            for func_no, func_support in enumerate (self.support):
                if cell_no in func_support:func_set.add (func_no)
            self.supported.append (func_set)
    def get_support (self):
        return self.support
    def get_supported (self):
        return self.supported
    def get_index (self, n):
        return self.index[n,:]
    def phi (self, x, z, node_index, cell_index, deriv='no'):
        cell = self.grid.cells[cell_index,:]
        x1, z1 = self.grid.points[node_index,:]
        i1 = np.where (cell==node_index)[0][0]
        i2 = (i1+1) % 3
        x2, z2 = self.grid.points[cell[i2],:]
        i3 = (i2+1) % 3
        x3, z3 = self.grid.points[cell[i3],:]
        denom = 2. * self.grid.cell_square[cell_index]
        alpha = (z2 - z3) / denom
        beta = (x3 - x2) / denom
        hat = 1. + alpha * (x-x1) + beta * (z-z1)
        if deriv == 'no':return 2. * hat * hat - hat
        elif deriv == 'x':return alpha * (4 * hat - 1.)
        elif deriv == 'y':return beta * (4 * hat - 1.)
    def psi (self, x, z, edge_index, cell_index, deriv='no'):
        edge = self.grid.edges[edge_index,:]
        cell = self.grid.cells[cell_index,:]
```

```
        for i2 in range(3):
            if cell[i2] not in edge: break
        i1 = (i2-1) % 3
        i3 = (i2+1) % 3
        x1, z1 = self.grid.points[cell[i1],:]
        x2, z2 = self.grid.points[cell[i2],:]
        x3, z3 = self.grid.points[cell[i3],:]
        denom = self.grid.cell_square[cell_index] ** 2
        delta3 = (x-x2) * (z3-z2) - (z-z2) * (x3-x2)
        delta1 = (x-x2) * (z1-z2) - (z-z2) * (x1-x2)
        if deriv == 'no':return - delta3 * delta1 /
self.grid.cell_square[cell_index] ** 2
        elif deriv == 'x':return (((z2-z3) * delta1 + (z2-z1) * delta3)
/self.grid.cell_square[cell_index] ** 2)
        elif deriv == 'y':return (((x3-x2) * delta1 + (x1-x2) * delta3)
/self.grid.cell_square[cell_index] ** 2)
    def global_basis_function (self, x, z, j):
        cell_index = self.grid.trifinder (x, z)
        if cell_index == -1:return 0.
        element, kind = self.index[j,:]
        if kind==0:
            if element not in self.grid.cells[cell_index,:]:return 0.
            return self.phi (x, z, element, cell_index)
        else:
            if cell_index not in
self.grid.edge_adjacents[element,:]:return 0.
            return self.psi (x, z, element, cell_index)
    def show_basis_function (self, index, density=1):
        if density==1:triang = self.grid.delaunay
        else:triang = self.grid.refine (density)
        z = [self.global_basis_function (*point, index) for point in zip
(triang.x, triang.y)]
        fig, ax = plt.subplots()
        ax.set_aspect('equal')
        tcf = ax.tricontourf(triang, z)
        fig.colorbar(tcf)
        ax.tricontour(triang, z, colors='k')
        ax.set_title('Basis function '.format(index))
        plt.show()
    def function (self, x, z, j, cell_index, deriv='no'):
        if self.index[j,1] == 0:return self.phi (x, z, self.index[j,0],
cell_index, deriv)
        else:return self.psi (x, z, self.index[j,0], cell_index, deriv)
    def get_value (self, x, z, vector, deriv='no'):
        value = 0.
        cell_index = self.grid.trifinder(x,z).item()
        if cell_index == -1:return 0.
        for j, component in enumerate (vector):
            if cell_index in self.support[j]:value += (self.function(x,
z, j, cell_index, deriv)* component)
        return value
    def plot_function_map (self, vector, density=1, fig=None,
ax=None,title=None, corrector=None):
        if fig is None or ax is None:fig, ax = plt.subplots()
```

```
        ax.set_aspect('equal')
        if density==1:triang = self.grid.delaunay
        else:triang = self.grid.refine (density)
        z = [self.get_value (*point, vector)for point in zip (triang.x,
triang.y)]
        tcf = ax.tricontourf (triang, z)
        fig.colorbar (tcf)
        ax.tricontour (triang, z, colors='k')
        if title is not None:ax.set_title (title)
        plt.show()
    def matrix_element (self, kernel, j1, j2, deriv1='no', deriv2='no'):
        joint = self.support[j1] & self.support[j2]
        result = 0.
        for cell in joint:
            if cell == -1: continue
            result += self.quad (lambda x,z: kernel(x,z)
                                 * self.function (x, z, j1, cell, deriv1)
                                 * self.function (x, z, j2, cell,
deriv2),
                                 * self.grid.get_cell_coords (cell),
                                 self.grid.cell_square[cell])
        return result
    def disjoint (self, i, j):
        return len (self.support[i] & self.support[j]) == 0
```