Національний технічний університет України
"Київський політехнічний інститут"

Фізико-технічний інститут

Курсова

з кредитного модуля «Методи обчислень»

на тему:

"Числове розв'язання диференційних рівнянь у часткових похідних"

Варіант № 8

Виконав:
студент 3 курсу ФТІ
Юрців Ілля Васильович

Київ - 2016

Зміст

1 Попередній огляд

1.1 Постановка задачі

У даній роботі запропоновано числовий розв'язок р-ня (1), що моделює процес розмноження бактерій, який здійснюється шляхом їх ділення.

$$\frac{\partial u}{\partial t} = \alpha(m_0 - \beta u)u + \sigma\Delta u; \tag{1}$$

Надалі будемо вважати, що процес відбувається в прямокутній області $\Omega \in \mathbb{R}^2$ ($0 \le x \le a$, $0 \le y \le b$). Змінна u позначає концентрацію мікроорганізмів у визначеній області. Порогове значення концентрації $u_{max} = \frac{m_0}{\beta}$.

Початкові та граничні умови:

$$\begin{cases} u|_{t=0} = U(\vec{x}), & \forall \vec{x} \in \Omega \\ u|_{\Gamma} = u_{\Gamma}, & \Gamma = \partial\Omega \end{cases}$$

де $U(\vec{x})$ — гладка класу $\mathbf{C}^2(\Omega)$ функція; крім того для узгодженості початкових та граничних умов покладемо $U(\vec{x})|_{\Gamma} = u_{\Gamma}$ і вважатимемо, що протягом усього розвитку системи значення u_{Γ} залишається постійним.

1.2 Розгляд рівняння

Рівняння (1) є узагальненням для двовимірного простору відомого р-ня Фішера (або частковим випадком з квадратичною нелінійністю р-ня Колмогорова-Петровського-Піскунова). В контексті популяційної динаміки одновимірний безрозмірний варіант цього р-ня був розглянутий Р. Фішером у 1937 р. у [3] де (1) було запропоновано для опису просторового росподілу домінантних алелей і був проведений аналіз його розв'язку у вигляді біжучих хвиль. Стислий але змістовний огляд узагальнення цього р-ня Колмогоровим, а також умови існування його розв'язку у вигляді біжучих хвиль для одновимірного випадку можна знайти в [7].

Модель, що задається р-ням (1), відноситься до типу реакційно-дифузійних моделей. Такі моделі поєднують в собі опис просторово розподіленої хімічної (біологічної, тощо) реакції — в нашому р-ні представлена логістичним нелінійним членом, — з описом дифузії реагентів через субстрат — член з лапласіаном у (1). Зазвичай такі моделі представляються системою ДРЧП, невідомі ф-ї в якій описують концентрації (зазвичай, або ж деяку іншу характеристику) реагентів (видів, популяцій у застосуванні до біології), що взаємодіють між собою і розповсюджуються у просторі. У нашому випадку розглядається одновидова модель розмноження бактерій на площині з одночасною дифузією. Ділення бактерій описується логістичним членом (який є узагальненням звичайної експоненційної моделі), що введений у (1) для врахування граничної спроможності середовища вміщати у себе популяцію. Без цього

нелінійного члена (1) являє собою звичайне параболічне р-ня дифузії, тож модель, як було вказано, поєднує в собі опис розмноження бактерій разом з їх одночасним перерозподілом у просторі.

У таблиці нижче наведена інтерпретація параметрів р-ня з точки зору побудови моделі та біологічного змісту останніх, подробиці можуть бути знайдені у [5].

Parameter	Meaning
α	Coef. of linear dependency between $\mathbf{R}(u)$ and "nutrient"
β	Coef. between decrease of "nutrient"and increase of population
m_0	Constant of availability of "nutrient"
σ	Coef. of diffusion
$n_0 = \frac{m_0}{\beta}$	Limit carrying capacity

Тут символом $\mathbf{R}(u)$ позначено логістичний член $\alpha m_0 u - \alpha\beta u^2$, який називають "growth rate або темп росту; він визначає характер "реакційного"вкладу у реакційно-дифузних моделях. Як вказано в [5] для подібних моделей, що розглядаються у біології, саме квадратичні нелінійні члени є найбільш характерними. Зауважимо, що знову згідно [5] коефіцієнт $-\alpha\beta$ визначає (оскільки маємо одну популяцію) внутрішньовидову конкуренцію при $\alpha \times \beta > 0$. Для біологів саме цей випадок є найцікавішим для досліджень у даній моеделі. Якщо коеф-т при квадратичному члені є додатнім, то цей випадок описує свого роду “коменсалізм”, коли взаємодія окремих особин дає позитивний ефект для розвитку обох. Тим не менш цей окремий випадок нами не розглядався детально, оскільки, як показав експеримент, запропонована у роботі схема дискретизації виявляється суттево нестійкою при $\alpha \times \beta < 0$ (зокрема при $\alpha < 0$, навіть для дуже малих значеннях останнього).

2 Вибір скінченнорізницевої схеми. Дискретизація

2.1 Попередні зауваження. Оцінки похідних у рівнянні дифузії за допомогою ряду Тейлора

Спочатку для демонстрації загальних міркувань розглянемо одновимірне р-ня дифузії

$$\frac{\partial u}{\partial t} = \sigma \frac{\partial^2 u}{\partial^2 x} \tag{2}$$

і отримаємо явний вигляд скінченнорізницевих операторів, якими ми будемо надалі користуватись, з розкладу в ряд Тейлора.

Розклад в ряд Тейлора значення функції u в околі точки $(x + \Delta x)$, де значення $u(x)$ — відоме, може бути записано як:

$$u(x + \Delta x) = u(x) + \Delta x \frac{\partial u}{\partial x} + \frac{\Delta x^2}{2} \frac{\partial^2 u}{\partial^2 x} + \frac{\Delta x^3}{6} \frac{\partial^3 u}{\partial^3 x} + O\left(\Delta x^4\right); \tag{3}$$

аналогічно для точки $(x - \Delta x)$:

$$u(x - \Delta x) = u(x) - \Delta x \frac{\partial u}{\partial x} + \frac{\Delta x^2}{2} \frac{\partial^2 u}{\partial^2 x} - \frac{\Delta x^3}{6} \frac{\partial^3 u}{\partial^3 x} + O\left(\Delta x^4\right) \tag{4}$$

Розв'язуючи (3) відносно $\frac{\partial u}{\partial x}$ і опускаючи всі члени вищих порядків, отримуємо

$$\frac{\partial u}{\partial x} = \frac{u(x + \Delta x) - u(x)}{\Delta x} + O\left(\Delta x\right)$$

аналогічно для (4) отримуємо

$$\frac{\partial u}{\partial x} = \frac{u(x) - u(x - \Delta x)}{\Delta x} + O\left(\Delta x\right)$$

У англомовній літературі останні два вирази прийнято називати *forward і backward difference operator* відповідно. Крім того, віднявши (4) від (3) отримаємо центрований різницевий оператор:

$$\frac{\partial u}{\partial x} = \frac{u(x + \Delta x) - u(x - \Delta x)}{2\Delta x} + O\left(\Delta x^2\right),$$

який, як бачимо, має вищий порядок точності, ніж два попередні.

Зрештою, додамо (4) і (3) й розв'яжемо відносно $\frac{\partial^2 u}{\partial^2 x}$:

$$\frac{\partial^2 u}{\partial^2 x} = \frac{u(x + \Delta x) - 2u(x) + u(x - \Delta x)}{\Delta x^2} + O\left(\Delta x^2\right)$$

Тепер (2) може бути переписано як

$$u(t + \Delta t) - u(t) = s_x \left(u(x + \Delta x) - 2u(x) + u(x - \Delta x)\right),$$

де $s_x = \frac{\sigma \Delta t}{\Delta x^2}$ — так зване число дифузії, яке є дуже важливим при оцінці стійкості різницевих

схем такого типу.

Нехай тепер потрібно розв'язати (2) на $[0,L]$ на проміжку часу від $t = 0$ до $t = T$ і відомо, що $u(0,x) = u(t,0) = u(t,L) = 0$. Введемо точки $n\Delta t$, $j\Delta x$,

$$n = 0, \dots \ N,\ j = 0, \dots \ M \text{ — цілі числа, } N = T/\Delta t,\ M = L/\Delta x$$

і перепишемо (2) згідно з попереднім, використовуючи нові позначення:

$$u_j^{n+1} - u_j^n = s_x \left(u_{j+1}^n - 2u_j^n + u_{j-1}^n \right),$$

де $u_j^n = u(n\Delta t, j\Delta x)$; окрім того початкові та граничні умови запишуться як:

$$u_j^0 = 0,\ u_0^n = u_M^n = 0.$$

Подібні міркування, що можуть бути легко відтворені для вищих розмірностей, і приводять якраз до поняття скінченнорізницевих схем як методу числового розв'язання ДРЧП. Замінюючи у постановці конкретної задачі часткові похідні їх наближеннями через різницеві оператори і обираючи фіксовані кроки Δt по часу і Δx $(\Delta y \dots)$ у просторі (для кожного окремого випадку різницевої схеми кроки можуть бути нерівномірними, змінюватись у процесі розв'язку і т.д.), ми переводимо задачу у скінченновимірний простір — так звану *сітку*, — для кожного вузла j якої ми знаходимо розв'язок у деякий момент часу n з отриманих різницевих р-нь, а для точок простору, які не потрапили у сітку, значення може бути відтворене, наприклад, інтерполюванням.

З вищенаведеного також легко бачити, що вибір різницевих операторів, які входять до різницевої схеми, впливає на її порядок точності. Це має велике значення для вибору конкретної схеми для розв'язку тієї чи іншої крайової задачі. Наприклад, для вищерозглянутої схеми порядок точності є $O\left(\Delta t, \Delta x^2\right)$. Звичайно доцільним є покращити порядок точності і *стійкість* (про що ще йтиме мова далі), проте це має свою ціну, а саме: ми приходимо до *неявних* схем, для яких на кожному кроці по часу потрібно розв'язувати СЛАУ (у випадку лінійної однорідної крайової задачі), або ж у більш загальному випадку, якщо не вносити спеціальні корективи, навіть систему нелінійних р-нь.

2.2 Вибір різницевої схеми для р-ня Фішера. Схеми типу Кранка-Ніколсона

Для підвищення порядку точності було запропоновано багато модифікацій найпростішої явної схеми. Детальний огляд і узагальнення у табличному вигляді видів різницевих схем для рівнянь параболічного типу наведено у [2] (стор. 192-195). Одним із найбільш розповсюджених є клас неявних схем, який отримується таким чином. Праву частину вихідного р-ня записуємо як суму двох різницевих операторів по просторовим координатам, один з яких береться у точці $n+1$ по часу з вагою λ, а інший у точці n по часу з вагою $(1-\lambda)$. Явно це виглядає для нашого р-ня (1) так:

$$\frac{u_{j,k}^{n+1}-u_{j,k}^{n}}{\Delta t}=\sigma\frac{\lambda(\delta_j^2u)_k^{n+1}+(1-\lambda)(\delta_j^2u)_k^n}{\Delta x^2}+ \\ +\sigma\frac{\lambda(\delta_k^2u)_j^{n+1}+(1-\lambda)(\delta_k^2u)_j^n}{\Delta y^2}+\lambda(f(u))_{jk}^{n+1}+(1-\lambda)(f(u))_{jk}^n \quad (5)$$

де $\left(\delta_j^2u\right)_k^n=u_{j+1,k}^n-2u_{j,k}^n+u_{j-1,k}^n$, $f(u)$ — реактивний логістичний член р-ня (1).
За рахунок такого усереднення похідна по часу де-факто розглядається у точці з індексом $(n+\frac{1}{2})$, тобто по часу ми маємо записаний у інших позначеннях центрований різницевий оператор, який, як ми бачили, має порядок точності $O\left(\Delta t^2\right)$, отже в цілому даний клас різницевих схем має порядок точності $O\left(\Delta t^2,\Delta x^2,\Delta y^2\right)$.

При $\lambda=\frac{1}{2}$ отримуємо відому схему Кранка-Ніколсона. Загальний "шаблон" для схем цього класу зображений на Рис. 1.

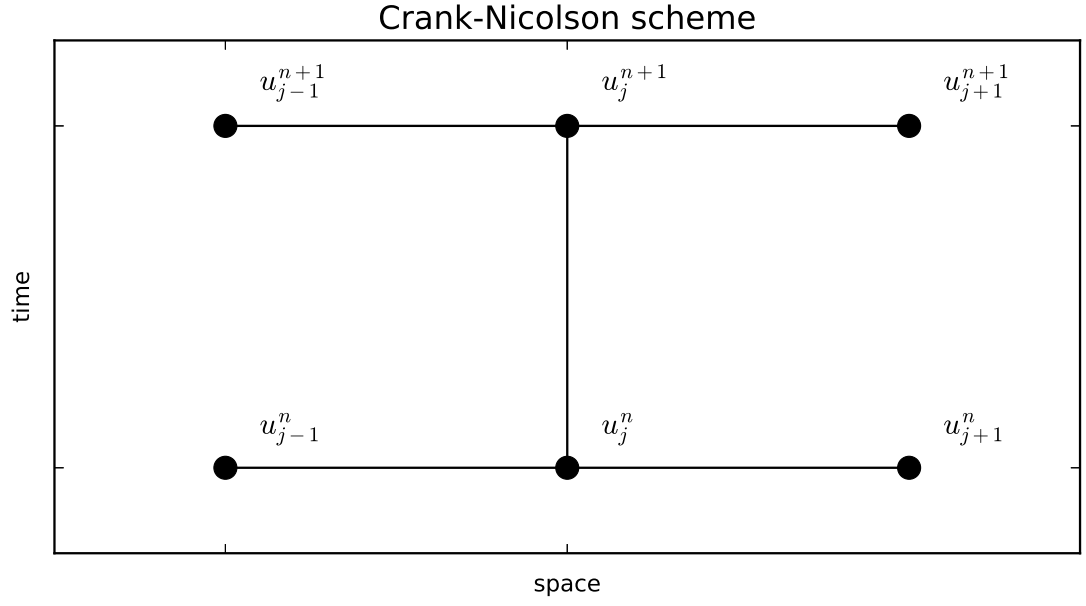


Рис. 1

Як неважко бачити, завдяки подібній дискретизації нашого р-ня ми отримуємо на кожному кроці по часу систему р-нь відносно невідомих значень шуканої ф-ї у точках з індексом $(n+1)$, крім того, поки що нелінійних (через квадратичний член). Однак в подальшому

ми позбавимось від цієї незручності і використаємо усі переваги даної схеми саме для лінеаризованого випадку. Розмірність такої системи $[(M+1)\times(N+1)]^2$, де M та N — к-ть вузлів по осям Ox і Oy відповідно, як у прикладі з попереднього пункту. Отже при достатньо маленьких кроках дискретизації розв'язування подібної системи може вимагати багато обчислювальних ресурсів; тим не менше, у лінійному випадку така система є дуже розрідженою і має разом з граничними умовами (які в даному випадку представлені у формі Дирихле) чітко виражений 5-діагональний вид, що дозволяє нам використовувати оптимізовані формати збереження і методи розв'язання розріджених систем.

2.3 Лінеаризація отриманої системи

Як уже було вказано, якщо використовувати для дисткретизації (1) схему типу Кранка-Ніколсона у її стандартному вигляді, то отримана система р-нь (5) є нелінійною, що призводить до суттєвих незручностей з обчислювальної точки зору (необхідності використовувати своєрідні, більш складні ніж для розріджених матриць, формати збереження, виключно ітераційні алгоритми розв'язку, накшталт Ньютона-Рафсона, або метод Крилова оптимізації вектор-функцій багатьох змінних, які вимагають обчислень якобіанів порядку (для нашого випадку) $[(M+1)\times(N+1)]^4$, що вимагає ресурсів ще більше, ніж зберігання 5-діагональної розрідженої матриці у звичайному "densed" вигляді). Тому нами запропоновано дещо інший підхід, який базується на зведенні вже отриманої системи до лінійного вигляду.

Нелінійність у (5) вносить квадратичний член $\left(u_{j,k}^{n+1}\right)^2$, отже зводити до лінійного вигляду будемо саме його. Розглянемо похідну за часом від функції $u^2(t,x,y)$:

$$\frac{\partial\left(u^2\right)}{\partial t}=2u\frac{\partial u}{\partial t}$$

що, використовуючи апроксимацію різницевим оператором по часу, може бути переписано як

$$\frac{(u^2)_{jk}^{n+1}-(u^2)_{jk}^{n}}{\Delta t}=\frac{2u_{jk}^{n}\left(u_{jk}^{n+1}-u_{jk}^{n}\right)}{\Delta t}$$

звідки маємо

$$\left(u_{jk}^{n+1}\right)^2=\left(u_{jk}^{n}\right)^2+2u_{jk}^{n}\left(u_{jk}^{n+1}-u_{jk}^{n}\right)$$

де величини у правій частині на кожному кроці по часу відомі. Тепер вводячи позначення $\omega_{jk}=\left(u_{j,k}^{n+1}-u_{j,k}^{n}\right)$ і записуючи граничні (за постановкою задачі значення на границі — константніпо часу) умови

$$\omega_{0,k}=\omega_{M,k}=\omega_{j,0}=\omega_{j,N}=0$$

ми приходимо до лінеаризованого варіанту системи (5), яка тепер є 5-діагональною СЛАУ. Розв'язуючи її на кожному кроці відносно ω_{jk} і покладаючи $u_{j,k}^{n+1}=\omega_{jk}+u_{j,k}^{n}$ приходимо до розв'язку початкової задачі.

3 Аналіз стійкості обраної різницевої схеми

Для аналізу стійкості відповідної лінеаризованої різницевої схеми (5) будемо користуватися спектральним методом фон Неймана, детальний розгляд якого та доведення відповідних теорем необхідності та достатності (в першу чергу для лінійного випадку) можна знайти у [1, 2].

Для початку визначимо згідно з [1] різницеву схему як *стійку відносно збурень вхідних даних*, якщо для її розв'язку виконано:

$$\max_{j,k} |u^n_{j,k}| \leq C \max_{j,k} |u^0_{j,k}|,\ \forall n = 0, \ldots, T,\ C = const,\ |C| < \infty$$

для довільної обмеженої ф-ї $u^0_{j,k} = \psi_{j,k}$.

Слідуючи за [2] знайдемо коефіцієнт переходу (підсилення) ξ для Фур'є-компоненти точного розв'язку різницевої схеми. Для нелінійних ДРЧП де-факто він залежить не лише від набору констант, що характеризують р-ня, і обраних кроків сітки, але й, власне кажучи, від самих розв'язків, що отримуються, тому можна лише сподіватись, що стійкість буде мати місце до деякого моменту часу t_1; крім того часто строгі аналітичні оцінки на коефіцієнт переходу є важкими і громіздкими, тому у даній роботі перевірку отриманої умови (в якій є явна залежність від $u^n_{j,k}$) було реалізовано у програмному коді і у випадку коли умова перестає виконуватись моделювання припинялось (в якості покращення методу, можна було б підганяти крок по часу, щоб задовільнити умову стійкості).

Розглянемо n-ту Фур'є-компоненту точного розв'язку схеми: $u^n_{j,k} = \xi^n e^{imj\Delta x} e^{ilk\Delta y}$; підставимо її у (5) з лінеаризованим квадратичним членом і після скорочень, виражаючи ξ отримаємо:

$$\xi = \frac{1 - (1-\lambda)\left[\Delta t \alpha m_0 + 2\psi\left(m\Delta x, l\Delta y\right)\right] - (1-2\lambda)\Delta t \alpha\beta \times u^n\left(m\Delta x, l\Delta y\right)}{1 - \lambda\left[\Delta t m_0 \alpha - 2\psi\left(m\Delta x, l\Delta y\right)\right] + 2\lambda\Delta t \alpha\beta \times u^n\left(m\Delta x, l\Delta y\right)} \tag{6}$$

де $\psi(m\Delta x, l\Delta y) = S_x\{1 - \cos(m\Delta x)\} + S_y\{1 - \cos(l\Delta y)\}$, $S_x = \frac{\sigma\Delta t}{\Delta x^2}$, $S_y = \frac{\sigma\Delta t}{\Delta y^2}$ — відповідні числа дифузії.

Відомо, що необхідною умовою стійкості є

$$\max_{m,l} |\xi| < 1$$

отже саме її і будемо використовувати у програмній реалізації.

4 Опис програмної реалізації

4.1 Об'єктна модель

Програмна реалізація розв'язку р-ня (1) була розроблена за допомогою мови Python, версія 2.7.6, та додаткових програмних пакетів для деяких специфічних задач лінійної алгебри, що виникають у процесі розв'язку, а саме: **scipy (0.17.0)** (для представлення розрідженої матриці системи у форматі DIAGONAL та використання відповідних солверів для розріджених систем) та **numpy (1.11.0)** для базових представлень масивів та операцій лінійної алгебри. Відповідні лістинги наведені у **Додатку I**.

Об'єктна модель реалізації складається з двох класів: **CNModel** та **CNSolver**. Перший репрезентує модель задачі, тобто зберігає в якості своїх атрибутів параметри рівняння та схеми дискретизації, функцію, що задає початкові та граничні умови, а також інкапсулює реалізацію методів, що готують модель для розв'язання та є основними методами, які викликає клієнтський код солвера. Нижче наведено параметри, що передаються в конструктор класу при створенні моделі:

Parameters	Description
alpha, beta, m_0, sigma	$\alpha, \beta, m_0, \sigma$ — Параметри р-ня (**float**)
dt, dx, dy	$\Delta t, \Delta x, \Delta y$ — кроки по часу, Ox, Oy відповідно (**float**)
T, M, N	Кількість вузлів по часу, Ox, Oy відповідно (**integer**)
lambd	λ — ваговий коеф-т (**float, belongs to [0, 1]**)
init_cond	$U(\vec{x})$ — ф-я початкових умов (**either callable or float**)

Методи классу включають в себе (Listing 1, Додаток I): **_get_footprint**, що одноразово обчислює коеф-ти в системі рівнянь, які стоять в кожному рядку при відповідних членах, **get_initial_state**, що формує сітку початкових значень, **prepare**, що є основним методом, який викликає клас **Solver**, і який готує модель до запуску солвера, ініціалізуючи всі необхідні параметри та **stability_factor**, що обчислює значення у вузлі $(i\Delta x, j\Delta y)$ ф-ї ψ, яка виникає у виразі для коеф-та підсилення (6).

Другий клас є своєрідним фреймворком для проведення усіх операцій (основних та підготовчих) по розв'язуванню задачі та відповідному зберіганню розв'язків; він жорстко зв'язується з моделлю, яку ми передаємо йому (**CNSolver.fit(CNModel)**). Для розв'язання кожної зі згенерованих моделей повинен бути створений свій солвер, у який "фітиться"модель і який агрегує відповідні розв'язки. Розв'язки у буфері солвера у процесі обробки зберігаються у так званій *flat* моделі — сітка, представлена у вигляді одновимірного numpy-масиву, рядок за рядком. Після обробки матричний вигляд відновлюється і записується у остаточний багатовимірний numpy-масив, кожен елемент якого представляє розв'язок на деякому кроці часу.

У наступному пункті ми розглянемо специфічні методи роботи з розрідженими

масивами даних, які використовуються класом **CNSolver** для динамічної генерації матриці системи у відповідному представленні та для оптимального розв'язання системи, представленої у подібному вигляді.

4.2 Формати представлень розріджених масивів. Проекційні алгоритми

Генерувати матрицю системи необхідно на кожній ітерації, оскільки її коеф-ти явно залежать від $u^n_{j,k}$, саме тому цю допоміжну операцію було реалізовано всередині класу **CNSolver** — відповідні значення додатково зберігаються у флет-моделі у буфері солвера і допомагають генерації. Як бачимо це вимагає значних витрат обчислювальних ресурсів, в тому числі й пам'яті машини, особливо якщо вимагати достатньо високої точності розв'язку. Матриця системи, як було вже вказано, є 5-діагональною і, за рахунок цього, дуже сильно розрідженою. Тому нами був використаний динамічний метод заповнення матриці виключно ненульовими ел-ми і подальше її конвертування в формат **DIAGONAL**. Сам формат описаний нижче. Динамічне заповнення забезпечив модуль **scipy.sparse**, у якому реалізований допоміжний формат **lil_matrix**, який є своєрідним узагальненим шаблоном для будь-якого формату збереження розріджених матриць. Він дозволяє додавати ненульовий елмент з вказанням конкретної позиції і в подальшому трансформувати матрицю для використання у солвері.

У нашій реалізації ми використовували формат представлення **DIAGONAL**, на противагу іншому найбільш розповсюдженому формату **CSR** (Compressed Sparse Row), оскільки він безпосередньо був розроблений для n-діагональних розріджених матриць і конвертація у нього для таких матриць виконується дещо швидше. Детальний огляд конкретних форматів збереження можна знайти у [6].

Збереження у форматі **DIAGONAL** передбачає наступне. Діагоналі зберігаються у прямокутному масиві **DIAG** розмірності $n \times Nd$, де Nd — к-ть ненульових діагоналей. Кожень стовпець представляє діагональ і у діагоналей, що мають менше n елементів на відповідних позиціях стоять деякі фіктивні "заповнювачі". Зсуви діагоналей відносно головної мають бути відомі і записуються вони у окремий одновимірний масив **IOFF** довжини Nd. Значення 0 відповідає головній діагоналі, -1 — першій субдіагоналі у нижньому трикутнику, +1 — першій супердіагоналі у верхньому трикутнику і т.д. Послідовність вектор-стовпчиків, що представляють діагоналі та їх зсувів у масиві **IOFF** повинні співпадати, при цьому абсолютне розташування самих вектор-стовпчиків відносно один одного у **DIAG** зазвичай суттєвого значення не має, хоча в деяких випадках може бути корисним прямий порядок.

Система р-нь, що представлена розрідженою матрицею LHS та вектор-стовпчиком RHS, що генеруються солвером на кожному кроці, представлена у специфічному форматі і повинна розв'язуватись за допомогою відповідних алгоритмів. Модуль **scipy.sparse.linalg** пропонує для розріджених матриць серію різних алгоритмів, що мають свої переваги та недоліки і ефективність та застосовність яких часто сильно залежить від властивостей

матриці лівої частини системи, тому перед нами постала проблема вибору найкращого із доступних методів. Нами було проведено невелике дослідження швидкості обробки кожним з алгоритмів системи великих розмірностей ($90601 \times 90601 \approx 13$Mb у серіалізованому вигляді). Для кожного алгоритму проводилась серія запусків для 5 матриць однакової розмірності, вказаної вище, згенерованих солвером на першому кроці по часу для моделей з різним набором параметрів і записаних у дамп-файл. Абсолютна точність наближення задавалась порядку 10^{-6}, на противагу значенню за замовчуванням для даного модуля: 10^{-4}; початкове наближення явно не задавалось, так що за замовчуванням алгоритм стартував зі значення $\vec{0}$. Час виконання був усереднений для кожного з алгоритмів і наведений у таблиці нижче.

Algorithm	Time of estimation, sec
GMRES	1.77282
QMR	1.83367
BiCGSTAB	0.97049
LGMRES	1.72245

Неважко, бачити, что суттево швидшим ніж інші є BiCGSTAB (Biconjugate Gradient Stabilised Algorithm), отже в якості солвера для нашої системи був обраний саме він.

Зробимо невелике зауваження щодо класу пропонованих алгоритмів. Всі вони детально розглянуті у [6]. Клас даних алгоритмів називається *проекційними алгоритмами на підпростір Крилова*, або просто *алгоритмами криловського типу*. Усі проекційні алгоритми спираються на ключову ідею, що чергове наближення для розв'язку лінійної системи $\mathbf{Ax} = \mathbf{b}$ може бути шукане в деякому підпросторі $\mathbb{K}_m$ розмірності m основного простору $\mathbb{R}^n$, який називається *subspace of candidate approximants* і для забезпечення цього має бути накладено m умов, що де-факто є умовами ортогональності шуканого наближення (якщо говорити більш точно, то чергової нев'язки, а не власне самого наближення) до m базисних векторів іншого підпростору $\mathbb{L}_m$, який називають *subspace of constraints*. Усі алгоритми криловського типу мають важливу спільну особливість: в якості $\mathbb{K}_m$ на кожному кроці m обирається підпростір Крилова $\mathbb{K}_m(\mathbf{A},\vec{v}) = span(\vec{v}, \mathbf{A}\vec{v}, \ldots, \mathbf{A}^{m-1}\vec{v})$ для деякого вектора $\vec{v}$, який найчастіше покладають рівним $\vec{r}_0$ — початковій нев'язці; тут $\mathbf{A}$ — матриця системи, що розв'язується. Уся різноманітність алгоритмів криловського типу полягає у різних методах генерації базису чергової лінійної оболонки та відповідних апріорних припущеннях щодо вигляду матриці $\mathbf{A}$. Чудовий опис алгоритму побудови біортогональних систем базисів для двох криловських підпросторів, один з яких спирається на саму матрицю $\mathbf{A}$, а інший на $\mathbf{A}^T$, і який є ключовим для обраного нами алгоритму BiCGSTAB, можна знайти у оригінальній роботі автора даного алгоритму [4].

5 Аналіз отриманих результатів

Запуски програмної реалізації проводились здебільшого на сітках 101×101 та 201×201 для квадратних областей $[0,1] \times [0,1]$ та $[0,10] \times [0,10]$ з однаковими просторовими кроками 0,01 та 0,1 відповідно. Моделювання проводилось для 50 кроків по часу. Для $\lambda = 0,5, \lambda = 0,75$ обирався крок по часу $\Delta t = 0,01$, для $\lambda = 0,25$ (0 — явна схема) заради забезпечення стійкості крок по часу обирався меншим, а саме у випадку $\Delta x = \Delta y = 0,01$ покладали $\Delta t = 0,0001$ і проводили моделювання для 100 кроків по часу. В якості $U(\vec{x})$ — функції початкових умов була обрана гаусіана

$$2\left[exp\left\{-\frac{(x-0,5)^2}{2}\right\} + exp\left\{-\frac{(y-0,5)^2}{2}\right\}\right]$$

для квадрату $[0,1]^2$ та

$$3\left[exp\left\{-\frac{(x-3)^2}{2 \times 1,5^2}\right\} + exp\left\{-\frac{(y-2,5)^2}{2 \times 1,5^2}\right\}\right]$$

для квадрату $[0,10]^2$. Графіки для цих двох випадків, представлені у формі heat-plot, наведено у **Додатку II**.

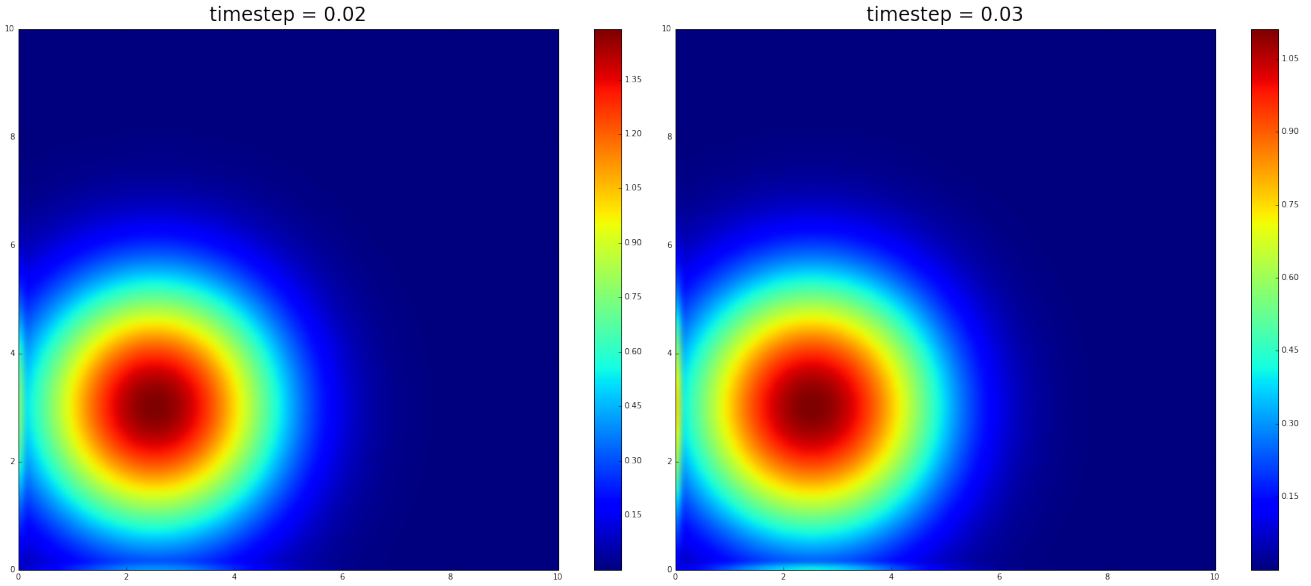
Крім того, було проаналізовано порівняльну поведінку точності розв'язків при однакових кроках дискретизації абсолютно явної схеми без додаткової лінеаризації, що її описано у підрозділі 2.3, та пропонованої неявної схеми при $\lambda = 0.5$ (0,75). З апріорних оцінок відомо, що неявна схема має по часу вищий порядок точності, ніж явна, проте виникло питання, наскільки швидко у явній схемі накопичуються похибки. Проводилось моделювання при $\Delta t = 0,001$ $\Delta x = \Delta y = 0,1$ на $[0,10]^2$ для 5 кроків по часу. Порівняння проводилось шляхом оцінки норми Фробеніуса різниці матриць, що представляють розв'язок, для кожного кроку по часу. Задавалась точність, з якою ці матриці повинні співпадати. У таблиці нижче наведено результати порівнянь.

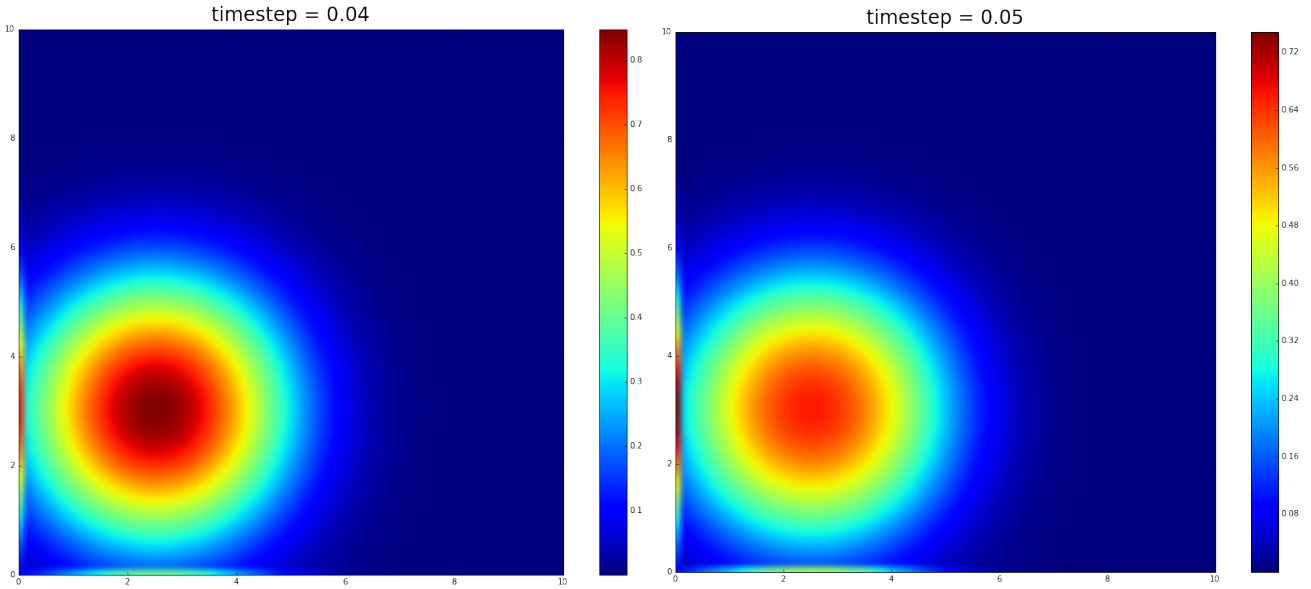
$\epsilon = 0.001$	Is equal?
1	True
2	True
3	False
4	False
5	False

Як бачимо, вже після другого кроку по часу результати починають відрізнятись більше ніж на $0,001 = \Delta t$, тобто накопичення похибок у явній схемі відбувається катастрофічно швидко.

Ще одним суттєвим результатом є той факт, що в нелінійних моделях умова стійкості фон Неймана є лише необхідною, але не достатньою, внаслідок залежності коеф-ту підсилення від отриманого на поточному кроці результату розв'язання. На графіках нижче

наведено результат моделювання для 5 кроків по часу на $[0{,}10]^2$ при тій же початковій ф-ї, що вказана вище і при таких значеннях параметрів моделі: $\alpha = -10$, $\beta = -1$, $m_0 = 2$, $\sigma = -0{,}01$, $\lambda = 0{,}75$, $\Delta t = 0{,}01$, $\Delta x = \Delta y = 0{,}1$. При цій комбінації параметрів умова фон Неймана для (6) виконується для всіх 5-ти кроків, при цьому на границі, де ф-я початкових умов, як і функція розв'язку швидко змінює характер росту і стрімко спадає, ми можемо бачити прояви нестійкості коливного харакеру.





Додаток I

Listing 1: class CNModel

```
import numpy as np
import types
from utils import cast_init_to_fun


class CNModel(object):

    def __init__(self, lambd, dt, dx, dy, T, M, N, sigma=1., alpha=1.,
                 beta=1., m_0=1., init_cond=None):
        self.lambd = lambd
        self.dt = float(dt)
        self.dx = float(dx)
        self.dy = float(dy)
        if (type(T) is not int or type(M) is not int or type(N) is not int):
            raise ValueError("Nodes number (T,M,N): int expected")
        self.T = T
        self.M = M
        self.N = N
        self.domain = (self.dt * self.T, self.dx * (self.M - 1), self.dy * (self.N
            - 1))
        print "Model on (0,%s) time interval, [0,%s]x[0,%s] plane created" %
            self.domain
        self.alpha = alpha
        self.beta = beta
        self.m_0 = m_0
        if callable(init_cond):
            self.init_cond = init_cond
        else:
            self.init_cond = cast_init_to_fun(init_cond)
        self.sigma = sigma

    def _get_footprint(self):
        return {'up': -self.lambd * self.S_x,
                'down': -self.lambd * self.S_x,
                'left': -self.lambd * self.S_y,
                'right': -self.lambd * self.S_y,
                'center': 1. - self.lambd * self.dt * self.m_0 * self.alpha + 2. *
                    self.lambd * self.S
                }
```

```
37
    def prepare(self):
39      self.n_0 = float(self.m_0) / self.beta
        self.S_x = (self.sigma * self.dt) / (self.dx)**2
41      self.S_y = (self.sigma * self.dt) / (self.dy)**2
        self.S = self.S_x + self.S_y
43      self.footprint = self._get_footprint()
        self.initial_state = self.generate_initial_state()
45
    def generate_initial_state(self):
47      init_grid = []
        for i in range(self.M):
49          for j in range(self.N):
                init_grid.append(self.init_cond([i*self.dx,j*self.dy]))
51      return np.asarray(init_grid, dtype=float)

53  def stability_factor(self,i,j):
        return 2.*(self.S_x*(1-np.cos(i*self.dx)) + self.S_y*(1-np.cos(j*self.dy)))
```

Listing 2: class CNSolver

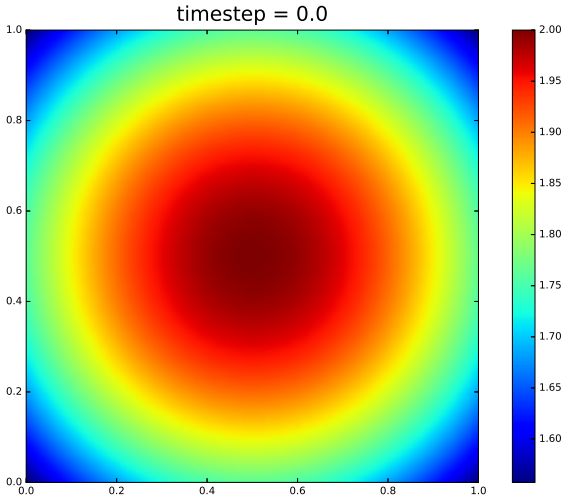
```
import scipy.sparse as sps
import numpy as np
from scipy.sparse.linalg.dsolve import linsolve
import  pickle

class CNSolver(object):
    def __init__(self):
        self.buff = None
        self.solutions = []

    def fit(self, mod):
        self.mod = mod
        self.mod.prepare()
        self.initial_state = self.mod.initial_state
        self.buff = self.initial_state
        self.solutions.append(self.buff)

    def _decompose_element_position(self, index):
        j = index % self.mod.M
        i = (index - j) / self.mod.M
        return i, j

    def is_stable(self, sol):
        l = self.mod.lambd
        dt = self.mod.dt
        a = self.mod.alpha
        b = self.mod.beta
        m = self.mod.m_0
        max_stable = 0
        for point in xrange(len(sol)):
            i, j = self._decompose_element_position(point)
            stable = (1.-(1.-l)*(dt*a*m + self.mod.stability_factor(i,j)) -
                (1-2.*l)*dt*a*b*sol[point])/\
                        (1.-l*(dt*m*a-2.*sol[point]*dt*a*b-self.mod.stability_factor(i,j
            if np.abs(stable)>max_stable:
                max_stable = np.abs(stable)
        if max_stable < 1:
            return True, max_stable
        else:
            return False, max_stable

    def _generate_LHS(self):
```

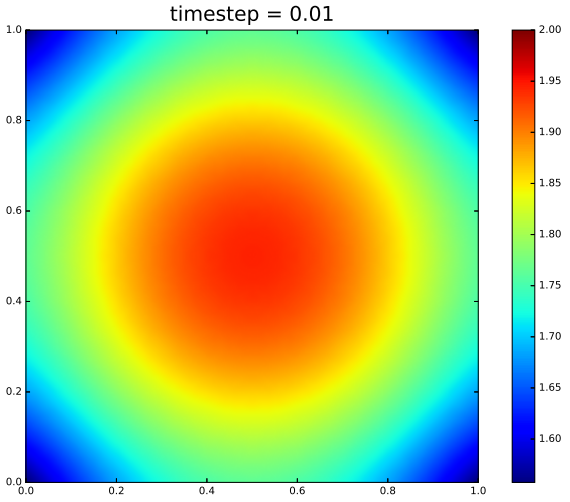
```
        print "Generating LHS"
        m = self.mod.M
        n = self.mod.N
        num_of_eqs = m * n
        LHS = sps.lil_matrix((num_of_eqs, num_of_eqs), dtype=np.float64)
        for q in xrange(num_of_eqs):
            i, j = self._decompose_element_position(q)
            if (i == 0 or i == (n - 1) or j == 0 or j == (m - 1)):
                LHS[q, q] = 1.
            else:
                LHS[q, q] = self.mod.footprint['center'] + 2. * self.buff[q] * \
                    self.mod.lambd * self.mod.dt * self.mod.alpha * self.mod.beta
                LHS[q, q - 1] = self.mod.footprint['left']
                LHS[q, q + 1] = self.mod.footprint['right']
                LHS[q, q - m] = self.mod.footprint['up']
                LHS[q, q + m] = self.mod.footprint['down']
        LHS = LHS.todia()
        return LHS

    def _generate_RHS(self):
        m = self.mod.M
        n = self.mod.N
        sx = self.mod.S_x
        sy = self.mod.S_y
        square_coef = self.mod.dt * self.mod.alpha * self.mod.beta
        center_coef = self.mod.dt*self.mod.alpha*self.mod.m_0 - 2. * self.mod.S
        rhs = []
        num_of_eqs = m * n
        for q in range(m):
            rhs.append(0.)

        for q in range(m, num_of_eqs - m):
            if q % m == 0 or q % m == m - 1:
                rhs.append(0.)
            else:
                rhs.append((center_coef - square_coef * self.buff[q]) *
                    self.buff[q] + sx * (
                     self.buff[q - 1] + self.buff[q + 1]) + sy * (self.buff[q - m]
                        + self.buff[q + m]))
        for q in range(num_of_eqs - m, num_of_eqs):
            rhs.append(0.)
        return rhs

    def solve(self):
```

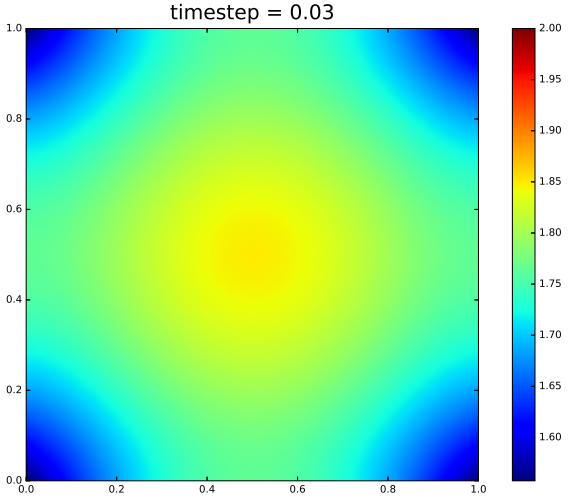
```
        debug = True
        sol = None
        for k in range(self.mod.T):
            print "Generating RHS  # %s" % k
            LHS = self._generate_LHS()
            RHS = self._generate_RHS()
            print "Solving % s system" % k
            sol = sps.linalg.bicgstab(LHS,RHS,tol='1e-6')[0]
            sol = sol + self.buff
            stable, coef_amplif = self.is_stable(sol)
            if not stable:
                print "No stability! Amplification coefficient = %s" % coef_amplif
                self.buff = self.initial_state
                break
            self.solutions.append(sol)
            self.buff = sol
        for i in range(len(self.solutions)):
            self.solutions[i] = self.solutions[i].reshape((self.mod.M, self.mod.N))
        self.buff = self.initial_state
```

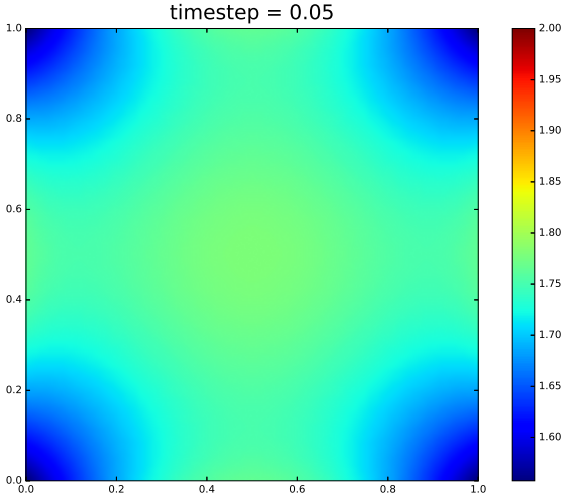
Додаток II

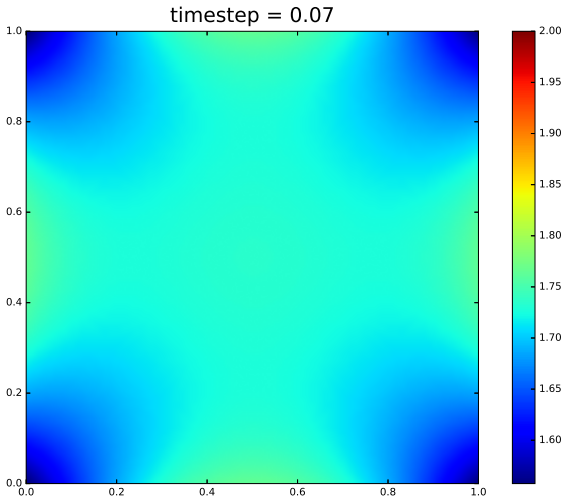
Нижче наведено графіки для $\lambda = 0{,}75$, $\Delta t = 0{,}01$, $\Delta x = \Delta y = 0{,}01$, $\alpha = 1$, $\beta = 1$, $\sigma = 1$

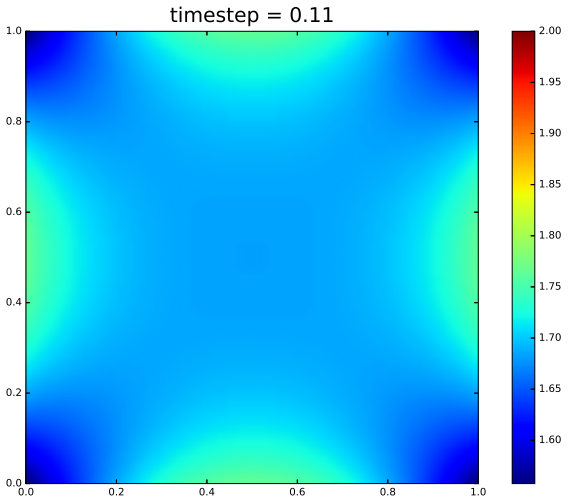


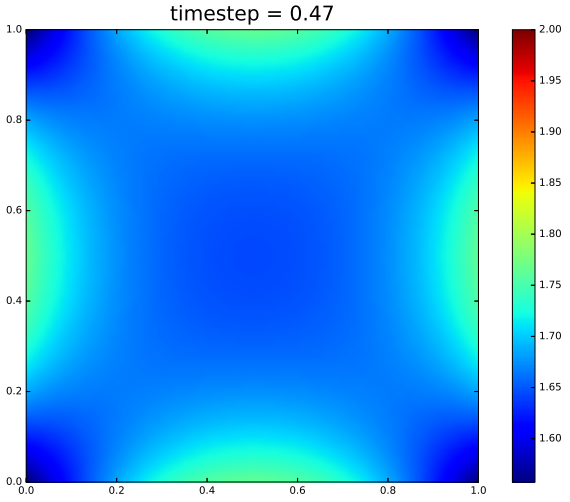


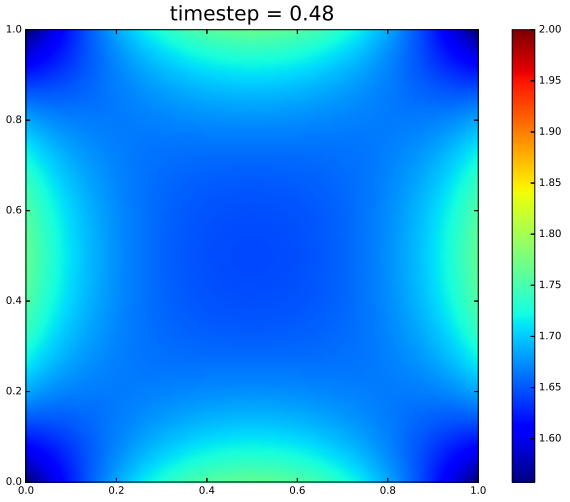




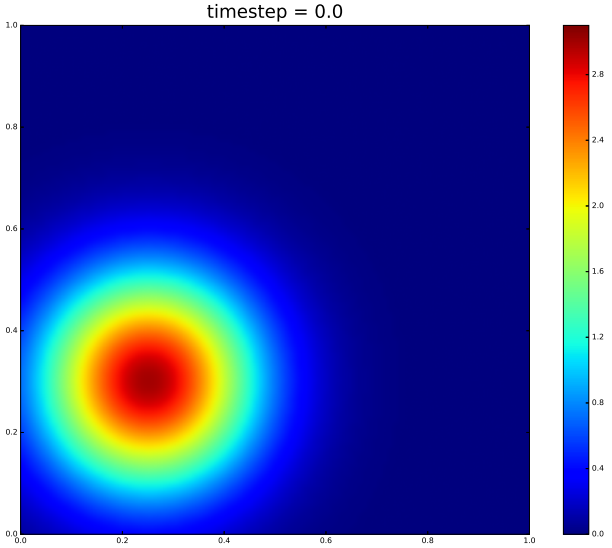

timestep = 0.07
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
2.00
1.95
1.90
1.85
1.80
1.75
1.70
1.65
1.60

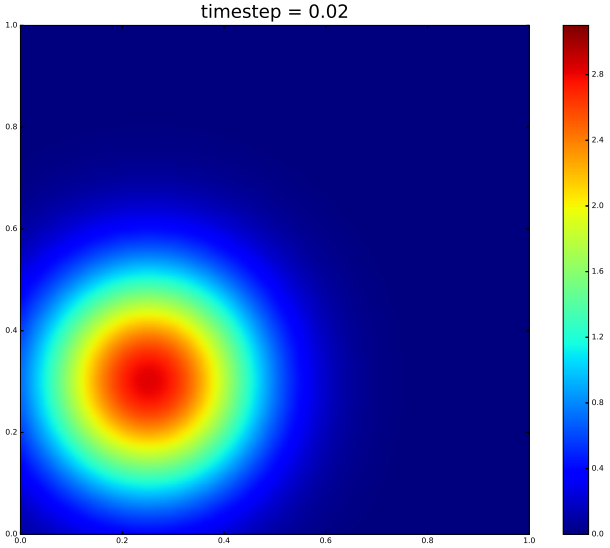

timestep = 0.11
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
2.00
1.95
1.90
1.85
1.80
1.75
1.70
1.65
1.60

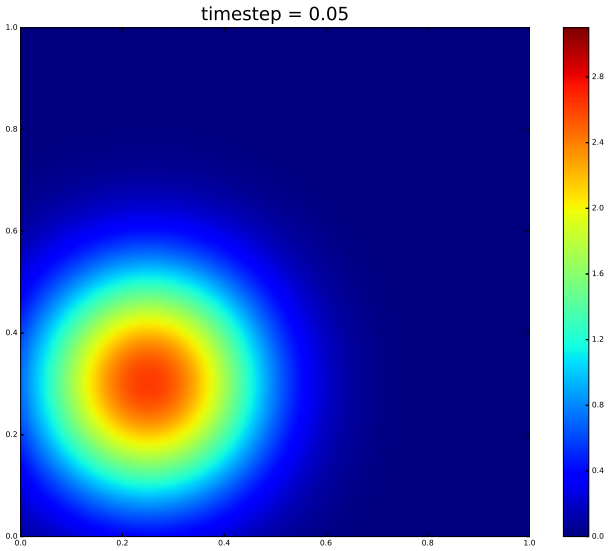

timestep = 0.47
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
2.00
1.95
1.90
1.85
1.80
1.75
1.70
1.65
1.60

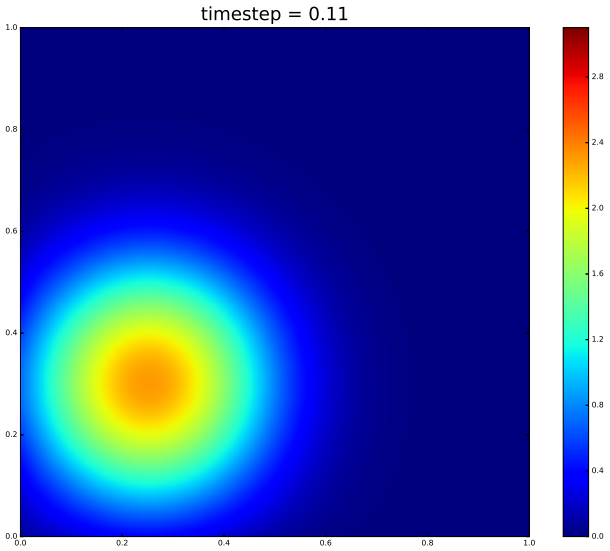

timestep = 0.48
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
2.00
1.95
1.90
1.85
1.80
1.75
1.70
1.65
1.60

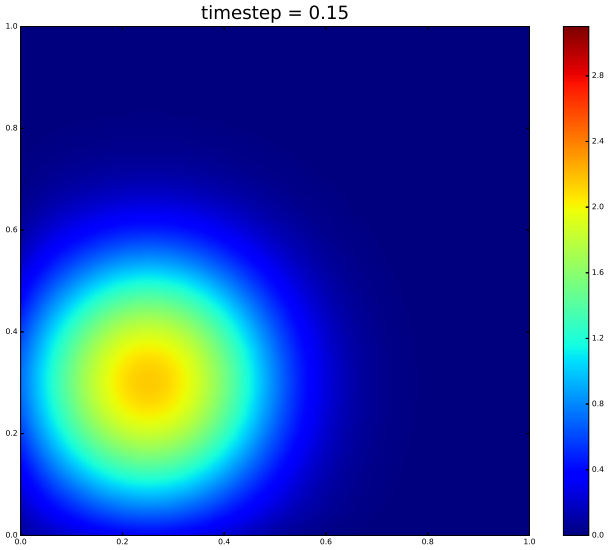
Нижче наведено графіки для $\lambda = 0{,}5$, $\Delta t = 0{,}01$, $\Delta x = \Delta y = 0{,}1$, $\alpha = 1$, $\beta = 1$, $\sigma = 1$

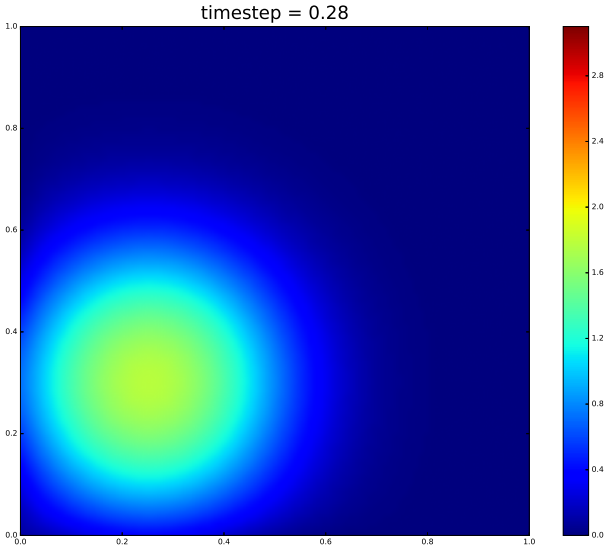


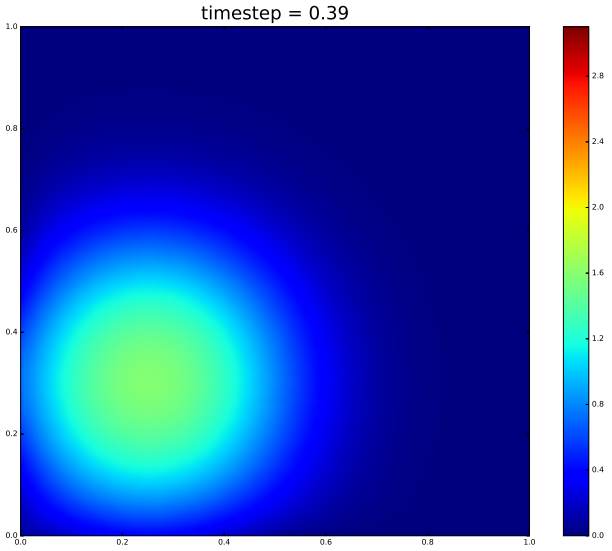


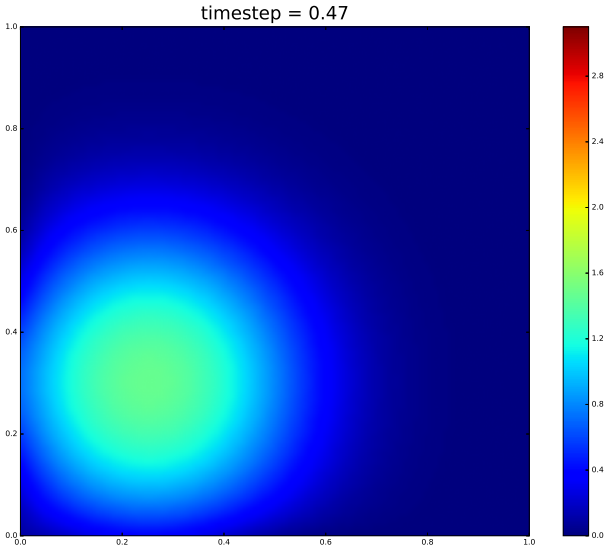





timestep = 0.15
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
2.8
2.4
2.0
1.6
1.2
0.8
0.4
0.0


timestep = 0.28
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
2.8
2.4
2.0
1.6
1.2
0.8
0.4
0.0


timestep = 0.39
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
2.8
2.4
2.0
1.6
1.2
0.8
0.4
0.0


timestep = 0.47
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
2.8
2.4
2.0
1.6
1.2
0.8
0.4
0.0

5 Список використаних джерел

1. Годунов С.К., Рябенький В.С. Разностные схемы. Введение в теорию. Изд.2, перераб. и доп. — Наука, 1977, 440 с.

2. Рихтмайер Р., Мортон К. Разностные методы решения краевых задач: Пер. с англ. — М.:Мир, 1972, 420 с.

3. R. A. Fisher. The wave of advance of advantageous genes, Ann. Eugenics 7:353–369, 1937

4. C. Lanczos, Solution of linear equations by minimized iterations, J. Res. Natl. Bur. Stand., 49 (1952), pp. 33–53.

5. B. Perthame, Parabolic Equations in Biology: Growth, reaction, movement and diffusion, Springer, 2015

6. Y. Saad, Iterative Methods for Linear Systems, SIAM, 2003.

7. L. N. Trefethen, K. Embree, The PDE Coffee Table Book, Unpublished, 2001