Київський національний університет
імені Тараса Шевченка
Факультет комп'ютерних наук та кібернетики
Кафедра моделювання складних систем

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 113 Прикладна математика

на тему:

Покращення методу чисельного наближення на основі методу Рунге-Кутта

Виконала студентка 4-го курсу
Злосчастьєва Діана Костянтинівна _____

Науковий керівник:
доцент кафедри моделювання
складних систем, **доктор** фіз.-мат. наук
Шатирко Андрій Володимирович _____

Студент _____

Роботу розглянуто й допущено до захисту на засіданні кафедри моделювання складних систем
«10» червня 2022 р., протокол № 18
Завідувач кафедри
Д.І. Черній _____

Київ — 2022

Зміст

Аннотація

Об'єктом дослідження роботи є лінійні та нелінійні задачі Коші, розв'язок яких я знаходитиму за допомогою чисельних методів; алгоритм методу Рунге-Кутта.

Мета роботи: знайти розв'язок за допомогою вже відомих методів та спробувати їх покращити. Заплановано порівняти різні методи та визначити, який доречніше використовувати для розв'язування однієї задачі

Актуальність роботи: ця робота буде дуже корисною для науково-учбового процесу, щоб скоротити час для розв'язування задач

Можливі сфери застосування: якщо описати будь-який фізичний, хімічний, технічний процес цього світу математичною мовою за допомогою рівнянь, то можна швидко знайти його наближений, але з малою похибкою, розв'язок.

Вступ

Чисельні методи - дуже цікавий, корисний і найбільш прикладний розділ математики. Чому саме він?

Добре, припустимо, що ми маємо будь-який процес цього світу: біологічний, фізичний, технічний. Спочатку ми намагаємось зрозуміти його суть з точки зору фізики процесу, виділяємо основні закономірності, формулюємо закони, яким він підкоряється, будуємо математичну модель, починаємо описувати його за допомогою формул у вигляді диференціальних, інтегральних рівнянь, систем і т.д. Практично будь-яке моделювання процесів закінчується чисельними методами. Наприклад, моделювання вихрових потоків в архітектурі або моделювання води в сучасних мультфільмах. Також у алгоритмах машинного зору. Але щоб зрозуміти цей процес, треба знайти розв'язок цих рівнянь. Майже всі рівняння, які ми отримуємо, в результаті будуть нелінійними, лише в окремих рідкісних випадках формули будуть виражені явно. Постає питання, як вирішувати ці нелінійні рівняння? Адже розв'язок має найбільш точно відповідати реальному процесу. Тому тут велику роль грають чисельні методи, за допомогою яких можна не тільки знайти наближений розв'язок задачі, але й визначити максимальну точність наближення, визначити стійкість методу до похибок початкової умови.

Точні методи вивчаються у курсах диференціальних рівнянь і дозволяють висловити рішення рівняння через елементарні функції або за допомогою квадратур від елементарних функцій.

Розділ I

Постановка задачі

Маємо таку постановку задачі [3].

Задача Коші:

$$\begin{cases} y'(t) = \varphi(t, y(t)) \quad \forall t \in I = [t_0, T] \\ y(t_0) = y_0 \end{cases}$$

Для $h > 0 : t_n = t_0 + nh \quad (n = 0, 1, ..., N), \quad (N + 1)$ вузлів, $h = \frac{T-t_0}{N}, \quad I_n = [t_n, t_{n+1}]$.

Всі методи вирішення задачі Коші для звичайних диференційних рівнянь діляться на точні, наближені та чисельні.

Точні методи вивчаються у курсах диференціальних рівнянь і дозволяють описати розв'язок рівняння через елементарні функції або за допомогою квадратур від елементарних функцій. Клас завдань, вирішення яких можна отримати точними методами порівняно вузький[5]:

- в деяких випадках розв'язок можливо виразити неявно. Наприклад, елементарні диференційні рівняння: $y'(t) = \frac{y(t)-t}{y(t)+t} \Rightarrow y$ виражений неявно $\Rightarrow \frac{1}{2}\, ln(t^2 + y^2(t)) + arctg(\frac{y(t)}{t})) = C$
- в інших випадках неможливо представити розв'язок навіть в неявній формі, це стосується, наприклад, рівняння: $y'(t) = e^{-t^2}$

У наближених методів розв'язок задачі Коші для звичайних диференційних рівнянь визначається як послідовність деяких функцій. В цій послідовності кожен елемент виражається через елементарні функції або квадратури від елементарних функцій. До наближених методів відносяться: розкладання рішення в узагальнений статичний ряд, метод Чаплигіна, метод Пікара, Канторовича та інші.

Наближені методи зручно застосовувати тоді, коли вдається знайти явний вираз для низки коефіцієнтів.

У всіх інших випадках доречно застосовувати чисельні методи. Алгоритм знаходження наближеного розв'язку такий:

Для кожного вузла t_n шукаємо невідоме значення u_n, яке наближує до точного значення $y_n = y(t_n)$

- набір зі $N + 1$ значень $\quad \{t_0, t_1 = t_0 + h, ..., t_n = T\}$ - це точки дискретизації
- набір зі $N + 1$ значень $\quad \{y_0, ..., y_n\}$ - точний дискретний розв'язок
- набір зі $N + 1$ значень $\quad \{u_0 = y_0, u_1, ..., u_n\}$ - чисельний розв'язок

Багатокрокові методи

Схеми[1], які ми будемо будувати, дозволяють в явній на неявній формі обчислити u_{n+1} знаючи значення $u_n, u_{n-1}, ..., u_{n-k}$ і, таким чином, можна послідовно обрахувати $u_1, u_2, ...$, починаючи з u_0 за рекурентною формулою, що має вигляд:

$$\begin{cases} u_0 = y_0 \\ ... \\ u_k = y_k \\ u_{n+1} = \Phi(u_{n+1}, u_n, \dots, u_{n-k} \quad \forall n = k, k+1, \dots, N-1 \end{cases}$$

Точніше, ми будемо розглядати схеми з $k = p + 1$ лінійними кроками, формула яких має вигляд:

$$u_{n+1} = \sum_{j=0}^{p} a_j u_{n-j} + h \sum_{j=0}^{p} b_j \varphi(t_{n-j}, u_{n-j}) + h b_{-1} \varphi(t_{n+1}, u_{n+1})$$

$$\forall n = p, p+1, ..., N-1$$

де $\{a_k\}$ і $\{b_k\}$ - задані коефіцієнти, а $p > 0$ - ціле число

Явний метод - якщо значення u_{n+1} знаходиться безпосередньо з u_k, для лінійної системи це позначається як $b_{-1} = 0$

Неявний метод - якщо значення u_{n+1} визначається тільки неявним співвідношенням. Для лінійної системи це еквівалентно $b_{-1} \neq 0$

Одноетапний метод - якщо для $\forall n \in N$: u_{n+1} залежить тільки від одиниці, і, можливо, від самого себе.

Багатоетапний метод - всі інші випадки

Методи Рунге-Кутта

Маємо задачу Коші:

$$\begin{cases} y'(t) = \varphi(t, y(t)) \quad \forall t \in I = [t_0, T] \\ y(t_0) = y_0 \end{cases}$$

Схеми Рунге-Кутта апроксимують інтеграл [8][10]

$$\int_{t_n}^{t_{n+1}} \varphi(t, y(t))\, dt$$

за квадратурною формулою в межах $[t_n, t_{n+1}]$

$$t_{n,i} = t_n + C_i h$$

$$int_{t_n}^{t_{n+1}} \varphi(t, y(t))\, dt = h \sum_{j=0}^{p} a_j u_{n-j}$$

Проблема полягає в тому, що якщо $c_i \neq 0$ і $c_i \neq 1$, то тоді точка $t_{n,i}$ не є точкою дискретизації та $y(t_{n,i})$ - невідоме.

Оцінка $\varphi(t_{n,i}, y(t_{n,i})$ відбувається у внутрішніх точках, а потім апроксимується іншою квадратурною формулою:

$$\varphi(t_{n,i}, y(t_{n,i})) \approx \varphi(t_{n,i}, y_n + h \sum_{j=1}^{s} a_{i,j} \varphi(t_{n,j}, y(t_{n,j})))$$

Метод Рунге-Кутта з $s \leq 1$ кроками записується таким чином:

$$\begin{cases} u_0 = y(t_0) = y_0 \\ u_{n+1} = u_n + h \sum_{j=1}^{s} b_i K_i \quad n = 0, 1, ..., N-1 \\ K_i = \varphi(t_n + hc_i, u_n + h \sum_{j=1}^{s} a_{i,j} K_j \quad s = 1, \ldots, s \end{cases}$$

Матриця Батчера

Коефіцієнти зазвичай складають два вектори [2]:

$$b = (b_1, b_2, \ldots, b_s)^\top$$
$$c = (c_1, c_2, \ldots, c_s)^\top$$

та матриця $A = (a_{ij})_{1 \leq j, j \leq s}$.
О з н а ч е н н я
Таблиця

$\mathbf{c}$	$\mathbf{A}$
	$\mathbf{b}^\top$

називається матрицею Батчера.

- якщо $a_{ij} = 0$ для $j \geq i$ (тобто матриця A - нижньотрикутна) - тоді метод явний, бо кожний K_i обчислюється на границі.

- у всіх інших випадках метод неявний і потрібно вирішити нелінійну систему розмірності s для обчислення K_i:

  – якщо $a_{ij} = 0$ для $j > i$ (A – нижньотрикутна матриця), тоді метод називається напівнеявний або діагонально неявний, тобто кожний K_i є розв'язком нелінійного рівняння:

$$\boxed{\mathrm{K}_i} = \psi\left(t_n + c_i h, u_n + h a_{ii}\boxed{\mathrm{K}_i} + h\sum_{j=1}^{i-1} a_{ij} K_j\right)$$

Таким чином напівнеявна схема передбачає розв'язок s незалежних лінійних рівнянь. Якщо при цьому всі діагональні члени рівні $a_{ij} = \gamma$: $\forall i = 1 \dots s$, то такий метод називають одно-діагональний неявний метод.

Порядок збіжності

Нехай ω - порядок збіжності метода [3].

О з н а ч е н н я

Послідовний метод Рунге-Кутта, якщо:

$$\begin{cases} \sum_{j=1}^{s} b_i = 1 \\ c_i = \sum_{j=1}^{s} a_{ij} \quad i = 1, \dots, s \end{cases}$$

З а у в а ж е н н я: для явного методу у нас буде $c_1 = a_{1,j} = 0$, тому $K_1 = \varphi(t_n, u_n)$ і $c_2 = a_{21}$, тому $K_2 = \varphi(t_n + c_2 h, u_n + h c_2 K_1)$.

Тоді:

Якщо $\sum_{j=1}^{s} b_j c_j = \frac{1}{2}$, тоді $\omega \geq 2$

Якщо $\begin{cases} \sum_{j=1}^{s} b_j c_j^2 = \frac{1}{3} \\ \sum_{i=1}^{s}\sum_{j=1}^{s} b_i a_{ij} c_j = \frac{1}{6} \end{cases}$ тоді $\omega \geq 3$

Якщо $\begin{cases} \sum_{j=1}^{s} b_j c_j^3 = \frac{1}{4} \\ \sum_{i=1}^{s}\sum_{j=1}^{s} b_i c_i a_{ij} c_j = \frac{1}{8} \\ \sum_{i=1}^{s}\sum_{j=1}^{s} b_i a_{ij} c_j^2 = \frac{1}{12} \\ \sum_{i=1}^{s}\sum_{j=1}^{s}\sum_{k=1}^{s} b_i a_{ij} a_{jk} c_k = \frac{1}{24} \end{cases}$ тоді $\omega \geq 4$

Т е о р е м а

Нехай існує s-кроковий метод Рунге-Кутта порядку ω.

- якщо метод явний, тоді $\omega \leq s$

- якщо метод неявний $\omega \leq 2s$
- якщо $s \geq 5$, тоді ω

A-стійкість

О з н а ч е н н я

A-стійка система [2], якщо:

Для$\forall \lambda \in \mathbb{C}$ с $\Re(\lambda) = -\beta$, де β — додатне дійсне число, розглянемо задачу Коші: $\begin{cases} y'(t) = \lambda y(t), t > 0, \\ y(0) = y_0 \end{cases}$ де $y_0 \neq 0$ — задане значення.

Його розв'язок: $y(t) = y_0 e^{\lambda t}$ тоді $\lim_{t \to +\infty} y(t) = 0$.

Нехай $h > 0$ заданий крок часу, $t_n = nh$ для $n \in \mathbb{N}$ $u_n \approx y(t_n)$ - наближення розв'язку y до часу t_n.

Якщо при можливих умовах на h, відбувається

$$\lim_{n \to +\infty} u_n = 0$$

тоді кажуть, що схема A-стійка.

У випадку, якщо $\lambda \in \Re^-$, тоді метод Рунге-Кутта записується з $s \geq 1$ ітераціями для $y^{'}(t) = -\beta y(t)$ наступним чином:

$$\begin{cases} u_0 = y_0, \\ u_{n+1} = y_n + h \sum\limits_{i=1}^{s} b_i K_i, \qquad n = 0, 1, \dots N-1 \\ K_i = -\beta \left(u_n + h \sum\limits_{j=1}^{s} a_{ij} K_j \right) \qquad i = 1, \dots s \\ \end{cases}$$

Коректно поставлені задачі (математично та чисельно)

Як відомо, що в цілому для того, щоб числова схема була збіжною, недостатньо того, щоб вона давала коректні результати з будь-якого диференціального рівняння. Проблема має бути [5]:

- математично коректно поставлена (наявність та унікальність рішення),
- чисельно коректно поставлена (мала похибка відносно початкових умов)
- має доступний час розрахунку

Математично коректно поставлена проблема

Задача Коші вважається математично коректно поставленим, якщо існує один і тільки один розв'язок.

Якщо це не так, то кажуть, що задача математично некоректно поставлена.

П р и к л а д математично некоректно поставленої задачі:

Наприклад, ми шукаємо функцію $yt \in \mathbb{R}^+ \mapsto y(t) \in \mathbb{R}$, що задовольняє:

$$\begin{cases} y'(t) = \sqrt[3]{y(t)}, & \forall t > 0, \\ y(0) = 0. \end{cases}$$

Легко перевірити, що для будь-якого $t \geq 0$, усі три функції

- $y_1(t) = 0$,
- $y_{2,3}(t) = \pm\sqrt{8\frac{t^3}{27}}$

є розв'язками цієї задачі Коші.

При цьому при використанні чисельного методу ми не знаємо, під який розв'зок підходить ця схема і різні схеми можуть підходити до різних рішень.

Чисельно коректно поставлена задача

Після знаходження чисельного розв'язку $\{u_n\}_{n=1}^N$ математично поставленої задачі Коші слід зауважити, наскільки мала помилка $|y(t_n) - u_n|$. Це залежить, звичайно, від обраної схеми, але також від проблеми, що розглядається.

Оскільки помилки округлення завжди призводять до похибки розв'язку, важливо знати, чи близький розв'язок збуреної проблеми до розв'язку незбуреної проблеми.

Кажуть, що задача Коші чисельно коректно поставлена, якщо розв'язання слабо порушеної задачі (друга кінцівка чи початковий стан) має розв'язок, близький до розв'язку вихідної задачі.

П р и к л а д чисельно некоректно поставленої задачі:

Нехай $\varphi(t, y) = 3t - 3y$ і $y(0) = \alpha$ (будь-яке число). Ми шукаємо функцію $yt \in \mathbb{R} \mapsto y(t) \in \mathbb{R}$, яка задовольняє

$$\begin{cases} y'(t) = 3y(t) - 3t, & \forall t \in \mathbb{R}, \\ y(0) = \alpha. \end{cases}$$

Його розв'язок, заданий в $\mathbb{R}$, має вигляд

$$y(t) = \left(\alpha - \frac{1}{3}\right) e^{3t} + t + \frac{1}{3}.$$

Порахуємо y у $t = 10$:

- якщо $\alpha = 1/3$, то $y(10) = \frac{31}{3}$,
- якщо $\alpha = 0.333333$, то $y(10) = (0.333333 - 1/3)e^{30} + 10 + 1/3 = -e^{30}/3000000 + 31/331/3 + 10^7/3$

Якщо спробуємо знайти розв'язок задачі Коші до t =10 з $\alpha = 1/3$, ми отримаємо $y(10)$ =31/3. З іншого боку, якщо ми зробимо розрахунок з наближенням $\alpha = 0.333333$ замість $1/3$, ми отримаємо $y(10) = 31/3 - e^{30}/30000000$ який є різницею з попереднім значенням $e^{30}/300000010^7/3$.

Цей приклад говорить нам, що невелика похибка на початковій умові (відносна похибка порядку 10^{-6}) може викликати дуже велику похибку на $y(10)$ (відносна похибка близько 10^6). Таким чином, якщо калькулятор обчислює розв'язок тільки зі значущими цифрами 6, то $\alpha = 1/3$ стає $\alpha = 0.333333$ і марно намагатися винайти чисельний метод обчислення $y(10)$. Справді, єдина помилка на початковій умові вже спричиняє неприпустиму помилку у розв'язку. Тут ми маємо справу з чисельно некоректно поставленою проблемою.

П р и к л а д чисельно коректно поставленої задачі

Розглянемо задачу Коші

$$\begin{cases} y'(t) = -y(t), \quad \forall t > 0, \\ y(0) = y_0 + \varepsilon. \end{cases}$$

Розв'язок $y(t) = y_0 e^{-t} + \varepsilon e^{-t}$: ефект збурення ε зменшується при $t \to +\infty$ з $\varepsilon e^{-t} \xrightarrow[t \to +\infty]{} 0$. Це говорить про те, що якщо помилка зроблена на етапі ітераційного методу, ефект цієї помилки з часом зменшується: задача чисельно коректно поставлена.

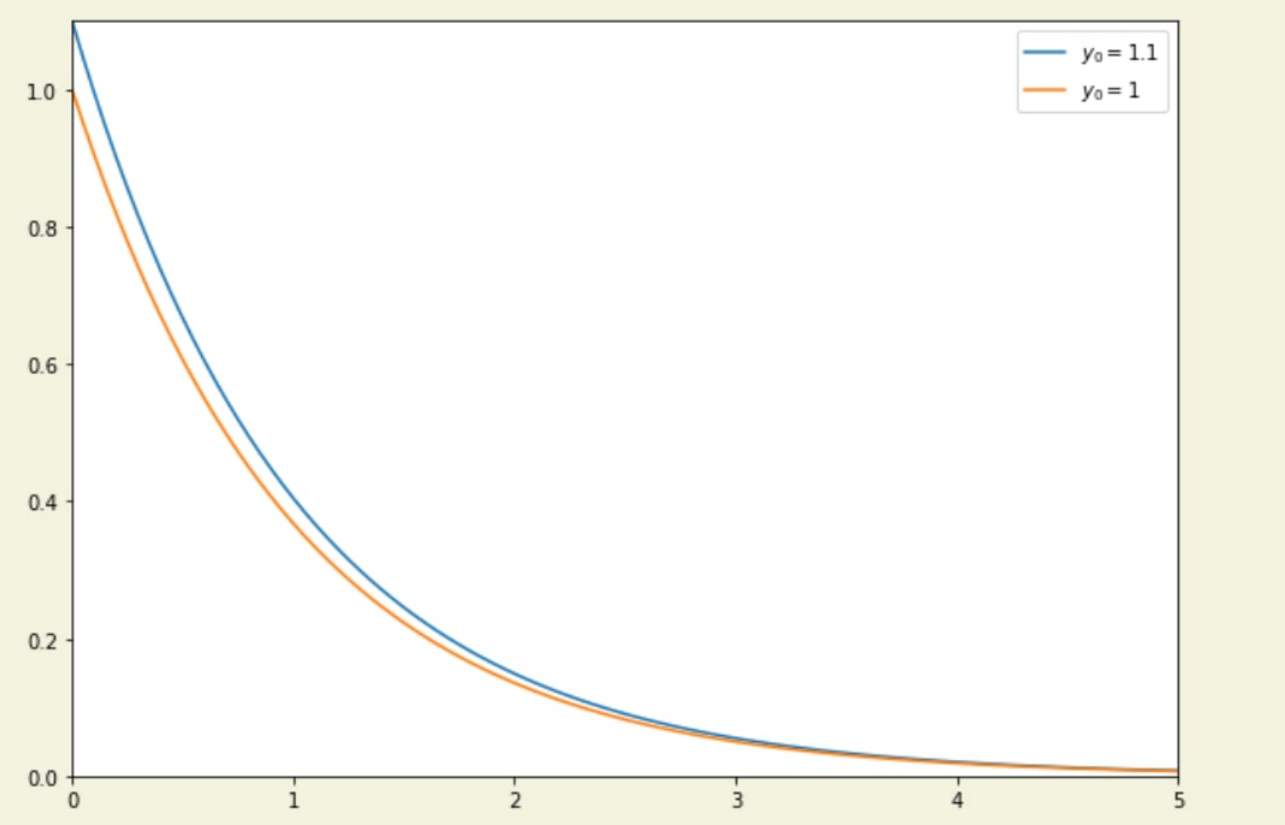


Рис. 1: Результат чисельно коректно поставленої задачі

Коректно обумовлена задача

Задача Коші вважається коректно обумовленою, якщо звичайні чисельні методи можуть знайти її розв'зок за розумну кількість операцій.

П р и к л а д некоректно (жорстко) обумовленої задачі:

$\varphi(t, y) = -\beta y$ та $y(0) = 1$.

Ми шукаємо функцію $yt \in \mathbb{R} \mapsto y(t) \in \mathbb{R}$, що задовольняє

$$\begin{cases} y'(t) = -\beta y(t), & \forall t \in \mathbb{R}, \\ y(0) = 1. \end{cases}$$

Ця задача

- математично коректно поставлена для будь-якого $\beta \in \mathbb{R}$: є один і тільки один розв'язок, він визначається на $\mathbb{R}$ і задається $y(t) = e^{-\beta t}$;
- чисельно коректно поставлена для будь-якого $\beta > 0$: порядок похибки розв'язку буде меншим за порядок похибки вихідної умови;
- некоректно обумовлена або жорстка: якщо ми намагаємося вирішити задачу Коші, коли β дуже великий, ми повинні зробити крок h дуже маленьким, і якою б не була обрана схема, розв'язок стає все більш жорстким.

Розділ II

Побудова неявних методів Рунге-Кутта

Для побудови неявних методів Рунге-Кутта [2], ми спочатку повинні вибрати квадратурну формулу порядку p. Наступна лема є основним результатом гаусівського інтегрування.

Л е м м а

Нехай $c_1, ..., c_s$ дійсні та різні, і нехай $b_1, ..., b_s$ визначаються умовою $B(s)$ (тобто формула є "інтерполяційною"). Тоді ця квадратурна формула має порядок $2s-k$, тоді і тільки тоді коли многочлен $M(x) = (x-c_1)(x-c_2)...(x-c_s)$ ортогональний всім многочленам ступеня$<= s-k-1$, тобто тоді і тільки тоді коли:

$M(x) = C(P_s(x) + \alpha_1 P_{s-1}(x) + ... + \alpha_k P_{s-k}(x))$

Звідси бачимо, що це квадратурні формули порядку $2s-k$ може бути задані у термінах k параметрів $\alpha_1, \alpha_2, ..., \alpha_k$

Далі можна обчислити матрицю W і нарешті, вибирається матриця X і тоді метод Рунге-Кутта з коефіцієнтами $A = WXW^{-1}$ має порядок не нижчий за $min(n+c+1, 2n+2)$

П р и к л а д

Ми шукаємо всі неявні методи Рунге-Кутта, які задовольняють $B(2s-2), C(s-1)$ і $D(s-2)$, тобто методи, які мають порядок принаймні $2s-2$. Ставимо

$M(x) = C(P_s(x) + \alpha_1 P_{s-1}(x) + \alpha_2 P_{s-2}(x))$

Якщо α_2 задовольняє:

$\alpha_2 < \frac{s-1}{s}\frac{\sqrt{2s+1}}{\sqrt{2s-3}}$

тоді корені M є дійсними та різними. Матриця W має властивість $T(s-1, s-2)$. Нарешті ставимо:

$$X = \begin{pmatrix} 1/2 & -\xi_1 & & & \\ \xi_1 & 0 & \ddots & & \\ & \ddots & \ddots & -\xi_{s-2} & \\ & & \xi_{s-2} & 0 & \beta_{s-1} \\ & & & \xi_{s-1} & \beta_s \end{pmatrix}$$

і отримаємо з $A = WXW^{-1}$ сімейство неявних методів Рунге Кутта порядку $2s-2$ з чотирма параметрами $\alpha_1, \alpha_2, \beta_s, \beta_{s-1}$

Усі методи таблиці є особливими випадками. Відповідні значення параметрів наведені на рис.2

Method	α_1	α_2	β_s	β_{s-1}
Gauss	0	0	0	$-\xi_{s-1}$
Radau IA	$\sqrt{2s+1}/\sqrt{2s-1}$	0	$1/(4s-2)$	$-\xi_{s-1}$
Radau IIA	$-\sqrt{2s+1}/\sqrt{2s-1}$	0	$1/(4s-2)$	$-\xi_{s-1}$
Lobatto IIIA	0	$-\sqrt{2s+1}/\sqrt{2s-3}$	0	0
Lobatto IIIC	0	$-\sqrt{2s+1}/\sqrt{2s-3}$	$1/(2s-2)$	$-\xi_{s-1}(2s-1)/(s-1)$

Рис. 2: Значення параметрів для особливих випадків

Якщо покласти $\alpha_1 = 0$ та $\alpha_2 = -\frac{\sqrt{2s+1}}{\sqrt{2s-3}}$ (квадратура Лобатто), отримуємо двопараметричне сімейство Чіпмена (1976).

Побудова напів-явних (діагональних) методів Рунге-Кутта

Неявний метод з повною матрицею sxs вимагає одночасного розв'язання ns неявних (загалом нелінійних) рівнянь на кожному кроці часу [2].

Один із спосібів обійти цю складність - використовувати нижню трикутну матрицю (a_{ij}) :

$$\begin{bmatrix} a_{11} & 0 & 0 & ... & 0 \\ a_{21} & a_{22} & 0 & ... & 0 \\ a_{s1} & a_{s2} & a_{s2} & ... & a_{ss} \end{bmatrix}$$

Тоді рівняння можна розв'язувати s- етапно з лише n-вимірною системою, яка розв'язується на кожному етапі. Такий метод називається діагонально неявний (DIRK).

Розв'язуючи n-вимірні системи за допомогою ітерацій типу Ньютона, вирішується лінійна система на кожному етапі з матрицею коефіцієнтів виду $I - ha_{ii}\partial f/\partial y$.

Якщо всі a_{ii} рівні, можна сподіватися повторно використовувати збережену LU-факторізацію. Коли ми хочемо підкреслити цю додаткову властивість для а DIRK, ми будемо називати його однодіагонально неявним (SDIRK).

Розглянемо схему SDIRK:

$$\begin{array}{c|cccc} c_1 & \gamma & & & \\ c_2 & a_{21} & \gamma & & \\ \vdots & \vdots & \vdots & \ddots & \\ c_s & a_{s1} & a_{s2} & \dots & \gamma \\ \hline & b_1 & b_2 & \dots & b_s \end{array}$$

з s кроками.

Умови порядку складаються з таких сум, як:

$\sum_{i,j,l} b_j a_{jk} a_{kl} = \frac{1}{6}$

Тепер у матриці A більше ненульових записів, ніж для явних методів, тому ця сума містить набагато більше коефіцієнтів, ніж раніше. Хитрість полягає в перенесенні до правої частини рівняння, яка набуває вигляду:

$\sum'_{i,j,l} b_j a_{jk} a_{kl} = \sum_{i,j,l} b_j (a_{jk} - \gamma\beta_{jk})(a_{kl} - \gamma\beta_{kl})$,
де β_{jk} позначає дельту Кронекера. Перемноживши, отримуємо:
$\sum'_{i,j,l} b_j a_{jk} a_{kl} = \sum_{i,j,l} b_j a_{jk} a_{kl} - \gamma(\sum_{j,l} b_j a_{jl} + \sum_{j,k} b_j a_{jk}) + \gamma^2 \sum_j b_j$
Для всіх сум праворуч вставляємо умови порядку і отримаємо:
$\sum'_{i,j,l} b_j a_{jk} a_{kl} = \frac{1}{6} - \gamma + \gamma^2$

Загальне правило полягає в тому, що з'являється змінний поліном γ, коефіцієнти якого є сумами $1/\gamma(u)$, де u проходить через усі дерева, отримані шляхом «замикання» однієї, двох, трьох і т. д. вершин t (за винятком кореня).

Отримані таким чином умови для четвертого порядку показані на рис. 3. Для $s = 2, p = 3 i s = 3, p = 4$ ці спрощені умови мають дуже мало ненульових доданків, і рівняння стають простими для вирішення.

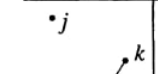
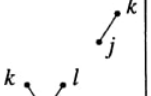
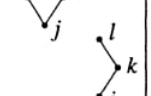
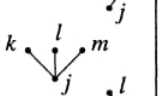
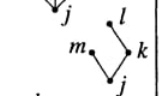
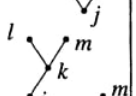
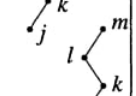
t	$\varrho(t)$	previous conditions	simplified conditions
	1	$\sum b_j = 1$	$\sum b_j = 1$
	2	$\sum b_j a_{jk} = \frac{1}{2}$	$\sum' b_j a_{jk} = \frac{1}{2} - \gamma$
	3	$\sum b_j a_{jk} a_{jl} = \frac{1}{3}$	$\sum' b_j a_{jk} a_{jl} = \frac{1}{3} - \gamma + \gamma^2$
	3	$\sum b_j a_{jk} a_{kl} = \frac{1}{6}$	$\sum' b_j a_{jk} a_{kl} = \frac{1}{6} - \gamma + \gamma^2$
	4	$\sum b_j a_{jk} a_{jl} a_{jm} = \frac{1}{4}$	$\sum' b_j a_{jk} a_{jl} a_{jm} = \frac{1}{4} - \gamma + \frac{3}{2}\gamma^2 - \gamma^3$
	4	$\sum b_j a_{jk} a_{kl} a_{jm} = \frac{1}{8}$	$\sum' b_j a_{jk} a_{kl} a_{jm} = \frac{1}{8} - \frac{5}{6}\gamma + \frac{3}{2}\gamma^2 - \gamma^3$
	4	$\sum b_j a_{jk} a_{kl} a_{km} = \frac{1}{12}$	$\sum' b_j a_{jk} a_{kl} a_{km} = \frac{1}{12} - \frac{2}{3}\gamma + \frac{3}{2}\gamma^2 - \gamma^3$
	4	$\sum b_j a_{jk} a_{kl} a_{lm} = \frac{1}{24}$	$\sum' b_j a_{jk} a_{kl} a_{lm} = \frac{1}{24} - \frac{1}{2}\gamma + \frac{3}{2}\gamma^2 - \gamma^3$

Рис. 3: Умови для четвертого порядку діагонального методу

Дуже точні методи SDIRK

Наша мета полягає в тому, щоб методи задовольнили
$a_{sj} = b_j j = 1, ..., s$
тобто в методах, для яких чисельний розв'язок y_1 ідентичний останній внутрішній ітерації [2].

Першим наслідком цієї властивості є те, що $R(\infty) = 0$

Умови порядку для таких методів замість можна ще спростити далі: розглянемо ще раз приклад, який тепер можна записати як:
$\sum_{j,k,l} a_{sj} a_{jk} a_{kl} = \frac{1}{6}$
Тепер ми маємо:
$\sum'_{j,k,l} a_{sj} a_{jk} a_{kl} = \sum_{j,k,l} (a_{sj} - \gamma\beta_{sj})(a_{jk} - \gamma\beta_{jk})(a_{kl} - \gamma\beta_{kl}) = \sum_{j,k,l} a_{sj} a_{jk} a_{kl} - \gamma(\sum_{j,k} a_{sj} a_{jk} + \sum_{j,l} a_{sj} a_{jl} + \sum_{k,l} a_{sk} a_{kl}) + \gamma^2(\sum_j a_{sj} + \sum_k a_{sk} + \sum_l a_{sl}) - \gamma^3$,
Знову підставивши відомі умови порядку, ми отримуємо:

$\sum'_{j,k,l} a_{sj}a_{jk}a_{kl} = \frac{1}{6} - \frac{3}{2}\gamma + 3\gamma^2 - \gamma^3$

Загальне правило схоже на наведене вище: різниця в тому, що всі вершини (включаючи корінь) тепер доступні для короткого замикання. Інший приклад, для дерева, який зображено на рис. 4 веде до наступної правої сторони:

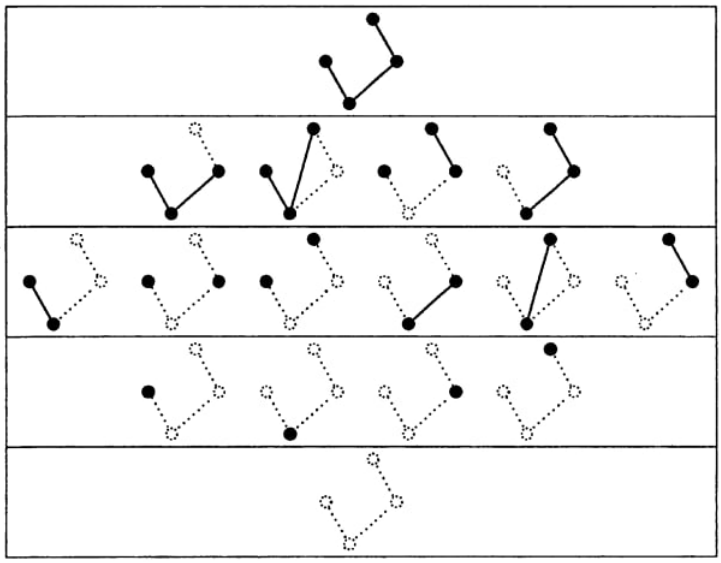


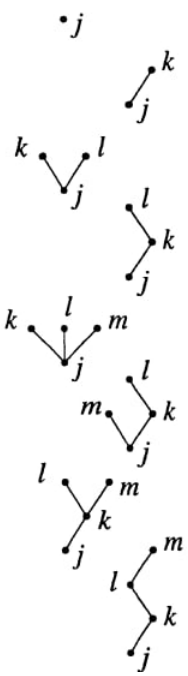
Рис. 4: Дерево короткого замикання

$\frac{1}{8} - \gamma(\frac{1}{3} + \frac{1}{3} + 1 \cdot \frac{1}{2} + \frac{1}{6}) + \gamma^2(\frac{1}{2} + 1 \cdot 1 + 1 \cdot 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{2}) - \gamma^3(1+1+1+1) + \gamma^4 = \frac{1}{8} - \frac{4}{3}\gamma + 4\gamma^2 - 4\gamma^3 + \gamma^4$

Отримані таким чином умови порядку зображені на рис. 5 для всіх дерев з порядком <= 4. Вирази $\sum'$ записуються явно для методу SDIRK з $s = 5$, що задовольняє умову:

	γ				
	a_{21}	γ			
	a_{31}	a_{32}	γ		
	a_{41}	a_{42}	a_{43}	γ	
	b_1	b_2	b_3	b_4	γ
	b_1	b_2	b_3	b_4	γ

$$c'_2 = a_{21}$$
$$c'_3 = a_{31} + a_{32}$$
$$c'_4 = a_{41} + a_{42} + a_{43}$$



$$\sum{}' a_{sj} = b_1 + b_2 + b_3 + b_4 = p_1$$

$$\sum{}' a_{sj}a_{jk} = b_2c_2' + b_3c_3' + b_4c_4' = p_2$$

$$\sum{}' a_{sj}a_{jk}a_{jl} = b_2c_2'^2 + b_3c_3'^2 + b_4c_4'^2 = p_3$$

$$\sum{}' a_{sj}a_{jk}a_{kl} = b_3a_{32}c_2' + b_4(a_{42}c_2' + a_{43}c_3') = p_4$$

$$\sum{}' a_{sj}a_{jk}a_{jl}a_{jm} = b_2c_2'^3 + b_3c_3'^3 + b_4c_4'^3 = p_5$$

$$\sum{}' a_{sj}a_{jk}a_{jl}a_{lm} = b_3c_3'a_{32}c_2' + b_4c_4'(a_{42}c_2' + a_{43}c_3') = p_6$$

$$\sum{}' a_{sj}a_{jk}a_{kl}a_{km} = b_3a_{32}c_2'^2 + b_4(a_{42}c_2'^2 + a_{43}c_3'^2) = p_7$$

$$\sum{}' a_{sj}a_{jk}a_{kl}a_{lm} = b_4a_{43}a_{32}c_2' = p_8$$

$$p_1 = 1 - \gamma$$

$$p_2 = \frac{1}{2} - 2\gamma + \gamma^2$$

$$p_3 = \frac{1}{3} - 2\gamma + 3\gamma^2 - \gamma^3$$

$$p_4 = \frac{1}{6} - \frac{3}{2}\gamma + 3\gamma^2 - \gamma^3$$

$$p_5 = \frac{1}{4} - 2\gamma + \frac{9}{2}\gamma^2 - 4\gamma^3 + \gamma^4$$

$$p_6 = \frac{1}{8} - \frac{4}{3}\gamma + 4\gamma^2 - 4\gamma^3 + \gamma^4$$

$$p_7 = \frac{1}{12} - \gamma + \frac{7}{2}\gamma^2 - 4\gamma^3 + \gamma^4$$

$$p_8 = \frac{1}{24} - \frac{2}{3}\gamma + 3\gamma^2 - 4\gamma^3 + \gamma^4$$

Рис. 5: Умови порядку для метода SDIRK

Розділ III

Python

Для розробки програми я використала інтерпретовану мову об'єктно-орієнтованого програмування високого рівня зі строгою динамічною типізацією (коли основна частина перевірок типів виконується під час виконання програми, а не під час компіляції) Python. Останнім часом вона є дуже зручним програмним середовищем для розв'язування математичних задач.

Мова Python почала розроблятись в кінці 1980-х років співробітником голландського інституту CWI Гвідо ван Россумом. Її було створено на основі деяких інших мов, як, наприклад, ABC (відступи для групування операторів, високорівневі структури даних (фактично, Python створювався як спроба виправити помилки, допущені при проєктуванні ABC)), Modula-3 (пакети, модулі, використання else спільно з try та except, іменовані аргументи функцій), Smalltalk (ООП), Java (модулі logging, unittest, threading), Fortran (зрізи масивів, комплексна арифметика), тощо. 3 грудня 2008 року, після тривалого тестування, вийшла перша версія Python 3000. Інтерпретатор Python має інтерактивний режим роботи, при якому введені з клавіатури вирази відразу ж виконуються, а результат виводиться на екран. Цей режим цікавий тим, що можна протестувати в реальному часі будь-який фрагмент коду, перш ніж використовувати його в основній програмі, або просто використовувати як калькулятор з великим набором функцій.

Крім стандартної бібліотеки існує багато інших, що надають інтерфейс до всіх системних викликів на різних платформах. Існує велика кількість прикладних бібліотек для Python у різноманітних галузях: веброзробка, бази даних, обробка зображень, обробка тексту, чисельні методи, програми операційної системи тощо.

Бібліотека NumPy для роботи з багатовимірними масивами дозволяє досягти продуктивності наукових розрахунків, порівнянної зі спеціалізованими пакетами. Для науково-технічної мети найбільшого поширення набуло використання matplotlib — бібліотеки з інтерфейсом, аналогічним MATLAB Plot Tool.

Python, як і багато інших інтерпретованих мов, які не застосовують, наприклад, JIT-компілятори, мають загальний недолік — порівняно низьку швидкість виконання програм. Однак, у випадку з Python цей недолік компенсується зменшенням часу розробки програми.

Числові експерименти та програмна реалізація методів Рунге-Кутта

В цьому розділі детально розглянемо код програми та результати експериментів.

У файлі `ode_solvers/ode_solver.py` визначено базовий абстрактний клас для розв'язання задач методом Рунге-Кутта. Для конкретної задачі `ode_problem`

визначені функція f, початкові умови $y0$ та часовий інтервал $t0, T$. Також клас ODESolver приймає компоненти таблиці Батчера: A, b, c.

```
class ODESolver:
    """ODESolver superclass

        ODE:
        u' = f(u, t)
        u(t_0) = U_t0
    """

    def __init__(self, ode_problem: ODEModel, A: np.array, b: np.array, c: np.array, tolerance: float):
        self.f = ode_problem.f
        self.y0 = ode_problem.y0.astype(float)  # initial condition
        self.num_init_conditions = len(self.y0)

        self.u = None  # solution
        self.i = None  # current number of step iteration

        self.h = (ode_problem.T - ode_problem.t0) / (ode_problem.number_of_points_to_discretization + 1)
        self.t = np.linspace(ode_problem.t0, ode_problem.T, ode_problem.number_of_points_to_discretization + 2)  # array of time points corresponding to solution

        self.tol = tolerance

        # setting Butcher table properties:
        self.A = A
        self.b = b
        self.c = c
        self.s = len(self.b)

    def step(self):
        ti, yi = self.t[0], self.y0  # initial condition points
        current_time_point = ti
        yield ti , np.array(yi)  # first point (begging point)
        for ti in self.t[1:]:
            yi += self.h * self.phi(current_time_point, yi)
            current_time_point = ti
            yield ti, np.array(yi)

    def solve(self):
        return np.array(list(self.step()))

    def phi(self, current_time, current_y):
        """Advance solution one time step."""
        raise NotImplementedError
```

Клас складається із трьох методів `step()`, `solve()`, `phi`. Метод `step()` - обчислює розв'язок для кожного кроку часу, викликаючи метод `phi()`, що робить крок конкретного методу Рунге-Кутта для даного часу. Метод () - абстрактний, це означає, що в класах, що наслідують даний він повинен бути реалізований.

Для неявних методів Рунге-Кутта, кожного разу, коли викликається метод `phi()`, що обчислює суму $b_i K_i$, обчислюється якобіан $J = \dfrac{df}{dy}(y_i)$ і разом з ним метод phi_solve, що повертає похідну $\varphi(t_{n,i}, y(t_{n,i}))$:

```
class ImplicitRungeKutta(ODESolver):

```

```
    def phi(self, t0, y0):
        """
        Calculates the summation of b_j*Y_j in one step of the RungeKutta method with
        y_{n+1} = y_{n} + h * sum_{j=1}^{s} b_{j}*Y
        where j=1,2,...,s, and s is the number of stages, b the nodes, and Y the stage values of the method.
        Parameters:
        -------------
        t0 = float, current timestep
        y0 = 1 x m vector, the last solution y_n. Where m is the length of the initial condition y_0 of the IVP.
        """
        M = 1000  # max number of newton iterations

        stage_der = np.array(self.s * [self.f(t0, y0)])  # initial value: Y_0
        J = jacobian(self.f)(t0, y0)
        stage_val = self.phi_solve(t0, y0, stage_der, J, M)

        return np.array([
            self.b @ stage_val.reshape(self.s, self.num_init_conditions)[:,j] for j in range(self.num_init_conditions)
        ])
```

В тілі методу $phi_{solve}()$, викликається функція $phi_{newton}()$, що стартує Ньютонівську ітерацію з максимальною кількістю ітерацій - , якщо ітерація не збіглася за цю кількість, генерується виняток. В цій реалізації, крок методу константний - $h = \frac{T-t0}{N}$.

```
        """
        This function solves the sm x sm system F(Y_i)=0 by Newtons method with an initial guess init_val.
        Parameters:
        -------------
        t0 = float, current timestep
        y0 = 1 x m vector, the last solution y_n. Where m is the length of the initial condition y_0 of the IVP.
        init_val = initial guess for the Newton iteration
        J = m x m matrix, the Jacobian matrix of f() evaluated in y_i
        M = maximal number of Newton iterations
        Returns:
        -------------
        The stage derivative  Y_i
        """

        JJ = np.eye(self.s * self.num_init_conditions) - self.h * np.kron(self.A, J)
        lu_factor = linalg.lu_factor(JJ)
        for i in range(M):
            init_val, norm_d = self.phi_newtonstep(t0, y0, init_val, lu_factor)
            if norm_d < self.tol:
                break
            elif i == M - 1:
                raise ValueError("The Newton iteration did not converge.")
        return init_val
        """
        This function solves the sm x sm system F(Y_i)=0 by Newtons method with an initial guess init_val.
        Parameters:
        -------------
        t0 = float, current timestep
```

```
        y0 = 1 x m vector, the last solution y_n. Where m is the length of the initial condition y_0 of the IVP.
        init_val = initial guess for the Newton iteration
        J = m x m matrix, the Jacobian matrix of f() evaluated in y_i
        M = maximal number of Newton iterations
        Returns:
        -------------
        The stage derivative  Y _i
        """

        JJ = np.eye(self.s * self.num_init_conditions) - self.h * np.kron(self.A, J)
        lu_factor = linalg.lu_factor(JJ)
        for i in range(M):
            init_val, norm_d = self.phi_newtonstep(t0, y0, init_val, lu_factor)
            if norm_d < self.tol:
                break
            elif i == M - 1:
                raise ValueError("The Newton iteration did not converge.")
        return init_val
```

Для кожного Ньютонівського кроку `phi_newtonstep()`, метод класу ImplicitRungeKutta розв'язує алгебраїчну систему рівнянь:

$$(I - hA \otimes J)d = -F$$

де

$$d = \varphi(t_{n+1,i}, y(t_{n+1,i})) - \varphi(t_{n,i}, y(t_{n,i})),$$

$$F_i = \varphi(t_{n,i}, y(t_{n,i})) - y_{n-1} - h\sum_{j=1}^{s} a_{ij} f(\varphi(t_{n,j}, y(t_{n,j}))).$$

```
    def phi_newtonstep(self, t0, y0, init_val, lu_factor):
        """
        Takes one Newton step by solvning
         G  (Y_i)(Y^(n+1)_i-Y^(n)_i) = -G(Y_i), where
        G(Y_i) = Y_i - y_n - h*sum(a_{ij}* Y _j ) for j = 1,...,s
        Parameters:
        -------------
        t0 = float, current timestep
        y0 = 1 x m vector, the last solution y_n. Where m is the length of the initial condition y_0 of the IVP.
        init_val = initial guess for the Newton iteration
        lu_factor = (lu, piv) see documentation for linalg.lu_factor
        Returns:
        The difference Y^(n+1)_i-Y^(n)_i
        """
        d = linalg.lu_solve(lu_factor, -self.F(init_val.flatten(), t0, y0))
        return init_val.flatten() + d, linalg.norm(d)

    def F(self, stage_der, t0, y0):
        """
        Returns the subtraction  Y _ {i}-f(t_{n}+c_{i}*h, Y_{i}), where Y are
        the stage values,  Y   the stage derivatives and f the function of
        the IVP  y  =f(t,y) that should be solved by the RK-method.
        Parameters:
        -------------
        stage_der = initial guess of the stage derivatives  Y
        t0 = float, current timestep
```

```
        y0 = 1 x m vector, the last solution y_n. Where m is the length of the initial condition y_0 of the IVP.
        """
        stage_der_new = np.empty((self.s, self.num_init_conditions))  # the i:th stage_der is on the i:th row
        for i in range(self.s):  # iterate over all stage_der
            stageVal = y0 + np.array([
                self.h * np.dot(self.A[i,:],
                stage_der.reshape(self.s, self.num_init_conditions)[:, j]) for j in range(self.num_init_conditions)
            ])
            stage_der_new[i, :] = self.f(t0 + self.c[i] * self.h, stageVal)  # the ith stage_der is set on the ith row
        return stage_der - stage_der_new.reshape(-1)
```

Реалізація методів $phi()$ для явного і діагонально неявного методів Рунге-Кутта відрізняється, оскільки різняться і самі методи.

Для явного методу:

```
class ExplicitRungeKutta(ODESolver):

    def __init__(self, ode_problem: ODEModel, A: np.array, b: np.array, c: np.array, tolerance: float):
        super().__init__(ode_problem, A, b, c, tolerance)
        self.h = self.t[1] - self.t[0]

    def phi(self, current_time, current_y):
        K = np.zeros(self.s, dtype=float)
        for s in range(self.s):
            x = current_time + self.c[s] * self.h
            y = current_y
            for j in range(s):
                y += self.A[s, j] * K[j] * self.h
            K[s] = self.f(x, y)

        return self.h * np.sum(K.T @ self.b)
```

Для діагонально неявного методу:

```

class DiagonallyImplicitRungeKutta(ImplicitRungeKutta):

    def __init__(self, ode_problem: ODEModel, A: np.array, b: np.array, c: np.array, tolerance: float):
        super().__init__(ode_problem, A, b, c, tolerance)

    def phi_solve(self, current_time, current_y, init_val, J, M):
        """
        This function solves F(Y_i)=0 by solving s systems of size m
            x m each.
         Newtons method is used with an initial guess init_val.

        Parameters:
        -------------
        t0 = float, current timestep
        y0 = 1 x m vector, the last solution y_n. Where m is the length of the initial condition y_0 of the IVP.
        init_val = initial guess for the Newton iteration
        J = m x m matrix, the Jacobian matrix of f() evaluated in y_i
        M = maximal number of Newton iterations

        Returns:
        -------------
```

```
        The stage derivative  Y _i
        """
        JJ = np.eye(self.num_init_conditions) - self.h * self.A[0,0] * J
        lu_factor = linalg.lu_factor(JJ)
        for i in range(M):
            init_val, norm_d = self.phi_newtonstep(current_time, current_y,
    init_val, J, lu_factor)
            if norm_d < self.tol:
                break
            elif i == M - 1:
                raise ValueError("The Newton iteration did noconverge.")
        return init_val



    def phi_newtonstep(self, current_time, current_y, init_val, J, lu_factor):
        """
        Takes one Newton step by solvning
         G  (Y_i)(Y^(n+1)_i-Y^(n)_i)=-G(Y_i)
        where G(Y_i) = Y_i -  h a Y _i  - y_n - h*sum(a_{ij}* Y _j ) for j=1,...,
    i-1

        Parameters:
        -------------
        t0 = float, current timestep
        y0 = 1 x m vector, the last solution y_n. Where m is the length of the
    initial condition y_0 of the IVP.
        init_val = initial guess for the Newton iteration
        lu_factor = (lu, piv) see documentation for linalg.lu_factor

        Returns:
        The difference Y^(n+1)_i-Y^(n)_i
        """
        x = []
        for i in range(self.s):  # solving the s mxm systems
            rhs = -self.F(
                init_val.flatten(), current_time, current_y
            )[i * self.num_init_conditions : (i + 1) * self.num_init_conditions
    ] + np.sum(
                [self.h * self.A[i,j] * J @ x[j] for j in range(i)],
                axis = 0
            )
            d = linalg.lu_solve(lu_factor, rhs)
            x.append(d)
        return init_val + x, linalg.norm(x)
```

Для різних методів Рунге-Кутта використовуються різні матриці Батчера, ми розглянемо декілька із них:

1.Матриця Ейлера (явний метод)

$$\begin{array}{c|c} \mathbf{0} & \mathbf{1} \\ \hline & \mathbf{0} \end{array}$$

2. Матриця середньої точки (явний метод)

$$\begin{array}{c|cc} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \hline & \mathbf{\frac{1}{2}} & \mathbf{\frac{1}{2}} \end{array}$$

3.

$$\begin{array}{c|ccc} \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{\frac{1}{2}} & \mathbf{\frac{1}{2}} & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{-1} & \mathbf{2} & \mathbf{0} \\ \hline & \mathbf{\frac{1}{6}} & \mathbf{\frac{2}{3}} & \mathbf{\frac{1}{6}} \end{array}$$

4.

$$\begin{array}{c|ccc} \frac{1}{2} - \frac{\sqrt{15}}{15} & \frac{5}{36} & \frac{2}{9} - \frac{\sqrt{15}}{15} & \frac{5}{36} - \frac{\sqrt{5}}{30} \\ \frac{1}{2} & \frac{5}{36} + \frac{\sqrt{15}}{24} & \frac{2}{9} & \frac{5}{36} - \frac{\sqrt{15}}{24} \\ \frac{1}{2} + \frac{\sqrt{15}}{10} & \frac{5}{36} + \frac{\sqrt{15}}{30} & \frac{2}{9} + \frac{\sqrt{15}}{15} & \frac{5}{36} \\ \hline & \frac{5}{18} & \frac{4}{9} & \frac{5}{18} \end{array}$$

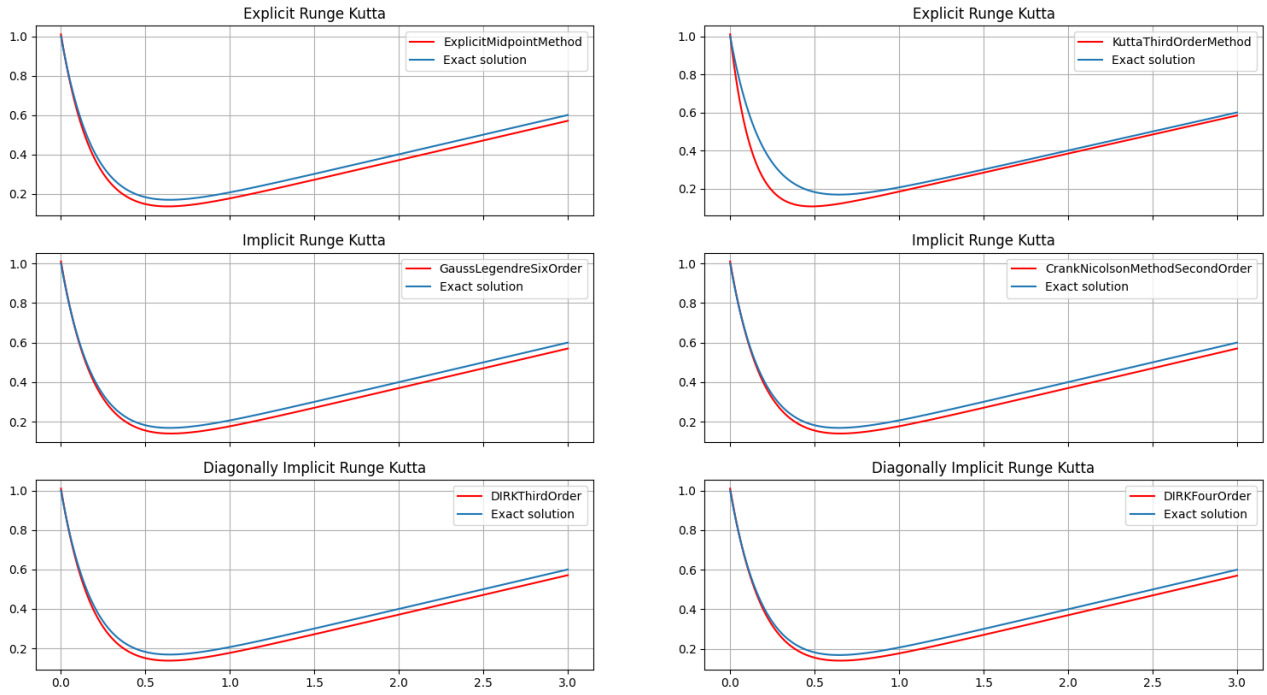
Тестування

```python
def solve(
    ode_problem: ODEModel,                # differential problem, which we want to solve,
    ode_solver: ode_solvers.ODESolver,  # ODE solver
    method: Callable,                   # Butcher matrix funciton
    tol: float                          # tolerance
):
    A, b, c = method()
    solver = ode_solver(ode_problem, A, b, c, tol)
    return solver.solve()



def solve_ode_test(
    ode_problem: ODEModel,          # differential problem, which we want to solve
    tol=1e-5                        # tolerance
):
    test_explicit_methods = [ForwardEuler, KuttaThirdOrderMethod]
    tests_implicit_methods = [GaussLegendreSixOrder, CrankNicolsonMethodSecondOrder]
    tests_diagonally_implicit_methods = [DIRKThirdOrder, DIRKFourOrder]

    test_methods = test_explicit_methods + tests_implicit_methods + tests_diagonally_implicit_methods

    # build exact solution points if exists:
    if ode_problem.exact_test_solution:
        time_points_exact = np.linspace(ode_problem.t0, ode_problem.T, ode_problem.number_of_points_to_discretization)
        exact_solution = ode_problem.exact_test_solution(time_points_exact)

    # create an figure to display plots
    figure, axes = generate_subplots(
        k=len(test_explicit_methods) + len(tests_implicit_methods) + len(tests_diagonally_implicit_methods),
        row_wise=True
    )

    noise = 0.01  # add some noise in order to look at solution when he very good)

    # EXPLISIT METHODS
```

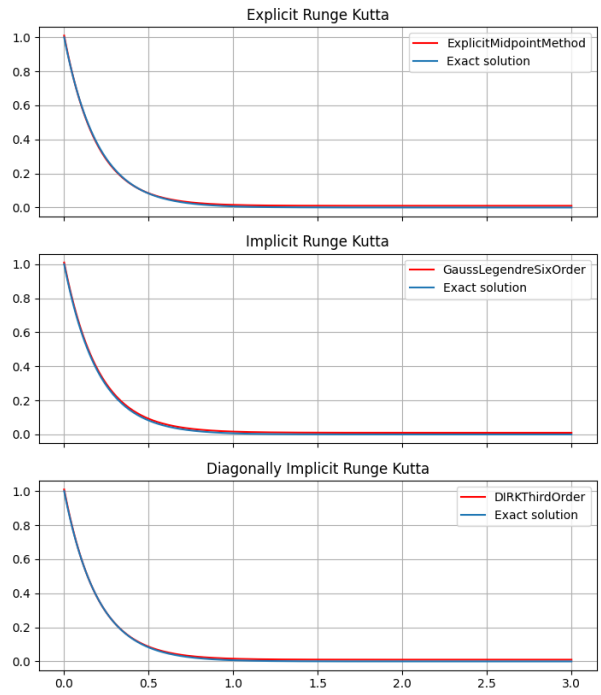
```
    for k, method in enumerate(test_explicit_methods):
        u = solve(
            ode_problem=ode_problem,
            ode_solver=ode_solvers.ExplicitRungeKutta,
            method=method,
            tol=tol
        )
        # plot result:
        axes[k].plot(u[:,0], u[:,1] + noise, color='red', label=f"{method.__name__}")
        if ode_problem.exact_test_solution:
            axes[k].plot(time_points_exact, exact_solution, label="Exact solution")
        axes[k].grid(True)
        axes[k].set_title("Explicit Runge Kutta")
        axes[k].legend()

    # IMPLISIT METHODS
    for k, method in enumerate(tests_implicit_methods):
        k += len(test_explicit_methods)

        u = solve(
            ode_problem=ode_problem,
            ode_solver=ode_solvers.ImplicitRungeKutta,
            method=method,
            tol=tol
        )
        # plot result:
        axes[k].plot(u[:,0], u[:,1] + noise, color='red', label=f"{method.__name__}")
        if ode_problem.exact_test_solution:
            axes[k].plot(time_points_exact, exact_solution, label="Exact solution")
        axes[k].grid(True)
        axes[k].set_title("Implicit Runge Kutta")
        axes[k].legend()

    # DIAGONALY IMPLISIT METHODS
    for k, method in enumerate(tests_diagonally_implicit_methods):
        k += len(test_explicit_methods + tests_implicit_methods)

        u = solve(
            ode_problem=ode_problem,
            ode_solver=ode_solvers.DiagonallyImplicitRungeKutta,
            method=method,
            tol=tol
        )
        # plot result:
        axes[k].plot(u[:,0], u[:,1] + noise, color='red', label=f"{method.__name__}")
        if ode_problem.exact_test_solution:
            axes[k].plot(time_points_exact, exact_solution, label="Exact solution")
        axes[k].grid(True)
        axes[k].set_title("Diagonally Implicit Runge Kutta")
        axes[k].legend()
    figure.canvas.set_window_title("Solution for ode problem")
    plt.show()


def example_1():
    problems = [
        problem_scalar_1,
```

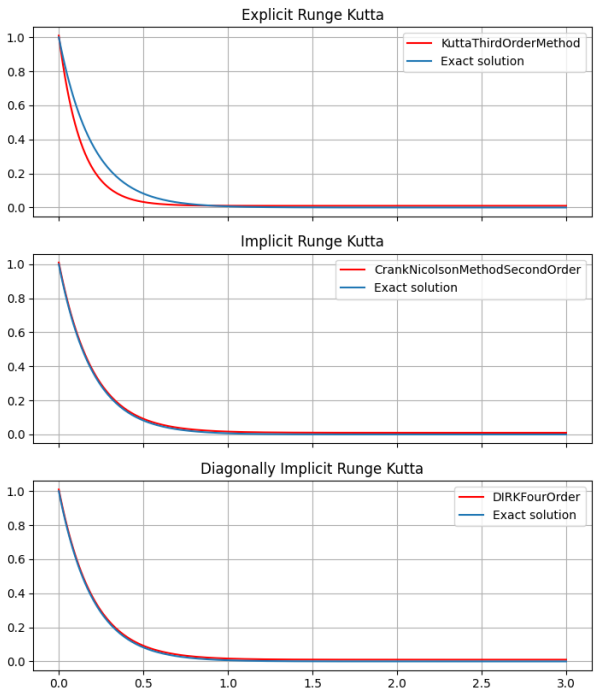
```
        problem__scalar_2,
        problem_nonatonomous_1,
    ]

    # solve problems all avalible methods:
    for problem in problems:
        solve_ode_test(ode_problem=problem)


def example_2():
    # without exact solution
    solve_ode_test(ode_problem=problem_nonatonomous_2)


if __name__ == "__main__":
    example_1()
    example_2()
```

Результати тестів

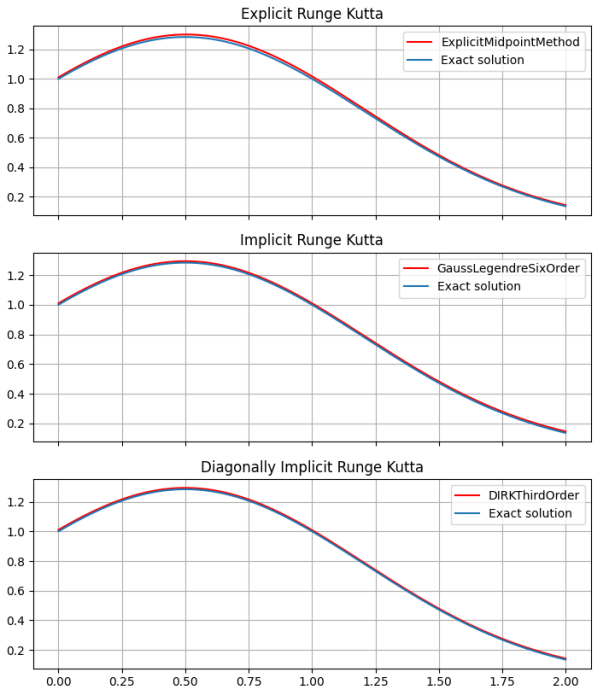


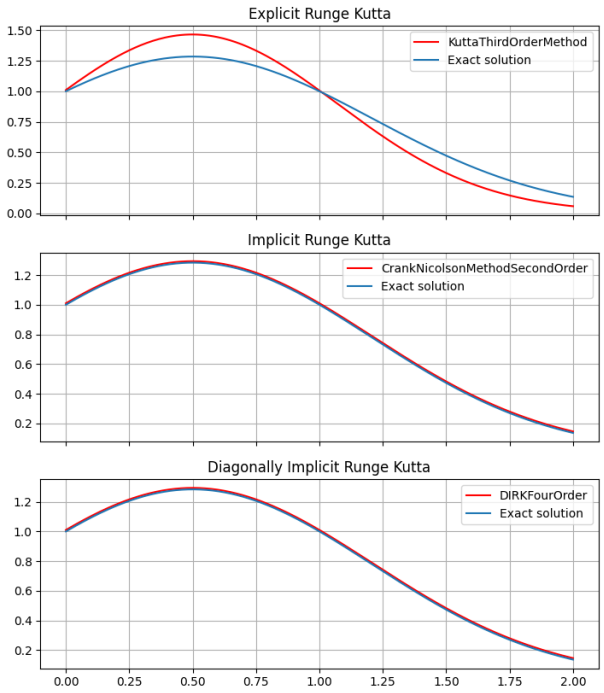
Тест для системи 1: $\begin{cases} y'(t) = -5y + t \qquad \forall t \in I[0,3], \\ y(0) = 1, \\ u_{real} = e^{-5t} + t/5 \end{cases}$



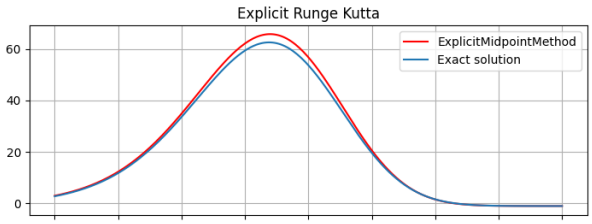


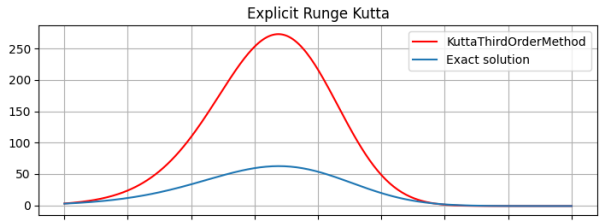
Тест для системи 2: $\begin{cases} y'(t) = -5y \quad \forall t \in I[0,3], \\ y(0) = 1, \\ u_{real} = e^{-5t} \end{cases}$

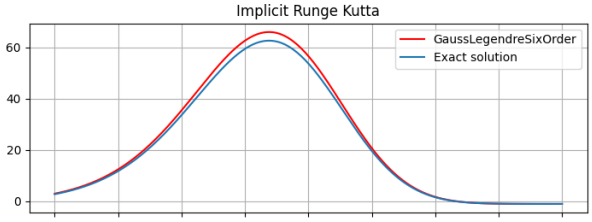


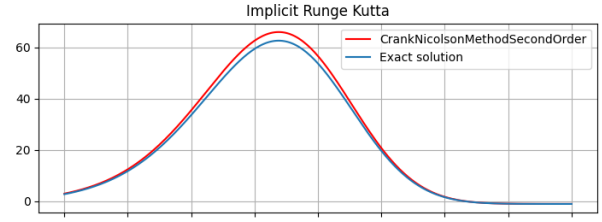


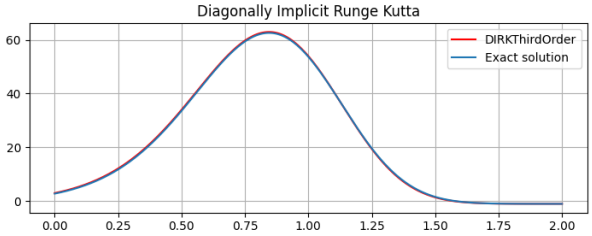
Тест для системи 3: $\begin{cases} y'(t) = y(1-2t) \quad \forall t \in I[0,2], \\ y(0) = 1, \\ u_{real} = e^{t-t^2} \end{cases}$

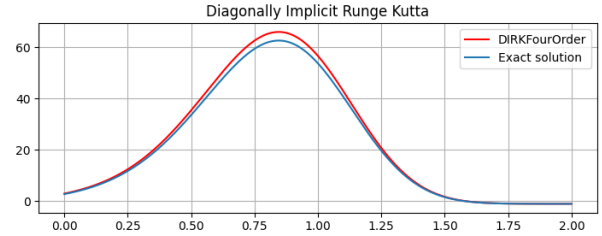




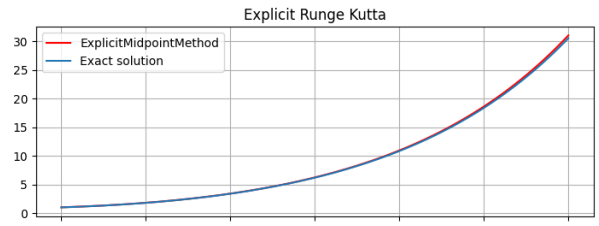


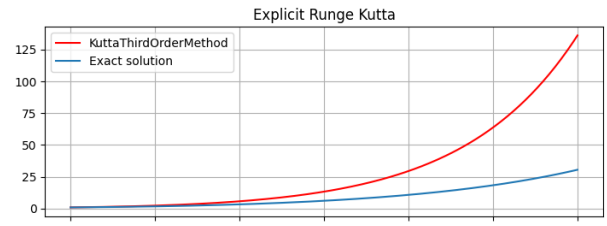


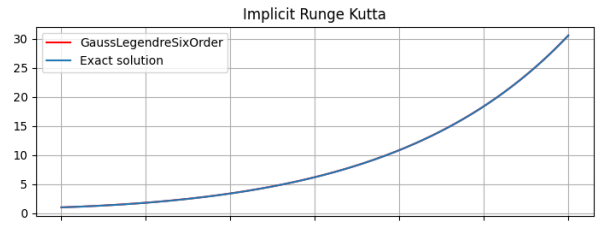


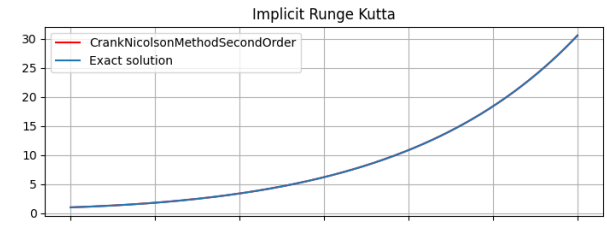


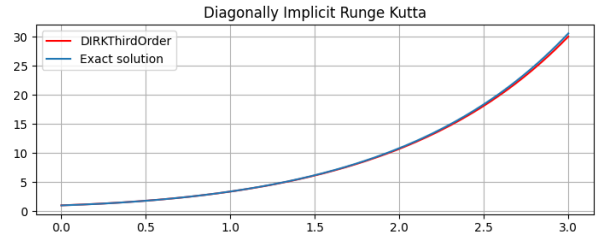
Тест для системи 4:
$$\begin{cases} y'(t) = (y+1)(5-7t^2) \quad \forall t \in I[0,2], \\ y(0) = 3, \\ u_{real} = e^{5*t-\frac{7*t^3}{3}} - 1 \end{cases}$$

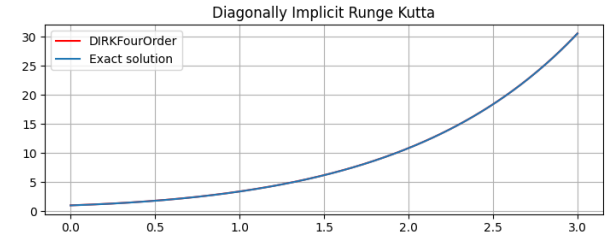




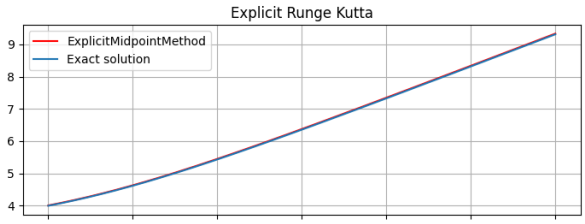


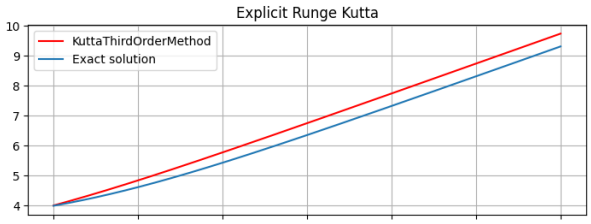


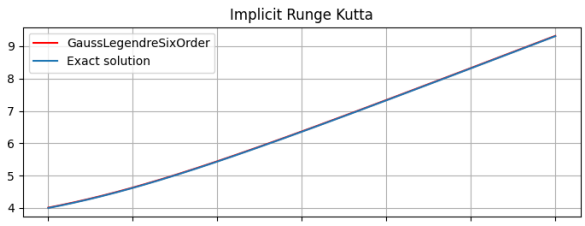


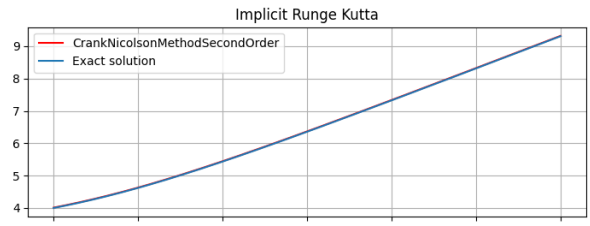


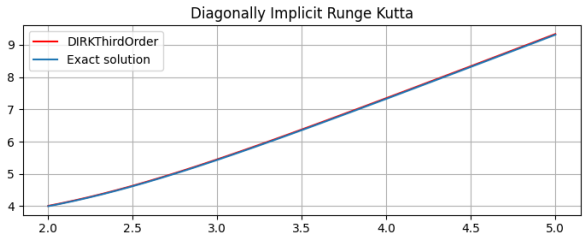
Тест для системи 5:
$$\begin{cases} y'(t) = sin(t) + y \qquad \forall t \in I[0,3] \\ y(0) = 1, \\ u_{real} = \frac{3e^t}{2} - \frac{sin(t)}{2} - \frac{cos(t)}{2} \end{cases}$$

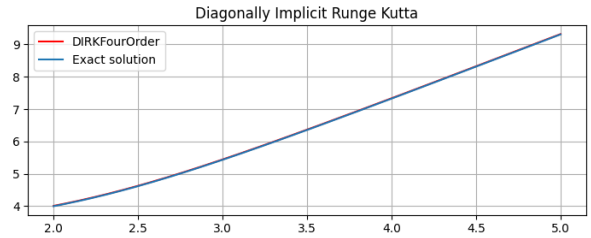












Тест для системи 6:
$$\begin{cases} y'(t) = e^{2t}/e^{y}) & \forall t \in I[2,5] \\ y(2) = 4, \\ u_{real} = \log(\frac{e^{2t}}{2} + \frac{e^{4}}{2}) \end{cases}$$

Висновок

В цій роботі було:

- досконало розглянуто методи Рунге-Кутта
- реалізовано програму для розв'язування лінійних та нелінійних задач Коші
- показано, що за допомогою чисельних методів неважко знаходити розв'язок, що має відносно малу похибку і тому дану програму можна використовувати для науково-учбового процесу для розв'язування задач.

Результати та їх точність підтверджено експериментально за допомогою інтерпретованої мови об'єктно-орієнтованого програмування Python.

Список використаних джерел

1. Самарский А. А., Гулин А. В. Численные методы: Учеб, пособие для вузов,— М.: Наука. Гл. ред. физ-мат. лит., 1989.— 432 с.— ISBN 5-02-013996-3.

2. E. Hairer, G. Wanner. (1996), Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems, Berlin, New York: Springer-Verlag

3.Iserles, Arieh (1996), A First Course in the Numerical Analysis of Differential Equations, Cambridge University Press

4. Lambert, J.D (1991), Numerical Methods for Ordinary Differential Systems. The Initial Value Problem

5. G. Faccanoni. Analyse numérique cours

6. Press, William H.; Teukolsky, Saul A.; Vetterling, William T.; Flannery, Brian P. (2007), "Section 17.1 Runge-Kutta Method Numerical Recipes: The Art of Scientific Computing (3rd ed.), Cambridge University Press

7.Stoer, Josef; Bulirsch, Roland (2002), Introduction to Numerical Analysis (3rd ed.), Berlin, New York: Springer-Verlag

8. Süli, Endre; Mayers, David (2003), An Introduction to Numerical Analysis, Cambridge University Press

9.C. Runge, "Ueber die numerische Auflösung von Differentialgleichungen"Math. Ann. , 46 (1895) pp. 167–178

10. Bakhvalov, N.S. (1977) Numerical Methods: Analysis, Algebra, Ordinary Differential Equations. MIR, Moscow.

Код програми

Репозиторій програми можна знайти за посиланням:

$$https://github.com/Imply0916/Diploma$$

```python
import autograd.numpy as np
from autograd import grad, jacobian
from scipy import linalg
from typing import Callable
import numpy.typing as npt

from ode_models import ODEModel


class ODESolver:
    """ODESolver superclass

        ODE:
        u' = f(u, t)
        u(t_0) = U_t0
    """

    def __init__(self, ode_problem: ODEModel, A: np.array, b: np.array, c: np.array, tolerance: float):
        self.f = ode_problem.f
        self.y0 = ode_problem.y0.astype(float)  # initial condition
        self.num_init_conditions = len(self.y0)

        self.u = None  # solution
        self.i = None  # current number of step iteration

        self.h = (ode_problem.T - ode_problem.t0) / (ode_problem.number_of_points_to_discretization + 1)
        self.t = np.linspace(ode_problem.t0, ode_problem.T, ode_problem.number_of_points_to_discretization + 2)  # array of time points corresponding to solution

        self.tol = tolerance

        # setting Butcher table properties:
        self.A = A
        self.b = b
        self.c = c
        self.s = len(self.b)

    def step(self):
        ti, yi = self.t[0], self.y0  # initial condition points
        current_time_point = ti
        yield ti , np.array(yi)  # first point (begging point)
        for ti in self.t[1:]:
            yi += self.h * self.phi(current_time_point, yi)
            current_time_point = ti
            yield ti, np.array(yi)

    def solve(self):
        return np.array(list(self.step()))

    def phi(self, current_time, current_y):
        """Advance solution one time step."""
```

```
51          raise NotImplementedError
```

Лістинг 1: ode_solvers/ode_solver.py

```
1  import autograd.numpy as np
2  from autograd import grad, jacobian
3  from scipy import linalg
4
5  from .ode_solver import ODESolver
6  from ode_models import ODEModel
7
8
9  class ExplicitRungeKutta(ODESolver):
10
11     def __init__(self, ode_problem: ODEModel, A: np.array, b: np.array, c: np.
       array, tolerance: float):
12         super().__init__(ode_problem, A, b, c, tolerance)
13         self.h = self.t[1] - self.t[0]
14
15     def phi(self, current_time, current_y):
16         K = np.zeros(self.s, dtype=float)
17         for s in range(self.s):
18             t = current_time + self.c[s] * self.h
19             y = current_y
20             for j in range(s):
21                 y += self.A[s, j] * K[j] * self.h
22             K[s] = self.f(t, y)
23
24         return self.h * K.T @ self.b
```

Лістинг 2: ode_solvers/explicit_runge_kutta.py

```
1  import autograd.numpy as np
2  from autograd import grad, jacobian
3  from scipy import linalg
4
5  from .ode_solver import ODESolver
6  from ode_models import ODEModel
7
8
9  class ImplicitRungeKutta(ODESolver):
10
11     def __init__(self, ode_problem: ODEModel, A: np.array, b: np.array, c: np.
       array, tolerance: float):
12         super().__init__(ode_problem, A, b, c, tolerance)
13
14
15     def phi(self, t0, y0):
16         """
17         Calculates the summation of b_j*Y_j in one step of the RungeKutta
       method with
18         y_{n+1} = y_{n} + h * sum_{j=1}^{s} b_{j}*Y
19         where j=1,2,...,s, and s is the number of stages, b the nodes, and Y
       the stage values of the method.
20         Parameters:
21         -------------
22         t0 = float, current timestep
23         y0 = 1 x m vector, the last solution y_n. Where m is the length of the
       initial condition y_0 of the IVP.
24         """
25         M = 1000  # max number of newton iterations
26
27         stage_der = np.array(self.s * [self.f(t0, y0)])  # initial value:
        Y _0
```

```
        J = jacobian(self.f)(t0, y0)
        stage_val = self.phi_solve(t0, y0, stage_der, J, M)

        return np.array([
            self.b @ stage_val.reshape(self.s, self.num_init_conditions)[:,j] for j in range(self.num_init_conditions)
        ])

    def phi_solve(self, t0, y0, init_val, J, M):
        """
        This function solves the sm x sm system F(Y_i)=0 by Newtons method with an initial guess init_val.
        Parameters:
        -------------
        t0 = float, current timestep
        y0 = 1 x m vector, the last solution y_n. Where m is the length of the initial condition y_0 of the IVP.
        init_val = initial guess for the Newton iteration
        J = m x m matrix, the Jacobian matrix of f() evaluated in y_i
        M = maximal number of Newton iterations
        Returns:
        -------------
        The stage derivative  Y _i
        """

        JJ = np.eye(self.s * self.num_init_conditions) - self.h * np.kron(self.A, J)
        lu_factor = linalg.lu_factor(JJ)
        for i in range(M):
            init_val, norm_d = self.phi_newtonstep(t0, y0, init_val, lu_factor)
            if norm_d < self.tol:
                break
            elif i == M - 1:
                raise ValueError("The Newton iteration did not converge.")
        return init_val


    def phi_newtonstep(self, t0, y0, init_val, lu_factor):
        """
        Takes one Newton step by solvning
         G  (Y_i)(Y^(n+1)_i-Y^(n)_i) = -G(Y_i), where
        G(Y_i) = Y_i - y_n - h*sum(a_{ij}* Y _j ) for j = 1,...,s
        Parameters:
        -------------
        t0 = float, current timestep
        y0 = 1 x m vector, the last solution y_n. Where m is the length of the initial condition y_0 of the IVP.
        init_val = initial guess for the Newton iteration
        lu_factor = (lu, piv) see documentation for linalg.lu_factor
        Returns:
        The difference Y^(n+1)_i-Y^(n)_i
        """
        d = linalg.lu_solve(lu_factor, -self.F(init_val.flatten(), t0, y0))
        return init_val.flatten() + d, linalg.norm(d)

    def F(self, stage_der, t0, y0):
        """
        Returns the subtraction  Y _ {i}-f(t_{n}+c_{i}*h, Y_{i}), where Y are
        the stage values,  Y   the stage derivatives and f the function of
        the IVP  y  =f(t,y) that should be solved by the RK-method.
        Parameters:
        -------------
        stage_der = initial guess of the stage derivatives  Y
```

```
        t0 = float, current timestep
        y0 = 1 x m vector, the last solution y_n. Where m is the length of the initial condition y_0 of the IVP.
        """
        stage_der_new = np.empty((self.s, self.num_init_conditions))  # the i:th stage_der is on the i:th row
        for i in range(self.s):  # iterate over all stage_der
            stageVal = y0 + np.array([
                self.h * np.dot(self.A[i,:],
                stage_der.reshape(self.s, self.num_init_conditions)[:, j]) for j in range(self.num_init_conditions)
            ])
            stage_der_new[i, :] = self.f(t0 + self.c[i] * self.h, stageVal)  # the ith stage_der is set on the ith row
        return stage_der - stage_der_new.reshape(-1)
```

Лістинг 3: ode_solvers/implicit_runge_kutta.py

```
# implementation of Singly Diagonally Implicit RungeKutta Method(SDIRK)

import autograd.numpy as np
from autograd import grad, jacobian
from scipy import linalg

from .implicit_runge_kutta import ImplicitRungeKutta
from ode_models import ODEModel


class DiagonallyImplicitRungeKutta(ImplicitRungeKutta):

    def __init__(self, ode_problem: ODEModel, A: np.array, b: np.array, c: np.array, tolerance: float):
        super().__init__(ode_problem, A, b, c, tolerance)

    def phi_solve(self, current_time, current_y, init_val, J, M):
        """
        This function solves F(Y_i)=0 by solving s systems of size m
            x m each.
        Newtons method is used with an initial guess init_val.

        Parameters:
        -------------
        t0 = float, current timestep
        y0 = 1 x m vector, the last solution y_n. Where m is the length of the initial condition y_0 of the IVP.
        init_val = initial guess for the Newton iteration
        J = m x m matrix, the Jacobian matrix of f() evaluated in y_i
        M = maximal number of Newton iterations

        Returns:
        -------------
        The stage derivative  Y _i
        """
        JJ = np.eye(self.num_init_conditions) - self.h * self.A[0,0] * J
        lu_factor = linalg.lu_factor(JJ)
        for i in range(M):
            init_val, norm_d = self.phi_newtonstep(current_time, current_y, init_val, J, lu_factor)
            if norm_d < self.tol:
                break
            elif i == M - 1:
                raise ValueError("The Newton iteration did noconverge.")
        return init_val
```

```
43
44
45
46      def phi_newtonstep(self, current_time, current_y, init_val, J, lu_factor):
47          """
48          Takes one Newton step by solvning
49           G  (Y_i)(Y^(n+1)_i-Y^(n)_i)=-G(Y_i)
50          where G(Y_i) = Y_i -  h a Y _i  - y_n - h*sum(a_{ij}* Y _j ) for j=1,...,
    i-1
51
52          Parameters:
53          -------------
54          t0 = float, current timestep
55          y0 = 1 x m vector, the last solution y_n. Where m is the length of the
    initial condition y_0 of the IVP.
56          init_val = initial guess for the Newton iteration
57          lu_factor = (lu, piv) see documentation for linalg.lu_factor
58
59          Returns:
60          The difference Y^(n+1)_i-Y^(n)_i
61          """
62          x = []
63          for i in range(self.s):  # solving the s mxm systems
64              rhs = -self.F(
65                  init_val.flatten(), current_time, current_y
66              )[i * self.num_init_conditions : (i + 1) * self.num_init_conditions
    ] + np.sum(
67                  [self.h * self.A[i,j] * J @ x[j] for j in range(i)],
68                  axis = 0
69              )
70              d = linalg.lu_solve(lu_factor, rhs)
71              x.append(d)
72          return init_val + x, linalg.norm(x)
```

Лістинг 4: ode_solvers/diagonally_implicit_runge_kutta.py

```
1  import autograd.numpy as np
2  from dataclasses import dataclass
3  from typing import Callable, Union
4
5
6  """
7  ODE:
8  u' = f(u, t)
9  u(t_0) = U_t0
10 """
11 @dataclass
12 class ODEModel:
13     f: Callable[[np.array, np.array], np.array]  # problem function - f(u, t)
14     exact_test_solution: Union[Callable[[np.array], np.array], None]  #
    function, when we know exact solution - u(t), None otherwise
15     t0: np.array  # because we made general solve methods for arbitrary
    dimentions, then start point may be in 3-dim t0 = (2, 4, 5)
16     T: np.array  # because we made general solve methods for arbitrary
    dimentions, then end point may be in 3-dim T = (6, 1, 0)
17     y0: np.array  # value in t0 point, also for arbitrary dimentions
18     number_of_points_to_discretization: int = 250
19
20
21 # 1) Scalar Differential Equation
22
23 problem_scalar_1 = ODEModel(
24     f = lambda t, y: -5. * y + t,
```

```
    exact_test_solution = lambda t: np.exp(- 5. * t) + t / 5,
    t0 = np.array([0]),
    T = np.array([3]),
    y0 = np.array([1])
)


problem_scalar_2 = ODEModel(
    f = lambda t, y: -5. * y,
    exact_test_solution = lambda t: np.exp(- 5. * t),
    t0 = np.array([0]),
    T = np.array([3]),
    y0 = np.array([1])
)


#======================
#  2) Nonautonomous ODE
problem_nonatonomous_1 = ODEModel(
    f = lambda t, y: y * (1 - 2 * t),
    exact_test_solution = lambda t: np.exp(t - t ** 2),
    t0 = np.array([0.]),
    T = np.array([2.]),
    y0 = np.array([1.])
)
problem_nonatonomous_2 = ODEModel(
    f = lambda t, y: (y + 1) * (5 - 7 * t**2),
    exact_test_solution = lambda t: 3.8 * np.exp(5 * t - 7 * t**3 / 3) - 1,
    t0 = np.array([0.]),
    T = np.array([2.]),
    y0 = np.array([3.])
)


#======================
# 3)  Nonlinear
problem_nonlinear_1 = ODEModel(
    f = lambda t, y: np.sin(t) + y,
    exact_test_solution = lambda t: 3 * np.exp(t) / 2 - np.sin(t) / 2 - np.cos(
   t) / 2 ,
    t0 = np.array([0]),
    T = np.array([3]),
    y0 = np.array([1])
)


problem_nonlinear_2 = ODEModel(
    f = lambda t, y: np.exp(2 * t) / np.exp(y),
    exact_test_solution = lambda t: np.log(np.exp(2 * t) / 2. + np.exp(4) / 2),
    t0 = np.array([2.]),
    T = np.array([5.]),
    y0 = np.array([4.])
)
```

Лістинг 5: ode_models.py

```
import numpy as np


# EXPLICIT METHODS:

def ForwardEuler():
    A = np.array([0])
```

```
    b = np.array([1])
    c = np.array([0])
    return A, b, c


def ExplicitMidpointMethod():
    A = np.array([
        [0, 0],
        [1, 0]
    ])
    b = np.array([1./2., 1./2.])
    c = np.array([0, 1.])
    return A, b, c


def KuttaThirdOrderMethod():
    A = np.array([
        [0, 0, 0],
        [1./2., 0, 0],
        [-1., 2., 0]
    ])
    b = np.array([1./6., 2./3., 1./6.])
    c = np.array([0, 1./2., 1.])
    return A, b, c


# IMPLISIT METHODS:

def GaussLegendreSixOrder():  # order 6
    A = np.array([
        [5/36, 2/9 - np.sqrt(15)/15, 5/36 - np.sqrt(15)/30],
        [ 5/36 + np.sqrt(15)/24, 2/9, 5/36 - np.sqrt(15)/24],
        [ 5/36 + np.sqrt(15)/30, 2/9 + np.sqrt(15)/15, 5/36]
    ])
    b = np.array([5/18, 4/9, 5/18])
    c = np.array([1/2 - np.sqrt(15)/10, 1/2, 1/2 + np.sqrt(15)/10])

    return A, b, c


def CrankNicolsonMethodSecondOrder():  # order 2
    A = np.array([
        [0, 0],
        [1./2., 1./2.]
    ])
    b = np.array([1/2, 1/2])
    c = np.array([0, 1])

    return A, b, c



# Diagonally Implicit Runge Kutta
def DIRKThirdOrder():  # order 4
    A = np.array([
        [1/2, 0, 0, 0],
        [1/6, 1/2, 0, 0],
        [-1/2, -1/2, 1/2, 0],
        [3/2, -3/2, 1/2, 1/2],
    ])
    b = np.array([3/2, -3/2, 1/2, 1/2])
    c = np.array([1/2, 2/3, 1/2, 1])

```

```
    return A, b, c


# Diagonally Implicit  R u n g e Kutta
def DIRKFourOrder():  # order 4
    A = np.array([
        [1/4, 0, 0, 0, 0],
        [1/2, 1/4, 0, 0, 0],
        [17/50, -1/25, 1/4, 0, 0],
        [371/1360, -137/2720, 15/544, 1/4, 0],
        [25/24, -49/48, 125/16, -85/12, 1/4]
    ])
    b = np.array([25/24, -49/48, 125/16, -85/12, 1/4])
    c = np.array([1/4, 3/4, 11/20, 1/2, 1])

    return A, b, c
```

Лістинг 6: butcher_tables.py

```
import math
import numpy as np
from matplotlib import pyplot as plt

def choose_subplot_dimensions(k):
    if k < 4:
        return k, 1
    elif k < 11:
        return math.ceil(k/2), 2
    else:
        # I've chosen to have a maximum of 3 columns
        return math.ceil(k/3), 3


def generate_subplots(k, row_wise=False):
    nrow, ncol = choose_subplot_dimensions(k)
    # Choose your share X and share Y parameters as you wish:
    figure, axes = plt.subplots(nrow, ncol,
                                sharex=True,
                                sharey=False)

    # Check if it's an array. If there's only one plot, it's just an Axes obj
    if not isinstance(axes, np.ndarray):
        return figure, [axes]
    else:
        # Choose the traversal you'd like: 'F' is col-wise, 'C' is row-wise
        axes = axes.flatten(order=('C' if row_wise else 'F'))

        # Delete any unused axes from the figure, so that they don't show
        # blank x- and y-axis lines
        for idx, ax in enumerate(axes[k:]):
            figure.delaxes(ax)

            # Turn ticks on for the last ax in each column, wherever it lands
            idx_to_turn_on_ticks = idx + k - ncol if row_wise else idx + k - 1
            for tk in axes[idx_to_turn_on_ticks].get_xticklabels():
                tk.set_visible(True)

        axes = axes[:k]
        return figure, axes
```

Лістинг 7: plot_tools.py

```
import numpy as np
```

```
from matplotlib import pyplot as plt
from typing import Callable

import ode_solvers

from ode_models import *
from plot_tools import *
from butcher_tables import *



def solve(
    ode_problem: ODEModel,                # differential problem, which we want to solve,
    ode_solver: ode_solvers.ODESolver,  # ODE solver
    method: Callable,                   # Butcher matrix funciton
    tol: float                          # tolerance
):
    A, b, c = method()
    solver = ode_solver(ode_problem, A, b, c, tol)
    return solver.solve()




def solve_ode_test(
    ode_problem: ODEModel,          # differential problem, which we want to solve
    tol=1e-5                        # tolerance
):
    test_explicit_methods = [ExplicitMidpointMethod, KuttaThirdOrderMethod]
    tests_implicit_methods = [GaussLegendreSixOrder, CrankNicolsonMethodSecondOrder]
    tests_diagonally_implicit_methods = [DIRKThirdOrder, DIRKFourOrder]

    test_methods = test_explicit_methods + tests_implicit_methods + tests_diagonally_implicit_methods

    # build exact solution points if exists:
    if ode_problem.exact_test_solution:
        time_points_exact = np.linspace(ode_problem.t0, ode_problem.T, ode_problem.number_of_points_to_discretization)
        exact_solution = ode_problem.exact_test_solution(time_points_exact)

    # create an figure to display plots
    figure, axes = generate_subplots(
        k=len(test_explicit_methods) + len(tests_implicit_methods) + len(tests_diagonally_implicit_methods),
        row_wise=True
    )

    noise = 0.01  # adding some noise in order to look at solution when he very good)

    # EXPLISIT METHODS
    for k, method in enumerate(test_explicit_methods):
        u = solve(
            ode_problem=ode_problem,
            ode_solver=ode_solvers.ExplicitRungeKutta,
            method=method,
            tol=tol
        )
        # plot result:
```

```
58          axes[k].plot(u[:,0], u[:,1] + noise, color='red', label=f"{method.
    __name__}")
59          if ode_problem.exact_test_solution:
60              axes[k].plot(time_points_exact, exact_solution, label="Exact
    solution")
61          axes[k].grid(True)
62          axes[k].set_title("Explicit Runge Kutta")
63          axes[k].legend()
64
65      # IMPLISIT METHODS
66      for k, method in enumerate(tests_implicit_methods):
67          k += len(test_explicit_methods)
68
69          u = solve(
70              ode_problem=ode_problem,
71              ode_solver=ode_solvers.ImplicitRungeKutta,
72              method=method,
73              tol=tol
74          )
75          # plot result:
76          axes[k].plot(u[:,0], u[:,1] + noise, color='red', label=f"{method.
    __name__}")
77          if ode_problem.exact_test_solution:
78              axes[k].plot(time_points_exact, exact_solution, label="Exact
    solution")
79          axes[k].grid(True)
80          axes[k].set_title("Implicit Runge Kutta")
81          axes[k].legend()
82
83      # DIAGONALY IMPLISIT METHODS
84      for k, method in enumerate(tests_diagonally_implicit_methods):
85          k += len(test_explicit_methods + tests_implicit_methods)
86
87          u = solve(
88              ode_problem=ode_problem,
89              ode_solver=ode_solvers.DiagonallyImplicitRungeKutta,
90              method=method,
91              tol=tol
92          )
93          # plot result:
94          axes[k].plot(u[:,0], u[:,1] + noise, color='red', label=f"{method.
    __name__}")
95          if ode_problem.exact_test_solution:
96              axes[k].plot(time_points_exact, exact_solution, label="Exact
    solution")
97          axes[k].grid(True)
98          axes[k].set_title("Diagonally Implicit Runge Kutta")
99          axes[k].legend()
100     figure.canvas.set_window_title("Solution for ode problem")
101     plt.show()
102
103
104
105
106 # Examples:
107
108
109 def example_simple_equetion_scalar_ode():
110     problems = [
111         problem_scalar_1,
112         problem_scalar_2
113     ]
114     # solve problems all avalible methods:
```

```
115     for problem in problems:
116         solve_ode_test(ode_problem=problem)
117 
118 
119 def example_nonautonomous_ode():
120     problems = [
121         problem_nonatonomous_1,
122         problem_nonatonomous_2
123     ]
124     # solve problems all avalible methods:
125     for problem in problems:
126         solve_ode_test(ode_problem=problem)
127 
128 
129 def example_nonlinear_ode_1():
130     solve_ode_test(ode_problem=problem_nonlinear_1)
131 
132 
133 def example_nonlinear_ode_2():
134     solve_ode_test(ode_problem=problem_nonlinear_2)
135 
136 
137 
138 if __name__ == "__main__":
139     example_simple_equetion_scalar_ode()
140     example_nonautonomous_ode()
141     example_nonlinear_ode_1()
142     example_nonlinear_ode_2()
```

Лістинг 8: main.py