МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

ЗВІТ

про виконання лабораторної роботи №3

з теми СИСТЕМИ НЕЛІНІЙНИХ РІВНЯНЬ.

МЕТОД НЬЮТОНА ТА Є-АЛГОРИТМ

з навчальної дисципліни: “Чисельні методи”

Виконав:

студент групи ІР-24

Шийка Андрій

Прийняв:

Сиротюк С. В.

Львів - 2025

Мета роботи: ознайомитися з найпоширенішим ітераційним методом розв’язування систем нелінійних рівнянь – методом Ньютона та екстраполяційним методом – Ɛ-алгоритмом.

Завдання:

№12

$$\begin{cases} x_1^2 + x_2^2 + 0{,}1 - x_1 = 0 \\ 2x_1x_2 - 0{,}1 - x_2 = 0 \end{cases}$$

поч. наближення $x_1^0 = 0$, $x_2^0 = 0$

ε -алгоритм	Стандартний метод Ньютона (оберт.матриці методом Гауса з вибор. гол. ел-тів по по всій матриці)	Метод січних (оберт.матриці методом Гауса з вибор. гол. ел-тів по рядку)

Короткі теоретичні відомості:

1. Системи нелінійних рівнянь

У векторній формі система записується як F(X) = 0. Розв'язок шукають ітераційними методами, локалізувавши корені та обравши початкове наближення X0. Критерієм збіжності є виконання умови малості похибки на суміжних ітераціях.

2. Метод Ньютона

Метод базується на розкладі функцій у ряд Тейлора з відкиданням членів вищих порядків. Ітераційна формула має вигляд:

X^(k) = X^(k-1) - (J^(k-1))^(-1) * F^(k-1),

де J — матриця Якобі, що містить частинні похідні функцій системи. Метод вимагає обернення матриці Якобі на кожному кроці. Збіжність методу квадратична, але залежить від вибору початкового наближення.

3. Метод січних

Цей метод є модифікацією методу Ньютона, де замість точного обчислення похідних використовується скінченно-різницева апроксимація. Матриця Якобі обчислюється як:

J_ij ≈ (f_i(..., x_j + h_j, ...) - f_i(...)) / h_j,

де крок h_j визначається різницею між поточною та попередньою ітерацією. Метод є двокроковим і потребує двох початкових наближень.

4. є-алгоритм (векторний)

Це метод екстраполяції для прискорення збіжності векторних послідовностей. Алгоритм будує таблицю значень за правилом ромба:

e_(k+1)^(j) = e_(k-1)^(j+1) + (e_(k)^(j+1) - e_(k)^(j))^(-1).

Для векторів використовується інверсія Самуельсона: V^(-1) = V / sum(Vi^2). Алгоритм застосовується до послідовності, згенерованої методом простої ітерації X = G(X).

Список ідентифікаторів констант, змінних, функцій:

Константи:

- EPSILON_PERCENT: задана точність обчислень (1e-5).

- Q_VAL, P_VAL: параметри для налаштування є-алгоритму.

Змінні:

- x, x_new, x_old: масиви для зберігання поточного та попередніх наближень вектора невідомих.

- J: двовимірний масив, матриця Якобі.

- F: масив значень функцій системи в точці x.

- e: тривимірний масив для зберігання таблиці є-алгоритму.

- pivot_type: рядок-прапорець для вибору стратегії в методі Гауса ('full' або 'row').

Функції:

- equations_F(x): повертає вектор значень лівих частин рівнянь.

- jacobian_analytic(x): повертає матрицю точних частинних похідних.

- iteration_function_G(x): перетворює систему до вигляду x=g(x) для є-алгоритму.

- solve_gauss(A, B, type): універсальна функція розв'язку СЛАР методом Гауса.

- method_epsilon_algorithm(...): основна процедура є-алгоритму.

- method_newton_standard(...): основна процедура методу Ньютона.

- method_secant(...): основна процедура методу січних.

Код програми:

```
import copy


EPSILON_PERCENT = 1e-5  # Задана відносна похибка у %
X0 = [0.0, 0.0]         # Початкове наближення
Q_VAL = 2               # Параметр q для е-алгоритму
P_VAL = 2               # Параметр p для е-алгоритму


def equations_F(x):
    """
    Вектор функцій F(X).
    f1 = x1^2 + x2^2 + 0.1 - x1
    f2 = 2*x1*x2 - 0.1 - x2
    """
    f1 = x[0]**2 + x[1]**2 + 0.1 - x[0]
    f2 = 2*x[0]*x[1] - 0.1 - x[1]
    return [f1, f2]


def jacobian_analytic(x):
    """
    Аналітична матриця Якобі для методу Ньютона.
    df1/dx1 = 2*x1 - 1;  df1/dx2 = 2*x2
    df2/dx1 = 2*x2;      df2/dx2 = 2*x1 - 1
```

```
    """
    n = len(x)
    J = [[0.0]*n for _ in range(n)]

    J[0][0] = 2*x[0] - 1
    J[0][1] = 2*x[1]

    J[1][0] = 2*x[1]
    J[1][1] = 2*x[0] - 1

    return J

def iteration_function_G(x):
    """
    Перетворення F(X)=0 -> X=G(X) для методу простої ітерації (та
e-алгоритму).
    Потрібно виразити x1, x2 так, щоб забезпечити збіжність (|G'| < 1).

    З рівняння 1: x1 = x1^2 + x2^2 + 0.1
    З рівняння 2: 2*x1*x2 - x2 = 0.1 => x2*(2*x1 - 1) = 0.1 => x2 = 0.1
/ (2*x1 - 1)
    """
    new_x = [0.0] * len(x)

    # x1 = g1(x)
    new_x[0] = x[0]**2 + x[1]**2 + 0.1

    # x2 = g2(x)
    denom = 2*x[0] - 1
    if abs(denom) < 1e-14: denom = 1e-14 # Захист від ділення на нуль
    new_x[1] = 0.1 / denom

    return new_x


def solve_gauss(matrix_A, vector_B, pivot_type):
```

```
    """
    Розв'язування СЛАР Ax = B методом Гауса.
    pivot_type згідно варіанту:
      'full' - вибір головного елемента по всій матриці (Метод Ньютона).
      'row'  - вибір головного елемента по рядку (Метод Січних).
    """
    n = len(vector_B)
    A = copy.deepcopy(matrix_A)
    B = copy.deepcopy(vector_B)

    # Масив для збереження порядку змінних (потрібен при перестановці
стовпців)
    col_order = list(range(n))

    # Прямий хід
    for k in range(n):
        pivot_row = k
        pivot_col = k
        max_val = 0.0

        if pivot_type == 'full':
            # Пошук max по всій підматриці A[k:n, k:n]
            for i in range(k, n):
                for j in range(k, n):
                    if abs(A[i][j]) > abs(max_val):
                        max_val = A[i][j]
                        pivot_row = i
                        pivot_col = j

            # Перестановка рядків
            A[k], A[pivot_row] = A[pivot_row], A[k]
            B[k], B[pivot_row] = B[pivot_row], B[k]

            # Перестановка стовпців
            for row in range(n):
```

```
                A[row][k], A[row][pivot_col] = A[row][pivot_col],
A[row][k]
            # Запам'ятовуємо зміну порядку змінних
            col_order[k], col_order[pivot_col] = col_order[pivot_col],
col_order[k]

        elif pivot_type == 'row':
            # Пошук max тільки в поточному рядку k серед стовпців j >= k
            for j in range(k, n):
                if abs(A[k][j]) > abs(max_val):
                    max_val = A[k][j]
                    pivot_col = j

            # Перестановка стовпців (щоб max елемент рядка став на
діагональ)
            for row in range(n):
                A[row][k], A[row][pivot_col] = A[row][pivot_col],
A[row][k]
            col_order[k], col_order[pivot_col] = col_order[pivot_col],
col_order[k]

            # Перевірка на 0 на діагоналі після перестановки
            if abs(A[k][k]) < 1e-14:
                 return None

        else:
            # Стандартний (по стовпцю) - не використовується у цьому
варіанті
            pass

        # Нормування
        pivot = A[k][k]
        if abs(pivot) < 1e-14: return None # Матриця вироджена

        for j in range(k, n):
            A[k][j] /= pivot
        B[k] /= pivot
```

```
        # Виключення
        for i in range(k + 1, n):
            factor = A[i][k]
            for j in range(k, n):
                A[i][j] -= factor * A[k][j]
            B[i] -= factor * B[k]

    # Зворотний хід
    X_temp = [0.0] * n
    for i in range(n - 1, -1, -1):
        sum_ax = sum(A[i][j] * X_temp[j] for j in range(i + 1, n))
        X_temp[i] = B[i] - sum_ax

    # Відновлення порядку змінних (якщо були перестановки стовпців)
    X_final = [0.0] * n
    for i in range(n):
        # X_temp[i] відповідає змінній з індексом col_order[i]
        X_final[col_order[i]] = X_temp[i]

    return X_final

def check_convergence_relative(x_new, x_old, epsilon_percent):
    """
    Перевірка умови збіжності: |(xi - xi_old)/xi| * 100% < epsilon
    """
    for i in range(len(x_new)):
        numerator = abs(x_new[i] - x_old[i])
        denominator = abs(x_new[i]) if abs(x_new[i]) > 1e-14 else 1.0

        # Якщо значення дуже близьке до 0, використовуємо абсолютну
похибку
        if abs(x_new[i]) < 1e-14:
            if numerator * 100.0 >= epsilon_percent: return False
        else:
```

```
            if (numerator / denominator) * 100.0 >= epsilon_percent:
                return False
    return True

# --- 3. РЕАЛІЗАЦІЯ МЕТОДІВ ---

def method_epsilon_algorithm(x0, eps_pct, p, q):
    """
    Векторний epsilon-алгоритм.
    Використовує ітераційну формулу X = G(X) для генерації послідовності Sn,
    потім застосовує правило ромба (Wynn's epsilon algorithm) з інверсією Самельсона.
    """
    print(f"\n=== 1. E-алгоритм (q={q}, p={p}, eps={eps_pct}%) ===")
    m = len(x0)
    n_seq = 2 * q + 1  # Кількість членів послідовності Sn (якщо q=m=2, то n=5)

    # 3D масив e[j][k][i], де:
    # j - індекс зсуву послідовності
    # k - індекс epsilon-стовпця (0 відповідає k=-1 в теорії, 1 -> k=0 (Sn), 2 -> k=1...)
    # i - компонент вектора (0..m-1)
    # Розмір: [n_seq+1] рядків, [n_seq+1] стовпців, [m] компонент
    e = [[[0.0]*m for _ in range(n_seq + 1)] for _ in range(n_seq + 50)]

    current_x = list(x0)

    # 1. Виконуємо p початкових ітерацій (без запису в таблицю)
    for _ in range(p):
        current_x = iteration_function_G(current_x)

    iteration_count = 0
    MAX_ITERS = 50

    while iteration_count < MAX_ITERS:
```

```
        iteration_count += 1

        # 2. Ініціалізація стовпця k=-1 нулями (індекс k_arr=0)
        for j in range(n_seq + 1):
            for i in range(m):
                e[j][0][i] = 0.0

        # 3. Заповнення стовпця k=0 (Sn) (індекс k_arr=1)
        # Перший елемент S0 - це поточне наближення після p ітерацій
        s_temp = list(current_x)
        for i in range(m):
            e[0][1][i] = s_temp[i]

        # Генеруємо S1...Sn
        converged_early = False
        for j in range(1, n_seq + 1):
            s_prev = list(s_temp)
            s_temp = iteration_function_G(s_temp) # Наступний член
послідовності

            for i in range(m):
                e[j][1][i] = s_temp[i]

            # Перевірка збіжності на етапі генерації (якщо Sn вже
зійшлося)
            if j == 1 and check_convergence_relative(s_temp, s_prev,
eps_pct):
                print(f"Збіжність досягнута на етапі генерації Sn.")
                print(f"Результат: {s_temp}")
                print(f"Дельта: {equations_F(s_temp)}")
                return s_temp

        # 4. Екстраполяція (заповнення таблиці)
        # k_arr йде від 1 (відповідає Sn) до n_seq
        # Формула: e[j][k+1] = e[j+1][k-1] + (e[j+1][k] - e[j][k])^-1
```

```
        for k_arr in range(1, n_seq): # k_arr - це поточний стовпець, обчислюємо k_arr+1
            for j in range(n_seq - k_arr):
                # Різниця сусідніх елементів у стовпці: V = e[j+1][k] - e[j][k]
                diff_V = [e[j+1][k_arr][i] - e[j][k_arr][i] for i in range(m)]

                # Інверсія Самельсона: V^(-1) = V / sum(Vi^2)
                norm_sq = sum(v*v for v in diff_V)
                if norm_sq < 1e-25: norm_sq = 1e-25

                inv_V = [v / norm_sq for v in diff_V]

                # Обчислення наступного стовпця
                for i in range(m):
                    e[j][k_arr+1][i] = e[j+1][k_arr-1][i] + inv_V[i]

        # Результат екстраполяції знаходиться в вершині ромба e[0][n_seq][...] (індекс k_arr = n_seq)
        extrapolated_x = [e[0][n_seq][i] for i in range(m)]

        print(f"Цикл {iteration_count}: Екстрапольований X = {[round(val, 6) for val in extrapolated_x]}")

        # Умова завершення: порівняння екстрапольованого результату з початковим для цього циклу
        if check_convergence_relative(extrapolated_x, current_x, eps_pct):
            print(f"Результат знайдено (e-алгоритм): {extrapolated_x}")
            print(f"Дельта F(X): {equations_F(extrapolated_x)}")
            return extrapolated_x

        current_x = extrapolated_x # Нове наближення для наступного циклу

    print("Перевищено ліміт ітерацій e-алгоритму.")
    return current_x
```

```python
def method_newton_standard(x0, eps_pct):
    """
    Стандартний метод Ньютона.
    X(k) = X(k-1) - J^(-1) * F(X(k-1))
    Обернення J виконується методом Гауса з вибором головного елемента
ПО ВСІЙ МАТРИЦІ.
    """
    print(f"\n=== 2. Метод Ньютона (Full Pivot, eps={eps_pct}%) ===")
    x = list(x0)
    k_iter = 0
    MAX_ITERS = 100

    while k_iter < MAX_ITERS:
        k_iter += 1

        F_val = equations_F(x)
        J_mat = jacobian_analytic(x)

        # СЛАР: J * dX = -F
        minus_F = [-val for val in F_val]

        # Виклик Гауса з 'full' pivoting
        delta_x = solve_gauss(J_mat, minus_F, pivot_type='full')

        if delta_x is None:
            print("Матриця Якобі вироджена. Метод зупинено.")
            return x

        x_new = [x[i] + delta_x[i] for i in range(len(x))]

        print(f"Ітерація {k_iter}: X = {[round(val, 8) for val in
x_new]}")

        if check_convergence_relative(x_new, x, eps_pct):
            print(f"Збіжність досягнута. Результат: {x_new}")
```

```
            print(f"Дельта F(X): {equations_F(x_new)}")
            return x_new

        x = x_new

    print("Перевищено ліміт ітерацій Ньютона.")
    return x


def method_secant(x0, eps_pct):
    """
    Метод січних (двокроковий).
    Використовує кінцево-різницеву матрицю Якобі.
    Обернення J виконується методом Гауса з вибором головного елемента ПО РЯДКУ.
    """
    print(f"\n=== 3. Метод Січних (Row Pivot, eps={eps_pct}%) ===")
    m = len(x0)

    # Для методу січних потрібно два наближення: X^(k) та X^(k-1).
    # Оскільки дано тільки одне (0,0), друге генеруємо штучно з малим зміщенням.
    x_older = list(x0)                  # X^(0)
    x_old = [val + 0.01 for val in x0] # X^(1) - штучне

    k_iter = 0
    MAX_ITERS = 100

    while k_iter < MAX_ITERS:
        k_iter += 1

        # Вектор кроку h = x_old - x_older
        h = [x_old[i] - x_older[i] for i in range(m)]

        # Обчислення кінцево-різницевої матриці Якобі (по стовпцях)
        # J_ij = (fi(x_old + h_vector_only_j) - fi(x_old)) / h_j
        J_diff = [[0.0]*m for _ in range(m)]
```

```
        F_old = equations_F(x_old)

        for j in range(m):
            # Формуємо тимчасовий вектор x_tilde, де змінено лише j-ту компоненту
            x_tilde = list(x_old)
            step = h[j] if abs(h[j]) > 1e-14 else 1e-5 # Захист від 0 кроку
            x_tilde[j] += step

            F_tilde = equations_F(x_tilde)

            for i in range(m):
                J_diff[i][j] = (F_tilde[i] - F_old[i]) / step

        # Розв'язуємо J * delta_x = -F(x_old)
        minus_F = [-val for val in F_old]

        # Виклик Гауса з 'row' pivoting
        delta_x = solve_gauss(J_diff, minus_F, pivot_type='row')

        if delta_x is None:
             print("Матриця Якобі (різницева) вироджена.")
             return x_old

        x_new = [x_old[i] + delta_x[i] for i in range(m)]

        print(f"Ітерація {k_iter}: X = {[round(val, 8) for val in x_new]}")

        if check_convergence_relative(x_new, x_old, eps_pct):
             print(f"Збіжність досягнута. Результат: {x_new}")
             print(f"Дельта F(X): {equations_F(x_new)}")
             return x_new

        # Зсув історії наближень
```

```
        x_older = x_old
        x_old = x_new

    print("Перевищено ліміт ітерацій методу січних.")
    return x_old

if __name__ == "__main__":
    method_epsilon_algorithm(X0, EPSILON_PERCENT, P_VAL, Q_VAL)
    method_newton_standard(X0, EPSILON_PERCENT)
    method_secant(X0, EPSILON_PERCENT)
```

Результат:

```
❯ python lab3/lab3.py

=== 1. E-алгоритм (q=2, p=2, eps=1e-05%) ===
Цикл 1: Екстрапольований X = [0.138135, -0.138158]
Цикл 2: Екстрапольований X = [0.138197, -0.138197]
Збіжність досягнута на етапі генерації Sn.
Результат: [0.138196600782586, -0.13819660085758037]
Дельта: [1.7386483919246132e-10, -9.887046736878347e-11]

=== 2. Метод Ньютона (Full Pivot, eps=1e-05%) ===
Ітерація 1: X = [0.1, -0.1]
Ітерація 2: X = [0.13333333, -0.13333333]
Ітерація 3: X = [0.13809524, -0.13809524]
Ітерація 4: X = [0.13819656, -0.13819656]
Ітерація 5: X = [0.1381966, -0.1381966]
Ітерація 6: X = [0.1381966, -0.1381966]
Збіжність досягнута. Результат: [0.13819660112501053, -0.13819660112501053]
Дельта F(X): [0.0, 0.0]

=== 3. Метод Січних (Row Pivot, eps=1e-05%) ===
Ітерація 1: X = [0.09958105, -0.10021263]
Ітерація 2: X = [0.14383767, -0.1359453]
Ітерація 3: X = [0.13784224, -0.138097]
Ітерація 4: X = [0.13819328, -0.13819524]
Ітерація 5: X = [0.1381966, -0.1381966]
Ітерація 6: X = [0.1381966, -0.1381966]
Збіжність досягнута. Результат: [0.13819660112499765, -0.1381966011250056]
Дельта F(X): [7.93809462606987e-15, 0.0]
```

Висновок:

У ході виконання лабораторної роботи було програмно реалізовано та досліджено методи розв'язування системи нелінійних рівнянь згідно варіанту 12.

1. Стандартний метод Ньютона з використанням методу Гауса (вибір головного елемента по всій матриці) продемонстрував високу швидкість збіжності (найменшу кількість ітерацій), що підтверджує теоретичну квадратичну збіжність методу при вдалому виборі початкового наближення.
2. Метод січних, реалізований з використанням скінченно-різницевої матриці Якобі та методу Гауса (вибір по рядку), також успішно знайшов розв'язок, але потребував більшої кількості обчислень функцій на кожній ітерації для побудови матриці.
3. є-алгоритм дозволив знайти розв'язок шляхом екстраполяції послідовності простих ітерацій. Успішність цього методу залежала від коректного перетворення системи рівнянь до вигляду X=G(X), що задовольняє умову стискаючого відображення.

Всі три методи дали однаковий вектор розв'язку в межах заданої точності ε=10^-5%.