МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНОМУ УНІВЕРСИТЕТІ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”

Кафедра систем штучного інтелекту

Лабораторна робота №3

з дисципліни

«Чисельні методи»

Виконав:
Студент групи ШІ-22
Михальчук Антон
Перевірила:
доцент кафедри СШІ
Гентош Леся Ігорівна

Львів-2025

Лабораторна робота № 3.

Тема: Ітераційні методи розв’язування систем лінійних алгебраїчних рівнянь.
Мета: набути навиків практичного використання ітераційних методів розв’язування СЛАР: методу Якобі, методу Зейделя, методу верхньої релаксації.

Варіант 15

Постановка завдання

• Скласти програму, яка реалізує знаходження розв’язку СЛАР за допомогою методу Якобі.

• Перевірити виконання наданних завдань.

• Перевірити виконання для матриць n=10000.

• Перевірити на достовірність, точність та стійкість.

Аналіз чисельних методів

Перше завдання

input1.txt

```
4
15 -4 -3  8   2
-4 10 -4  2 -12
-3 -4 10  2  -4
 8  2  2 12   6
```

Розв’язок:

input2.txt

```
3
3 -3 4 1
-3 -1 0 2
4 0 -4 3
```

Оцінка:

evaluation1.txt

```
Method name: solve_jacobi
Machine error for IEEE 754 standard ε ≈ 2.2e-16
Matrix size: 4
Execution time: 0.22156286239624023
Iterations: 1449
Converged: True
Epsilon used: 1e-14

spectral_radius: 0.9801633898171331
cond: 93.5485000146621
residual_norm: 2.4474871185953987e-12
relative_residual_norm: 2.0357247988674054e-15
a_priori_iterations_estimate: 3387
diagonal_dominance: True

Norms of iteration matrix C:
  1-norm (col-sum): 1.3966666666666667
  2-norm (spectral): 1.0224784298849021
  inf-norm (row-sum): 0.99

Benchmark solution available.
Absolute error vs benchmark: 1.0267405226489306e-11
Relative error vs benchmark: 2.407924799192832e-13
Stability error (solving perturbed system): 5.2372955844671635e-08

End of evaluation.
```

Друге завдання

input2.txt

```
3
3 -3 4 1
-3 -1 0 2
4 0 -4 3
```

Результат:

evaluation2.txt output2.txt

```
-0.125000000000031641356201816961402073502540588378906 25
-1.624999999999944266804163817141670733690261840820312 5
-0.875000000000001443289932012703502550721168518066406 25
```

Оцінка:

evaluation2.txt

```
Method name: solve_jacobi
Machine error for IEEE 754 standard ε ≈ 2.2e-16
Matrix size: 3
Execution time: 0.060114383697509766
Iterations: 184
Converged: True
Epsilon used: 1e-14

spectral_radius: 0.8469204725567879
cond: 3.5051002879120627
residual_norm: 2.963566048481161e-13
relative_residual_norm: 1.837704776599161e-14
a_priori_iterations_estimate: 1100
diagonal_dominance: True

Norms of iteration matrix C:
  1-norm (col-sum): 1.1636363636363636
  2-norm (spectral): 0.9322213721575836
  inf-norm (row-sum): 0.9696969696969697

Benchmark solution available.
Absolute error vs benchmark: 6.410497349061475e-14
Relative error vs benchmark: 3.4654501383133245e-14
Stability error (solving perturbed system): 3.4177662482722554e-09

End of evaluation.
```

Отримані результати показують, що, навіть, при початковому невиконанні умов збіжності, за допомогою еквівалентних перестановок можна отримати діагональну перевагу, проте це є можливим лише для матриць з розмірністю до 4, через суперекспоненційну складність алоритму перестановок.

Обчислення для n=10000

Завдяки вбудованим можливостям генерування випадкових чисел у бібліотеці Numpy, було створено матрицю розміру 10000 з діагональним переваженням, що дало хороші початкові умови: мале число зумовленості, менші 1 спектральний радіус та норми.

evaluation3.txt

```
Method name: solve_jacobi
Machine error for IEEE 754 standard ε ≈ 2.2e-16
Matrix size: 10000
Execution time: 688.4082899093628
Iterations: 6
Converged: True
Epsilon used: 1e-14

spectral_radius: 0.007270875676247319
cond: 1.0501832712435737
residual_norm: 2.2691472260197968e-09
relative_residual_norm: 1.887388597971707e-12
a_priori_iterations_estimate: 48
diagonal_dominance: True

Norms of iteration matrix C:
  1-norm (col-sum): 0.6384462993931148
  2-norm (spectral): 0.01441673123976976
  inf-norm (row-sum): 0.6250000000000002

Benchmark solution available.
Absolute error vs benchmark: 1.0519718767796718e-14
Relative error vs benchmark: 1.458604253617073e-13
Stability error (solving perturbed system): 1.2668501448013966e-10

End of evaluation.
```

Аналіз на достовірність:

Абсолютні та відносні похибки були обчислені через другу норму різниці між нашим розв’язком та еталонним розв’язком (benchmark solution):

Абсолютна похибка: 1.051e-14

Відносна похибка: 1.458e-13

Такі надзвичайно низькі похибки, що наближаються до машинного епсилон (2.2e-16), свідчать про високу достовірність та правильність отриманого чисельного розв’язку.

Аналіз на точність:

Відносні похибки знаходяться на рівні очікуваної високої точності, що повністю узгоджується з винятково низьким числом обумовленості матриці (`cond(A) ≈ 1.05`).

Теоретична межа похибки є надзвичайно низькою:

$\delta_a \approx \mathrm{cond}(A) \cdot \varepsilon_{\mathrm{mach}} \approx 1.05 \cdot 2.22 \cdot 10^{-16} \approx 2.31 \cdot 10^{-16}$

Отримана відносна похибка розв’язку (`1.458e-13`) є вкрай малою і підтверджує чудову обумовленість задачі.

Відносна норма нев’язки також є дуже малою:

$\frac{||Ax-b||}{||A||\cdot||b||} \approx 1.887 \cdot 10^{-12}$

Це підтверджує, що метод Якобі (завдяки швидкій збіжності, `6` ітерацій) дав високоточний розв’язок.

Аналіз на стійкість:

Стійкість оцінювалася як друга норма різниці між нашим розв’язком та розв’язком для модифікованих вхідних даних (з доданим малим шумом). Похибка стійкості склала:

$\mathrm{Stability\ error} \approx 1.266 \cdot 10^{-10}$

Різниця між розв’язками (приблизно в 10-му знаку після коми) демонструє, що метод зберігає високу числову стійкість щодо невеликих варіацій у вхідних даних. Це очікуваний результат, оскільки низьке число обумовленості матриці `cond(A) ≈ 1.05` свідчить про те, що задача є дуже стійкою.

Висновки

Було реалізовано метод Якобі. При невиконанні необхідної умови (спектральний радіус менше 1), було задіяно метод еквівалентних перестановок, що працює лише з матрицями розмірністю менше 5. Якщо ж необхідна умова виконується, метод Якобі виявляється вигіднішим за інші прямі методи, адже дає можливість отримати достатньо точний результат за кілька ітерацій. Як у нашому випадку з 6 ітераціями для матриці розмірністю 10000.

Код програмної реалізації

```
--- FILE: create_inputs.py ---
import numpy as np
from pathlib import Path

n = 1024
ZERO_PERCENTAGE = 0
INPUT_ID = 5
GENERATE_RIGHT_MATRIX = True

def generate_strictly_diagonally_dominant(n, alpha=1.5, offdiag_range=(-1.0, 1.0), b_range=(-10, 10)):
    A = np.random.uniform(offdiag_range[0], offdiag_range[1], size=(n, n))
    np.fill_diagonal(A, 0.0)
    off_diag_sums = np.sum(np.abs(A), axis=1)
    diag_vals = off_diag_sums * alpha + np.finfo(float).eps * np.ones(n)
    signs = np.where(np.random.rand(n) < 0.5, -1.0, 1.0)
    diag_vals = diag_vals * signs
    np.fill_diagonal(A, diag_vals)
    b = np.random.uniform(b_range[0], b_range[1], size=n)
    return A, b

if GENERATE_RIGHT_MATRIX:
    alpha = 1.6
    matrix_A, b = generate_strictly_diagonally_dominant(n, alpha=alpha)
    matrix = np.hstack([matrix_A, b.reshape(-1, 1)])
else:
    random_mask = np.random.rand(n, n + 1)
    zero_mask = random_mask < ZERO_PERCENTAGE
    matrix = np.random.uniform(-1000, 1000, size=(n, n + 1))
    matrix[zero_mask] = 0

output_dir = Path("..") / "inputs"
output_dir.mkdir(parents=True, exist_ok=True)

if n > 1000:
    file_path = output_dir / f"input{INPUT_ID}.npz"
    np.savez_compressed(file_path, a=matrix[:, :n], b=matrix[:, n])
    print(f"Matrices saved to file {file_path}")
else:
    file_path = output_dir / f"input{INPUT_ID}.txt"
    with open(file_path, 'w') as f:
        f.write(f"{n}\n")
        np.savetxt(f, matrix)
```

```
    print(f"Matrix saved to file {file_path}")

--- FILE: data_handler.py ---
import numpy as np
from pathlib import Path
import os

def read_sole_data(input_id):
    input_dir = Path("..") / "inputs"
    input_dir.mkdir(parents=True, exist_ok=True)
    npz_path = input_dir / f"input{input_id}.npz"
    txt_path = input_dir / f"input{input_id}.txt"

    try:
        if os.path.exists(npz_path):
            print(f"Reading data from file {npz_path}...")
            data = np.load(npz_path)
            if 'a' in data and 'b' in data:
                a = data['a']
                b = data['b']
                print("Data loaded successfully.")
                return a, b
            else:
                print("Error: NPZ file does not contain keys 'a' and 'b'.")
                return None, None

        elif os.path.exists(txt_path):
            print(f"Reading data from file {txt_path}...")
            full_data = np.genfromtxt(txt_path, skip_header=1, delimiter=None, filling_values=0)
            if full_data.ndim < 2:
                print("Error: Not enough data for matrix and vector.")
                return None, None
            a = full_data[:, :-1].copy()
            b = full_data[:, -1].copy()
            if a.shape[0] != a.shape[1] or a.shape[0] != b.shape[0]:
                print("Error: dimensions do not match.")
                return None, None
            print("Data loaded successfully.")
            return a, b

        else:
            print("Error: Data file not found.")
            return None, None
```

```
    except Exception as error:
        print(f"Error reading file: {error}")
        return None, None

def method_evaluation(a, x, b, evals, eval_options, execution_time, input_id, method):
    evaluation_dir = Path("..") / "evaluations"
    evaluation_dir.mkdir(parents=True, exist_ok=True)
    txt_path = evaluation_dir / f"evaluation{input_id}.txt"

    with open(txt_path, 'w') as f:
        f.write(f"Method name: {getattr(method, '__name__', str(method))}\n")
        f.write(f"Machine error for IEEE 754 standard ε ≈ {2.2e-16}\n")
        f.write(f"Matrix size: {evals.get('matrix_size', a.shape[0])}\n")
        f.write(f"Execution time: {execution_time}\n")
        f.write(f"Iterations: {evals.get('iterations', 'N/A')}\n")
        f.write(f"Converged: {evals.get('converged', 'N/A')}\n")
        f.write(f"Epsilon used: {evals.get('epsilon', 'N/A')}\n\n")

        scalar_keys = ['spectral_radius', 'cond', 'residual_norm', 'relative_residual_norm',
                       'a_priori_iterations_estimate', 'diagonal_dominance']
        for key in scalar_keys:
            if key in evals:
                f.write(f"{key}: {evals[key]}\n")

        if 'norms' in evals and isinstance(evals['norms'], dict):
            f.write("\nNorms of iteration matrix C:\n")
            for nkey, nval in evals['norms'].items():
                f.write(f"  {nkey}: {nval}\n")

        if eval_options.get('benchmark', True):
            try:
                x_benchmark = np.linalg.solve(a, b)
                abs_err = float(np.linalg.norm(x - x_benchmark))
                rel_err = float(abs_err / np.linalg.norm(x_benchmark)) if np.linalg.norm(x_benchmark) != 0 else
None
                f.write("\nBenchmark solution available.\n")
                f.write(f"Absolute error vs benchmark: {abs_err}\n")
                f.write(f"Relative error vs benchmark: {rel_err}\n")
            except Exception:
                f.write("\nBenchmark solution could not be computed (singular or unstable matrix).\n")

        if eval_options.get('stability_error', True) and x is not None:
            try:
                epsilon = 1e-8
```

```
            b_perturbed = b + epsilon * np.random.randn(*b.shape)
            x_perturbed, _ = method(a, b_perturbed)
            stability_error = float(np.linalg.norm(x_perturbed - x))
            f.write(f"Stability error (solving perturbed system): {stability_error}\n")
        except Exception:
            f.write("Stability error: Could not be calculated due to singularity or instability.\n")

    f.write("\nEnd of evaluation.\n")

  print(f"Evaluation complete. Results saved to {txt_path}")

def save_solution(x, input_id, decimal_places):
  output_dir = Path("..") / "outputs"
  output_dir.mkdir(parents=True, exist_ok=True)
  npz_path = output_dir / f"output{input_id}.npz"
  txt_path = output_dir / f"output{input_id}.txt"

  if x is not None:
    n = len(x)
    if n > 1000:
      np.savez_compressed(npz_path, x=x)
      print(f"Solution saved to {npz_path}")
    else:
      np.savetxt(txt_path, x, fmt=f'%.{decimal_places}g')
      print(f"Solution saved to {txt_path}")
  else:
    with open(txt_path, 'w') as f:
      f.write("Matrix is singular.")
    print(f"Matrix is singular. Saved to {txt_path}")

--- FILE: jacobi.py ---
import numpy as np
import warnings
from evaluations import *

warnings.filterwarnings('ignore', category=RuntimeWarning)

def solve_jacobi(A_orig, b_orig, eps=1e-10, max_iter_est=100_000, eval_options=None):
  if eval_options is None:
    eval_options = {
      'cond': True,
      'spectral_radius': True,
      'norms': True,
      'residual': True,
```

```
        'benchmark': True,
        'iterations': True,
        'a_priori': True,
        'stability_error': True
    }


A = A_orig.copy()
b = b_orig.copy()
evals = {}
n = A.shape[0]
col_perm = list(range(n))
k_est = np.inf
scales = np.ones(n)

print(f"\n======== Starting Jacobi Solver for {n}x{n} System ========")
print(f"Target Epsilon (eps): {eps}")

diag_dom = is_strictly_diagonally_dominant(A)
evals['diagonal_dominance'] = bool(diag_dom)
print("Initial diagonal dominance:", diag_dom)

if not diag_dom:
    A_new, b_new, success, col_perm, scales = try_make_diagonally_dominant(A.copy(), b.copy())
    evals['reordered_for_dominance'] = bool(success)
    if success:
        A, b = A_new, b_new
        diag_dom = is_strictly_diagonally_dominant(A)
        evals['diagonal_dominance'] = diag_dom
        print("Using reordered system for Jacobi.")
    else:
        print("WARNING: Matrix is not diagonally dominant and reordering failed. Convergence not
guaranteed.")

diag = np.diag(A).astype(float)
if np.any(np.isclose(diag, 0.0)):
    raise ValueError("Zero (or near-zero) diagonal element detected; cannot apply Jacobi.")

C, d = compute_iteration_matrix(A, b)
print("\nCalculated Iteration Matrix C and Vector d.")

if eval_options.get('spectral_radius', True):
    rho = get_spectral_radius(C)
    evals['spectral_radius'] = rho
```

```
if eval_options.get('norms', True):
    norms = get_norms(C)
    evals['norms'] = norms

if eval_options.get('cond', True):
    try:
        cond = float(np.linalg.cond(A_orig))
        evals['cond'] = cond
    except np.linalg.LinAlgError:
        evals['cond'] = None

if eval_options.get('a_priori', True):
    a_priori_iters = get_a_priori(C, d, eps)
    evals['a_priori_iterations_estimate'] = a_priori_iters
    k_est = a_priori_iters if (a_priori_iters is not None) else max_iter_est

k_max = int(min(k_est if np.isfinite(k_est) else max_iter_est, max_iter_est))

x = d.copy()
converged = False
k = 0

print("\n--- Starting Iteration Process ---")
try:
    for k in range(1, k_max + 1):
        x_new = C @ x + d
        diff_norm = np.linalg.norm(x_new - x, ord=np.inf)

        if diff_norm < eps:
            x = x_new
            converged = True
            break
        x = x_new
    else:
         print(f"Did not converge after max iterations ({k_max}).")

except RuntimeWarning:
    print("WARNING: Runtime error occurred during iteration.")

evals['iterations'] = int(k)
evals['converged'] = bool(converged)
print("--- Iteration Process Finished ---")
```

```
    x_scaled = x * scales
    x_final = np.zeros_like(x)
    x_final[col_perm] = x_scaled

    if eval_options.get('residual', True):
        evals['residual_norm'], evals['relative_residual_norm'] = get_residual(A_orig, x_final, b_orig)

    evals['epsilon'] = float(eps)
    evals['matrix_size'] = int(n)

    return x_final, evals

--- FILE: main.py ---
import warnings
import time
import numpy as np
from jacobi import solve_jacobi
from data_handler import read_sole_data, method_evaluation, save_solution

np.seterr(divide='raise', invalid='raise')
warnings.simplefilter('error', RuntimeWarning)

INPUT_ID = 2
DECIMAL_PLACES = 60
METHOD = solve_jacobi
EPSILON = 10e-15

EVAL_OPTIONS = {
    'cond': True,
    'spectral_radius': True,
    'norms': True,
    'residual': True,
    'benchmark': True,
    'iterations': True,
    'a_priori': True,
    'stability_error': True
}

def main():
    a, b = read_sole_data(INPUT_ID)
    if a is None or b is None:
        return

    start_time = time.time()
```

```
    x, evals = METHOD(a, b, eps=EPSILON, eval_options=EVAL_OPTIONS)
    end_time = time.time()
    execution_time = end_time - start_time

    method_evaluation(a, x, b, evals, EVAL_OPTIONS, execution_time, INPUT_ID, METHOD)

    save_solution(x, INPUT_ID, DECIMAL_PLACES)

if __name__ == '__main__':
    main()

--- FILE: evaluations.py ---
import numpy as np
from itertools import permutations, product
from typing import Tuple, List


def apply_col_scaling(A: np.ndarray, scales: Tuple[float, ...]) -> np.ndarray:
    S = np.diag(scales)
    return A.dot(S)


def is_strictly_diagonally_dominant(M: np.ndarray) -> bool:
    diag = np.abs(np.diag(M))
    off = np.sum(np.abs(M), axis=1) - diag
    return np.all(diag > off)


def try_make_diagonally_dominant(A: np.ndarray, b: np.ndarray, max_scale: int = 12) -> Tuple[
    np.ndarray, np.ndarray, bool, List[int], np.ndarray]:
    print("--- Attempting to make matrix A diagonally dominant ---")
    n = A.shape[0]
    default_scales = np.ones(n, dtype=float)

    if is_strictly_diagonally_dominant(A):
        print("Matrix already strictly diagonally dominant")
        return A.copy(), b.copy(), True, list(range(n)), default_scales

    scale_values = list(range(1, max_scale + 1))

    if n == 3:
        iterator = product(scale_values, repeat=3)
    elif n <= 4:
        print(f"WARNING: n={n}. Scale search will be very slow ({len(scale_values) ** n} combinations).")
```

```
        iterator = product(scale_values, repeat=n)
    else:
        print(f"WARNING: n={n}. Skipping scale search, trying permutations only.")
        iterator = [tuple(np.ones(n, dtype=int))]

    for row_perm in permutations(range(n)):
        A_row = A[list(row_perm), :]
        b_row = b[list(row_perm)]

        for col_perm in permutations(range(n)):
            A_perm = A_row[:, list(col_perm)]

            if is_strictly_diagonally_dominant(A_perm):
                print(«  Found diagonally dominant form (permutations only)")
                return A_perm, b_row, True, list(col_perm), default_scales

            if n <= 4:
                scale_iterator = product(scale_values, repeat=n) if n > 3 else iterator
                for scales_int in scale_iterator:
                    scales = np.array(scales_int, dtype=float)
                    A_scaled = apply_col_scaling(A_perm, scales)
                    if is_strictly_diagonally_dominant(A_scaled):
                        print(f"  Found dominant form with scaling! Scales: {scales}")
                        return A_scaled, b_row, True, list(col_perm), scales

            if n == 3:
                iterator = product(scale_values, repeat=3)

    print("Could not make the matrix diagonally dominant with given search limits.")
    print("--- End dominance attempt ---")
    return A, b, False, list(range(n)), default_scales


def compute_iteration_matrix(A, b):
    diag = np.diag(A)
    inv_diag = 1.0 / diag
    L = np.tril(A, k=-1)
    R = np.triu(A, k=1)

    C = -(L + R) * inv_diag[:, np.newaxis]
    d = b * inv_diag

    return C, d
```

```
def get_spectral_radius(A):
    try:
        eigvals = np.linalg.eigvals(A)
        rho = float(np.max(np.abs(eigvals)))
    except np.linalg.LinAlgError:
        rho = np.inf
        print("Could not compute spectral radius.")

    return rho

def get_norms(A):
    norms = {
        "1-norm (col-sum)": float(np.linalg.norm(A, ord=1)),
        "2-norm (spectral)": float(np.linalg.norm(A, ord=2)),
        "inf-norm (row-sum)": float(np.linalg.norm(A, ord=np.inf))
    }

    return norms

def get_a_priori(C, d, eps):
    norm_C_inf = np.linalg.norm(C, ord=np.inf)
    a_priori_iters = None
    if norm_C_inf < 1.0:
        x0 = d.copy()
        x1 = C @ x0 + d
        delta = np.linalg.norm(x1 - x0, ord=np.inf)
        if delta > 0:
            numerator_arg = eps * (1 - norm_C_inf) / delta
            if numerator_arg > 0:
                try:
                    a_priori_iters = int(np.ceil(np.log(numerator_arg) / np.log(norm_C_inf)))
                    a_priori_iters = max(a_priori_iters, 1)
                except Exception:
                    a_priori_iters = None

    return a_priori_iters

def get_residual(A, x, b):
    try:
        residual_vec = A @ x - b
        residual_norm = float(np.linalg.norm(residual_vec))
        denom = (np.linalg.norm(A) * np.linalg.norm(x))
        relative_residual = float(residual_norm / denom) if denom != 0 else None
    except Exception:
```

```
    residual_norm = None
    relative_residual = None

return residual_norm, relative_residual
```