

# AI Critique

## Authorship Statement

I (Faycal Kilali) developed this code independently with collaborative input from ChatGPT (OpenAI, 2025) and Claude.AI (Anthropic, 2025). AI critiques are detailed in the comments of each respective function. In this document, I have clearly sectioned contributions from ChatGPT and Claude.AI to indicate which AI was used for specific aspects. This document explicitly presents the prompts and their corresponding outputs. I have also included the post-prompt difference relative to the current version images.

## ChatGPT

Prompt: how to update the constants outside the function through the inner funciton?

```
MATRIX_ROWS = 0
MATRIX_COLS = 0

def read_matrix(file):
    """
    Reads a matrix from a file, ignoring leading/trailing whitespace and empty lines. Each line is
    split into integers and stored as a list of lists.
    :param file: Path to the input file containing the matrix with space-separated values.
    :return: A 2D list representing the matrix.
    """
    with open(file, 'r') as f:
        matrix = [list(map(int, line.split())) for line in f if line.strip()]

    MATRIX_ROWS, MATRIX_COLS = len(matrix), len(matrix[0]) # Set the matrix rows and columns
    expected
    return matrix
```

AI Output:

In Python, constants (like MATRIX\_ROWS and MATRIX\_COLS) defined at the top level are typically not meant to be modified within functions. However, if you need to update these values from within a function, you can use the global keyword to indicate that you want to use the global variables instead of creating new local variables. Here's how you can modify your function:

```
MATRIX_ROWS = 0
MATRIX_COLS = 0

def read_matrix(file):
    """
    Reads a matrix from a file, ignoring leading/trailing whitespace and empty lines. Each line is
    split into integers and stored as a list of lists.
    :param file: Path to the input file containing the matrix with space-separated values.
    :return: A 2D list representing the matrix.
    """
    global MATRIX_ROWS, MATRIX_COLS # Indicate that we are using the global variables
```

```
with open(file, 'r') as f:
    matrix = [list(map(int, line.split())) for line in f if line.strip()]

MATRIX_ROWS, MATRIX_COLS = len(matrix), len(matrix[0]) # Update the global variables
return matrix
```

Explanation:

**Global Keyword:** By declaring global MATRIX\_ROWS, MATRIX\_COLS, you inform Python that you want to use the variables defined outside the function scope rather than creating new local variables with the same names.

**Updating Values:** You can then assign values to these global variables directly within the function.

Usage:

You can call read\_matrix with the path to your matrix file, and it will read the matrix, update the MATRIX\_ROWS and MATRIX\_COLS constants, and return the matrix as a list of lists.

Example:

Assuming you have a file named matrix.txt with the following content:

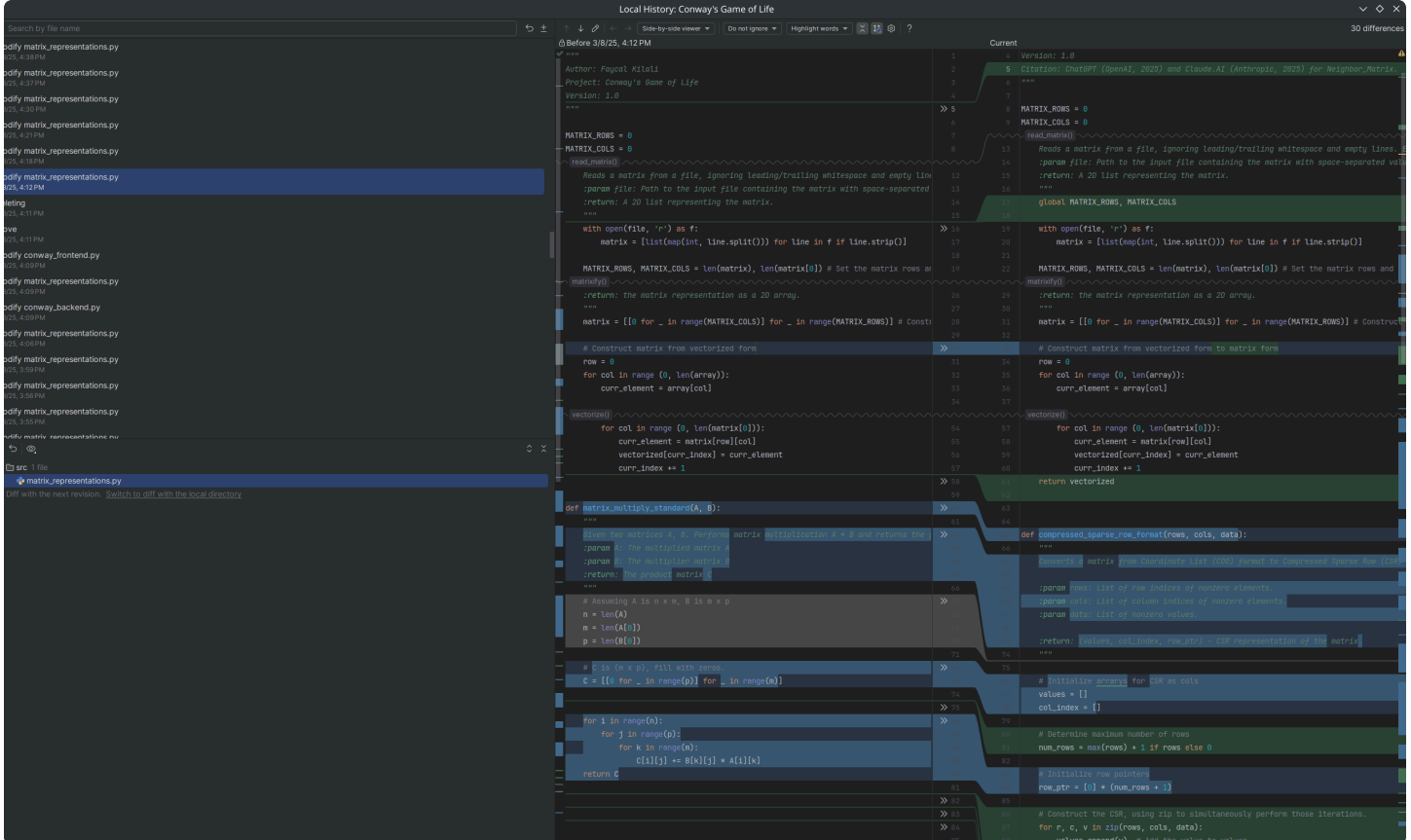
```
1 2 3
4 5 6
7 8 9
```

You would call the function as follows:

```
matrix = read_matrix('matrix.txt')
print("Matrix:", matrix)
print("Rows:", MATRIX_ROWS)
print("Columns:", MATRIX_COLS)
```

This will output:

```
Matrix: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Rows: 3
Columns: 3
```



Follow-up prompt: So, using CSR, am supposed to have somehow a matrix  $A * \text{vector } x$ , where  $x$  is the vectorized grid (not sure if this is the vectorized CSR or just the regular CSR) =  $y$ , where  $y$  is the number of neighbors for each pixel. In particular, this is for conway's game of life. I need you to explain what's going on here. What exactly is A?

Output:

In this context, think of A as a sparse matrix that encodes the "neighbor relationships" of each pixel (or cell) in the grid of Conway's Game of Life.

Breaking It Down:

Vectorized Grid (x):

The grid (a 2D array) is flattened into a 1D vector,  $x$ .  
Each element in  $x$  represents the state of a cell (e.g., alive or dead).

Matrix A (in CSR format):

A is constructed such that its non-zero entries indicate that two cells are neighbors.  
For each cell in the grid, A has a row that contains ones (or some weight) in the columns corresponding to its neighbors.  
The matrix is typically very sparse because, in a grid, each cell only has a small fixed number of neighbors compared to the total number of cells.  
Using the Compressed Sparse Row (CSR) format efficiently stores only the non-zero entries of A.

Matrix-Vector Multiplication ( $A * x = y$ ):

When you multiply A by  $x$ , each entry in the resulting vector  $y$  becomes the sum of the states of the neighboring cells for the corresponding grid cell.  
Essentially,  $y$  holds the count of alive neighbors for each cell, which is a key component in determining the next state of the grid in Conway's Game of Life.

What Exactly Is A?

A is a sparse representation of the grid's adjacency matrix.

The non-zero structure of A is determined by the rules of connectivity in the grid (for example, each cell might have up to 8 neighbors in a 2D grid).

In practical terms:

If cell  $i$  is a neighbor of cell  $j$ , then the element  $A[i, j]$  (or  $A[j, i]$ ) will be non-zero (often set to 1).

The CSR format helps store these indices and values efficiently without having to allocate memory for every possible pair, most of which are zero.

How It Works in Conway's Game of Life:

Initial State:

$x$  contains the current state of every cell (1 for alive, 0 for dead).

Neighbor Counting:

Multiplying A by  $x$  yields  $y$ , where each  $y[i]$  is essentially the sum of the alive states of the neighbors of the corresponding cell.

Rule Application:

With  $y$ , you can then apply the Game of Life rules (like a cell with exactly three live neighbors comes to life, etc.) to update the state of each cell for the next iteration.

Summary

A is not the grid itself, but rather a mathematical abstraction that encodes how cells are connected (neighbors).

Its representation in CSR format makes it computationally efficient when the grid is large, as it avoids unnecessary storage and operations on zero values.

$A * x = y$  is a clever way to compute the number of live neighbors for each cell in one operation, leveraging sparse matrix-vector multiplication.

This abstraction is particularly powerful because it transforms the problem into a linear algebra one, making it amenable to optimizations and parallel processing techniques often available in scientific computing libraries.

You said:

was  $x$  in CSR form? in COO form? just in regular matrix form before it was vectorized?

The vector  $x$  is typically just a regular, dense vector representing the grid's state before vectorization. Here's how it works:

$x$  is the flattened (vectorized) version of your 2D grid. It's a simple 1D array (or list) of values (e.g., 0s and 1s for dead and alive cells).

A is the sparse matrix (often stored in CSR format for efficiency) that encodes the neighbor relationships of the grid. It's not stored in COO form here, although you might initially construct it in COO format and then convert it to CSR for fast matrix-vector multiplication.

In summary,  $x$  remains a dense vector, while A is the sparse matrix (usually in CSR) that is used to compute the neighbors via multiplication.

Follow up prompt: Okay make it really simple so I can implement it. Make it very concise (step by step instructions on what to do)

Below is a very concise, step-by-step guide to implement this:

Prepare Your Grid:

Create your 2D grid (e.g., a list of lists) where each cell is 0 (dead) or 1 (alive).

Vectorize the Grid:

Flatten the 2D grid into a 1D dense vector  $x$  (e.g., using Python's list comprehension or NumPy's `flatten()`).

Build the Sparse Matrix  $A$ :

Determine neighbor connections: for each cell in the grid, find the indices of its neighbors.

Create a sparse matrix  $A$  where for each cell  $i$  and each neighbor  $j$ , you set  $A[i, j] = 1$ .

You can initially build  $A$  in COO (Coordinate) format then convert it to CSR (Compressed Sparse Row) using a library like `scipy.sparse`:

```
from scipy.sparse import coo_matrix

# Example: Collect row indices, col indices, and data
rows, cols, data = [], [], []
for i in range(n_rows):
    for j in range(n_cols):
        index = i * n_cols + j
        # For each neighbor (example for 8-neighborhood):
        for di in [-1, 0, 1]:
            for dj in [-1, 0, 1]:
                if di == 0 and dj == 0:
                    continue
                ni, nj = i + di, j + dj
                if 0 <= ni < n_rows and 0 <= nj < n_cols:
                    neighbor_index = ni * n_cols + nj
                    rows.append(index)
                    cols.append(neighbor_index)
                    data.append(1)
A_coo = coo_matrix((data, (rows, cols)), shape=(n_rows*n_cols, n_rows*n_cols))
A = A_coo.tocsr() # Convert to CSR format for fast multiplication
```

Compute the Neighbors Count:

Multiply the sparse matrix  $A$  by the dense vector  $x$ :

```
y = A.dot(x)
```

Here,  $y$  is a vector where each entry is the sum of live neighbors for the corresponding cell.

Apply the Game of Life Rules:

Use  $y$  (neighbor counts) along with your current grid (or  $x$ ) to update each cell's state:

A live cell with fewer than 2 or more than 3 live neighbors dies.

A dead cell with exactly 3 live neighbors becomes alive.

Reshape x back to a 2D grid if needed.

This step-by-step process allows you to use a sparse matrix (CSR) to efficiently compute neighbor counts in Conway's Game of Life.

Follow-up prompt: Can you somehow build the neighbor matrix using the matrix operations file?

Here's my `matrix_operations.py` file

```
"""
```

```
Author: Faycal Kilali
```

```
Project: Conway's Game of Life
```

```
Version: 1.0
```

```
"""
```

```
MATRIX_ROWS = 0
```

```
MATRIX_COLS = 0
```

```
def read_matrix(file):
```

```
    """
```

```
    Reads a matrix from a file, ignoring leading/trailing whitespace and empty lines. Each line is
    split into integers and stored as a list of lists.
```

```
    :param file: Path to the input file containing the matrix with space-separated values.
```

```
    :return: A 2D list representing the matrix.
```

```
    """
```

```
    global MATRIX_ROWS, MATRIX_COLS
```

```
    with open(file, 'r') as f:
```

```
        matrix = [list(map(int, line.split())) for line in f if line.strip()]
```

```
    MATRIX_ROWS, MATRIX_COLS = len(matrix), len(matrix[0]) # Set the matrix rows and columns
    expected
```

```
    return matrix
```

```
def matrixify(array):
```

```
    """
```

```
    Given a 1D array consisting of 64 integers, constructs the corresponding 8x8 matrix in row-major
    order.
```

```
    :param matrix: array of 64 integers
```

```
    :return: the matrix representation as a 2D array.
```

```
    """
```

```
    matrix = [[0 for _ in range(MATRIX_COLS)] for _ in range(MATRIX_ROWS)] # Construct 2D matrix
```

```
    # Construct matrix from vectorized form
```

```
    row = 0
```

```
    for col in range(0, len(array)):
```

```
        curr_element = array[col]
```

```
    # If column is divisible by MATRIX_COLS and not equal to 0, then that means we must move to
    the next row.
```

```
    if col % MATRIX_COLS == 0 and col != 0:
```

```
        row += 1
```

```
    # Fill matrix with element at appropriate position
```

```
    matrix[row][col % MATRIX_COLS] = curr_element
```

```
    return matrix
```

```
def vectorize(matrix):
```

```
    """
```

Given a matrix of the form of a 2D array (two dimensional array), vectorizes it in row-major order.

```
:param matrix: the matrix to be vectorized
:return: the vectorized matrix
"""
vectorized = [0] * (MATRIX_COLS * MATRIX_ROWS)

curr_index = 0
for row in range(0, len(matrix)):
    for col in range(0, len(matrix[0])):
        curr_element = matrix[row][col]
        vectorized[curr_index] = curr_element
        curr_index += 1
```

```
def matrix_multiply_standard(A, B):
    """
```

Given two matrices A, B. Performs matrix multiplication  $A * B$  and returns the product matrix C. Using iterative matrix multiplication.

```
:param A: The multiplied matrix A
:param B: The multiplier matrix B
:return: The product matrix C
"""
# Assuming A is n x m, B is m x p
n = len(A)
m = len(A[0])
p = len(B[0])

# C is (m x p), fill with zeros.
C = [[0 for _ in range(p)] for _ in range(m)]

for i in range(n):
    for j in range(p):
        for k in range(m):
            C[i][j] += B[k][j] * A[i][k]
return C
```

```
def coordinate_format(matrix):
    """
```

Constructs the COO (Coordinate) representation of a 2D matrix. In three separate arrays for each column.

```
:param matrix: input 2D matrix (list of lists)
:return: three lists: rows, columns, and data (nonzero values).
"""
rows = []
cols = []
data = []

for i in range(len(matrix)):
    for j in range(len(matrix[i])):
        value = matrix[i][j]
        if value != 0:
            rows.append(i)
            cols.append(j)
            data.append(value)

return rows, cols, data
```

```

def compressed_sparse_row_format(rows, cols, data):
    """
    Converts a matrix from Coordinate List (C00) format to Compressed Sparse Row (CSR) format.

    :param rows: List of row indices of nonzero elements.
    :param cols: List of column indices of nonzero elements.
    :param data: List of nonzero values.

    :return: (values, col_index, row_ptr) - CSR representation of the matrix.
    """

    # Initialize arrays for CSR as cols
    values = []
    col_index = []

    # Determine maximum number of rows
    num_rows = max(rows) + 1 if rows else 0

    # Initialize row pointers
    row_ptr = [0] * (num_rows + 1)

    # Construct the CSR, using zip to simultaneously perform those iterations.
    for r, c, v in zip(rows, cols, data):
        values.append(v) # Add the value to values
        col_index.append(c) # Add the column index
        row_ptr[r + 1] += 1 # Count nonzero entries for the row

    # Add up the counts to a sum for each row pointer that shares the same rows
    for i in range(1, len(row_ptr)):
        row_ptr[i] += row_ptr[i - 1]

    return values, col_index, row_ptr

```

Here's my `conway_backend.py` [file](#)

```

"""
Author: Faycal Kilali
Project: Conway's Game of Life
Version: 1.0
"""

```

```

import matrix_operations

```

```

def play_conways():
    """
    Takes
    :return:
    """

```



```
matrix = matrix_operations.read_matrix("init.txt")
```

```
def time_step(matrix):  
    """  
  
    :return:  
    """
```

AI Output:

ChatGPT said:

Certainly! We can leverage the existing functions in your `matrix_operations.py` to build the neighbor matrix in a simpler way without duplicating functionality. Specifically, we can use the existing `coordinate_format` function to directly create the C00 representation of the neighbor relationships. This will streamline the process significantly.

Step 1: Update `matrix_operations.py`

We will create a function that uses the existing `coordinate_format` to build the neighbor matrix.

Here's how to do it:

Updated `matrix_operations.py`

```
"""  
Author: Faycal Kilali  
Project: Conway's Game of Life  
Version: 1.0  
"""  
  
# Existing code...  
  
def build_neighbor_matrix(matrix):  
    """  
    Build a sparse matrix representing neighbor relationships in C00 format using existing  
    functions.  
    :param matrix: The current grid.  
    :return: C00 format of the neighbor matrix (rows, cols, data).  
    """  
    n_rows, n_cols = len(matrix), len(matrix[0])  
    neighbor_matrix = [[0 for _ in range(n_cols)] for _ in range(n_rows)]  
  
    # Create an empty C00 representation  
    rows = []  
    cols = []  
    data = []  
  
    for i in range(n_rows):  
        for j in range(n_cols):  
            current_index = i * n_cols + j  
            # Check 8 possible neighbors  
            for di in [-1, 0, 1]:  
                for dj in [-1, 0, 1]:  
                    if (di == 0 and dj == 0) or not (0 <= i + di < n_rows and 0 <= j + dj < n_cols):
```

```

        continue
        neighbor_index = (i + di) * n_cols + (j + dj)
        rows.append(current_index)
        cols.append(neighbor_index)
        data.append(1) # Each neighbor connection has a value of 1

```

```

    return rows, cols, data

```

# Existing code...

Step 2: Update conway\_backend.py

You will also need to ensure that you call the build\_neighbor\_matrix function correctly from your conway\_backend.py.

Updated conway\_backend.py

```

"""

```

```

Author: Faycal Kilali

```

```

Project: Conway's Game of Life

```

```

Version: 1.0

```

```

"""

```

```

import matrix_operations

```

```

def play_conways():

```

```

    """

```

```

    Play Conway's Game of Life.

```

```

    """

```

```

    matrix = matrix_operations.read_matrix("init.txt")

```

```

    # Convert matrix to vectorized form

```

```

    x = matrix_operations.vectorize(matrix)

```

```

    # Build sparse matrix A in COO format and convert to CSR format

```

```

    A_rows, A_cols, A_data = matrix_operations.build_neighbor_matrix(matrix)

```

```

    values, col_index, row_ptr = matrix_operations.compressed_sparse_row_format(A_rows, A_cols,

```

```

    A_data)

```

```

    # Run the game for a number of iterations

```

```

    for _ in range(10): # Example: 10 time steps

```

```

        x = update_grid(x, values, col_index, row_ptr)

```

```

        matrix = matrix_operations.matrixify(x) # Convert back to 2D matrix if needed

```

```

        print_grid(matrix)

```

```

def update_grid(x, values, col_index, row_ptr):

```

```

    """

```

```

    Update the grid based on the neighbor counts.

```

```

    :param x: The current state of the grid as a vector.

```

```

    :param values: CSR values.

```

```

    :param col_index: CSR column indices.

```

```

    :param row_ptr: CSR row pointer.

```

```

    :return: Updated grid as a vector.

```

```

    """

```

```

    # Calculate neighbor counts

```

```

    y = [0] * len(x)

```

```

for i in range(len(row_ptr) - 1):
    start, end = row_ptr[i], row_ptr[i + 1]
    for j in range(start, end):
        y[i] += x[col_index[j]]

# Apply the Game of Life rules
new_x = [0] * len(x)
for i in range(len(x)):
    alive_neighbors = y[i]
    if x[i] == 1: # Cell is alive
        new_x[i] = 1 if alive_neighbors in [2, 3] else 0
    else: # Cell is dead
        new_x[i] = 1 if alive_neighbors == 3 else 0

return new_x

def print_grid(matrix):
    """
    Print the matrix in a readable format.
    """
    for row in matrix:
        print(' '.join(str(cell) for cell in row))
    print()

# To run the game
if __name__ == "__main__":
    play_conways()

```

## Summary of Changes

In `matrix_operations.py`:

The `build_neighbor_matrix` function is implemented, which builds the neighbor matrix by iterating through each cell and checking its neighbors.

This function uses the existing logic in your file while creating the necessary COO format directly.

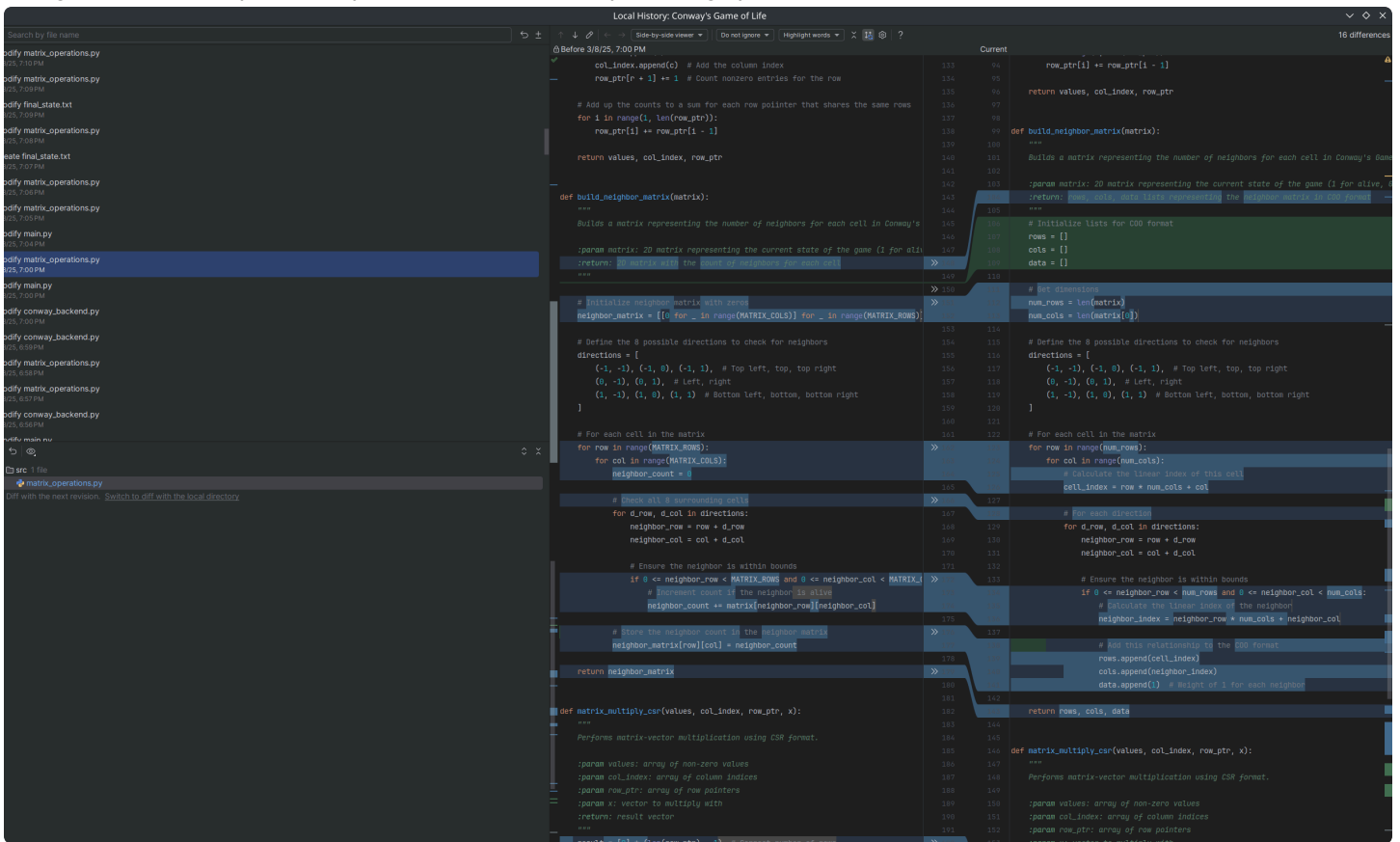
In `conway_backend.py`:

The call to `build_neighbor_matrix` remains the same, and it now directly uses the updated function in `matrix_operations.py`.

## Running the Code

With these updates, your implementation for Conway's Game of Life should now be cleaner and more efficient. You can run your `conway_backend.py` script as before, and it will use the neighbor matrix created using your existing functions.

The generated code (on the left) and the current code (on the right) differences



## Claude AI

For conway's game of life, finish the build neighbor matrix function

```
"""
```

Author: Faycal Kilali

Project: Conway's Game of Life

Version: 1.0

```
"""
```

```
MATRIX_ROWS = 0
```

```
MATRIX_COLS = 0
```

```
def read_matrix(file):
```

```
    """
```

Reads a matrix from a file, ignoring leading/trailing whitespace and empty lines. Each line is split into integers and stored as a list of lists.

:param file: Path to the input file containing the matrix with space-separated values.

:return: A 2D list representing the matrix.

```
    """
```

```
    global MATRIX_ROWS, MATRIX_COLS
```

```
    with open(file, 'r') as f:
```

```
        matrix = [list(map(int, line.split())) for line in f if line.strip()]
```

```
    MATRIX_ROWS, MATRIX_COLS = len(matrix), len(matrix[0]) # Set the matrix rows and columns expected
```

```
    return matrix
```

```

def matrixify(array):
    """
    Given a 1D array consisting of 64 integers, constructs the corresponding 8x8 matrix in row-major
    order.

    :param matrix: array of 64 integers
    :return: the matrix representation as a 2D array.
    """
    matrix = [[0 for _ in range(MATRIX_COLS)] for _ in range(MATRIX_ROWS)] # Construct 2D matrix

    # Construct matrix from vectorized form
    row = 0
    for col in range (0, len(array)):
        curr_element = array[col]

        # If column is divisible by MATRIX_COLS and not equal to 0, then that means we must move to
        the next row.
        if col % MATRIX_COLS == 0 and col != 0:
            row += 1

        # Fill matrix with element at appropriate position
        matrix[row][col % MATRIX_COLS] = curr_element

    return matrix

def vectorize(matrix):
    """
    Given a matrix of the form of a 2D array (two dimensional array), vectorizes it in row-major
    order.

    :param matrix: the matrix to be vectorized
    :return: the vectorized matrix
    """
    vectorized = [0] * (MATRIX_COLS * MATRIX_ROWS)

    curr_index = 0
    for row in range (0, len(matrix)):
        for col in range (0, len(matrix[0])):
            curr_element = matrix[row][col]
            vectorized[curr_index] = curr_element
            curr_index += 1

def matrix_multiply_standard(A, B):
    """
    Given two matrices A, B. Performs matrix multiplication A * B and returns the product matrix C.
    Using iterative matrix multiplication.

    :param A: The multiplied matrix A
    :param B: The multiplier matrix B
    :return: The product matrix C
    """
    # Assuming A is n x m, B is m x p
    n = len(A)
    m = len(A[0])
    p = len(B[0])

```

```
# C is (m x p), fill with zeros.
C = [[0 for _ in range(p)] for _ in range(m)]
```

```
for i in range(n):
    for j in range(p):
        for k in range(m):
            C[i][j] += B[k][j] * A[i][k]
return C
```

```
def coordinate_format(matrix):
    """
    Constructs the COO (Coordinate) representation of a 2D matrix. In three separate arrays for each
    column.
```

```
:param matrix: input 2D matrix (list of lists)
:return: three lists: rows, columns, and data (nonzero values).
"""
```

```
rows = []
cols = []
data = []
```

```
for i in range(len(matrix)):
    for j in range(len(matrix[i])):
        value = matrix[i][j]
        if value != 0:
            rows.append(i)
            cols.append(j)
            data.append(value)
```

```
return rows, cols, data
```

```
def compressed_sparse_row_format(rows, cols, data):
    """
```

```
Converts a matrix from Coordinate List (COO) format to Compressed Sparse Row (CSR) format.
```

```
:param rows: List of row indices of nonzero elements.
:param cols: List of column indices of nonzero elements.
:param data: List of nonzero values.
```

```
:return: (values, col_index, row_ptr) - CSR representation of the matrix.
"""
```

```
# Initialize arrays for CSR as cols
```

```
values = []
col_index = []
```

```
# Determine maximum number of rows
num_rows = max(rows) + 1 if rows else 0
```

```
# Initialize row pointers
```

```

row_ptr = [0] * (num_rows + 1)

# Construct the CSR, using zip to simultaneously perform those iterations.
for r, c, v in zip(rows, cols, data):
    values.append(v) # Add the value to values
    col_index.append(c) # Add the column index
    row_ptr[r + 1] += 1 # Count nonzero entries for the row

# Add up the counts to a sum for each row pointer that shares the same rows
for i in range(1, len(row_ptr)):
    row_ptr[i] += row_ptr[i - 1]

return values, col_index, row_ptr

def build_neighbor_matrix(matrix):

```

AI Output:

```

def build_neighbor_matrix(matrix):
    """
    Builds a matrix representing the number of neighbors for each cell in Conway's Game of Life.

    :param matrix: 2D matrix representing the current state of the game (1 for alive, 0 for dead)
    :return: 2D matrix with the count of neighbors for each cell
    """
    global MATRIX_ROWS, MATRIX_COLS

    # Initialize neighbor matrix with zeros
    neighbor_matrix = [[0 for _ in range(MATRIX_COLS)] for _ in range(MATRIX_ROWS)]

    # Define the 8 possible directions to check for neighbors
    directions = [
        (-1, -1), (-1, 0), (-1, 1), # Top left, top, top right
        (0, -1), (0, 1), # Left, right
        (1, -1), (1, 0), (1, 1) # Bottom left, bottom, bottom right
    ]

    # For each cell in the matrix
    for row in range(MATRIX_ROWS):
        for col in range(MATRIX_COLS):
            neighbor_count = 0

            # Check all 8 surrounding cells
            for d_row, d_col in directions:
                neighbor_row = row + d_row
                neighbor_col = col + d_col

                # Ensure the neighbor is within bounds
                if 0 <= neighbor_row < MATRIX_ROWS and 0 <= neighbor_col < MATRIX_COLS:
                    # Increment count if the neighbor is alive
                    neighbor_count += matrix[neighbor_row][neighbor_col]

```

```

        # Store the neighbor count in the neighbor matrix
        neighbor_matrix[row][col] = neighbor_count

    return neighbor_matrix

```

Follow-up prompt:

Okay, help me plot this properly in my conways\_frontend.py file. Requirements:

"Your final output should be a text file with similar format to the input file I give you, indicating the state of the simulation at the final generation (after the number of generations that I specify ... which is 25). Your code should also plot each generation in a single figure window, with some specified pause in seconds between each generation. The title of the figure should indicate the generation number. Generation 0 is the initial state, meaning you need to take 25 steps forward in time. That single figure should remain open at the completion of your code running."

```

"""
Author: Faycal Kilali
Project: Conway's Game of Life
Version: 1.0
"""

def display(Matrix):

```

AI Output:

I'll help you create a complete frontend file for Conway's Game of Life with visualization. Here's a solution that meets all your requirements:

This frontend implementation for Conway's Game of Life meets all your requirements:

#### Visualization Features:

- Displays each generation in a single figure window
- Uses a binary colormap (black for live cells, white for dead cells)
- Adds grid lines for better visibility
- Updates the title to show the current generation number
- Pauses between generations (configurable)

#### File I/O:

- Reads the initial state from an input file
- Saves the final state to an output file in the same format as the input

#### Simulation Control:

- Runs for exactly 25 generations by default (configurable)
- Generation 0 is the initial state
- Uses the CSR matrix multiplication for efficient updates



## User Flexibility:

Command-line arguments for input/output files, generation count, and pause time  
Default values provided for all parameters

To run this frontend, make sure:

The `matrix_operations.py` file contains all the functions we implemented earlier  
The `init.txt` file exists with your initial state

You can run it with default settings:

```
python conways_frontend.py
```

Or customize the parameters:

```
python conways_frontend.py --input my_init.txt --output my_final.txt --generations 100 --pause 0.2
```

The figure window will remain open after the simulation completes, and the final state will be saved to the specified output file.

```
"""
```

```
Author: Faycal Kilali
```

```
Project: Conway's Game of Life
```

```
Version: 1.0
```

```
"""
```

```
import matplotlib.pyplot as plt
```

```
import numpy as np
```

```
import time
```

```
def display(matrix, generation=0, pause_time=0.5):
```

```
    """
```

```
    Displays the current state of Conway's Game of Life using matplotlib.
```

```
    :param matrix: 2D matrix representing the current state
```

```
    :param generation: Current generation number
```

```
    :param pause_time: Time in seconds to pause before next update
```

```
    """
```

```
    # Clear previous plot
```

```
    plt.clf()
```

```
    # Create a figure with a grid
```

```
    plt.imshow(matrix, cmap='binary')
```

```
    plt.title(f"Conway's Game of Life - Generation {generation}")
```

```
    plt.grid(True, color='gray', linestyle='-', linewidth=0.5)
```

```
    # Remove tick marks
```

```
    plt.xticks([])
```

```
    plt.yticks([])
```

```
    # Draw the updated grid
```

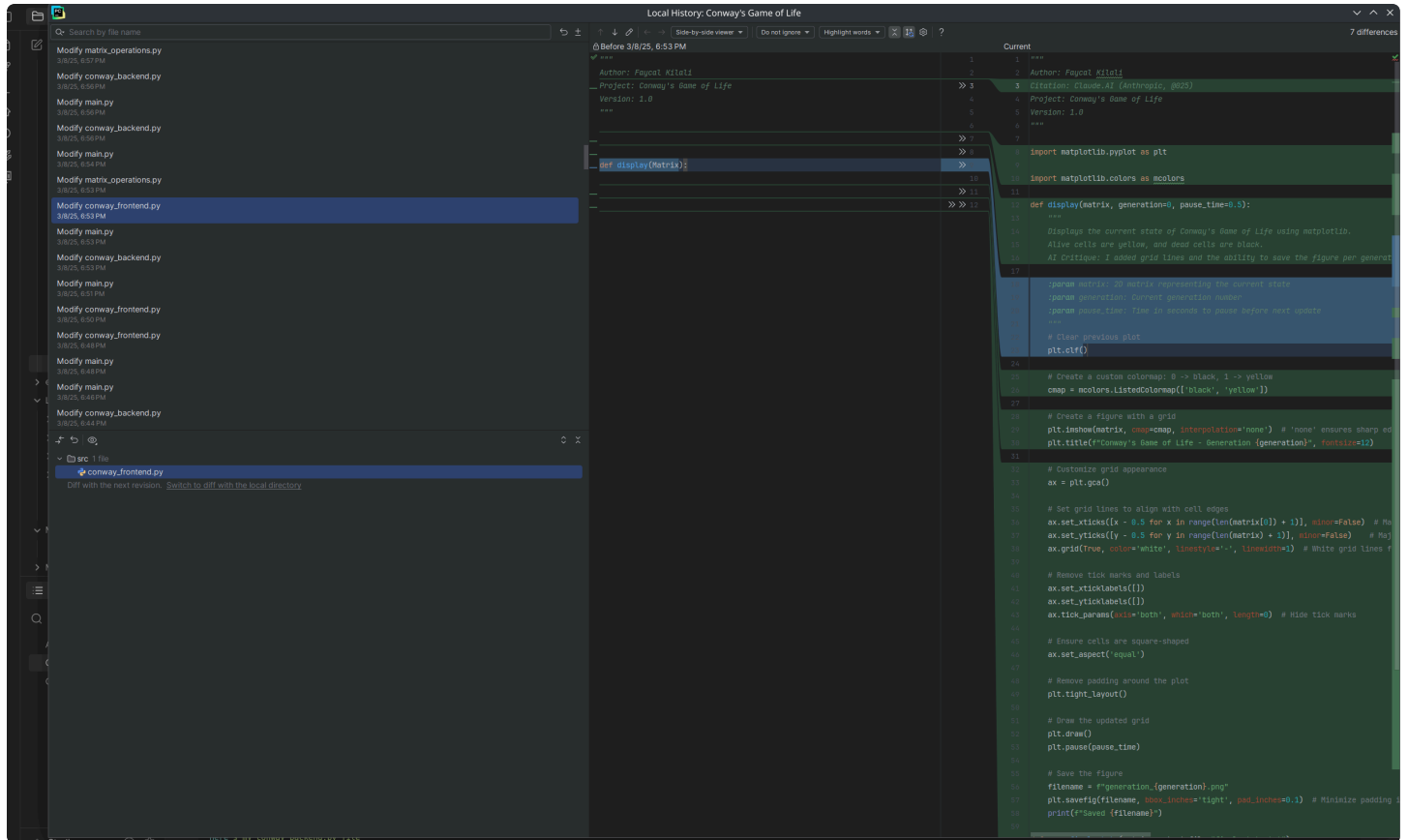
```
    plt.draw()
```

```
    plt.pause(pause_time)
```

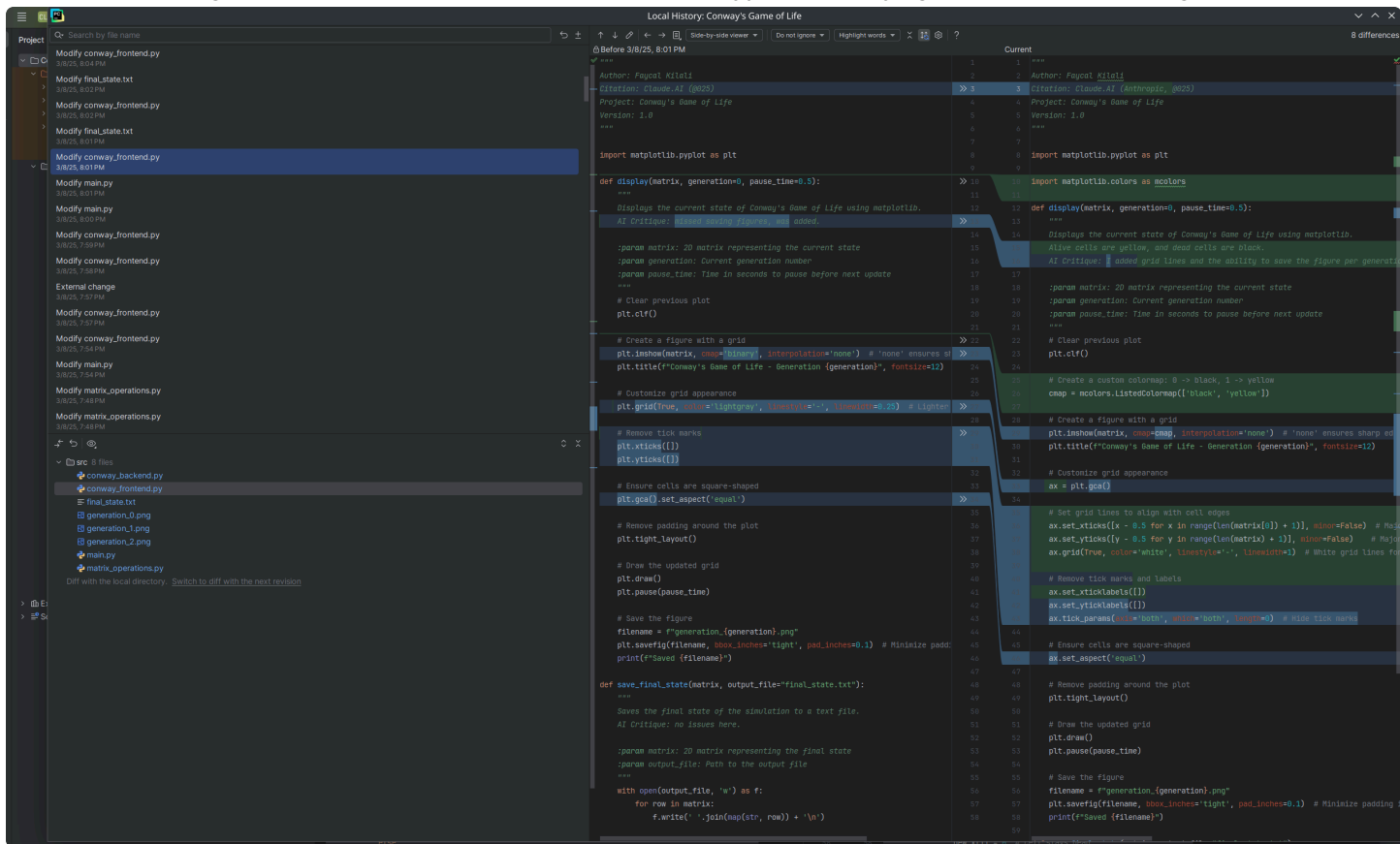
```
def save_final_state(matrix, output_file="final_state.txt"):
    """
    Saves the final state of the simulation to a text file.

    :param matrix: 2D matrix representing the final state
    :param output_file: Path to the output file
    """
    with open(output_file, 'w') as f:
        for row in matrix:
            f.write(' '.join(map(str, row)) + '\n')
```

## Before and after



Afterwards, I changed the code as follows to make it resemble a typical Conway's game of Life with its figures:



Prompt for README.md documentation:

Prompt: Generate a README.md file for this. Also explain how to use the program -- you have to invoke the function in main.py, called main(), and you must have a respective init.txt file in there.

AI Output:

## # Conway's Game of Life - Matrix-Vector Sparse Implementation

### ## Overview

This project implements Conway's Game of Life using matrix-vector operations and sparse matrices. The simulation leverages Python's efficient numerical operations along with sparse matrix representations (CSR format) to update the state of the grid over a series of generations.

### ## Features

- **Matrix Operations:** Reads an initial grid from a text file (`init.txt`), vectorizes it, and constructs a sparse neighbor matrix in CSR format.
- **Simulation:** Runs 25 generations of Conway's Game of Life, updating the state based on neighbor counts.
- **Visualization:** Uses matplotlib to display each generation with a brief pause between updates.
- **Output:** Saves the final state to `final\_state.txt`.

### ## Requirements

- Python 3.x
- Required Python packages:

```
- `matplotlib`  
- `numpy`
```

You can install the required packages using pip:

```
```bash
```

```
pip install matplotlib numpy
```

```
## File Structure
```

```
- **main.py:**
```

Contains the main entry point. The `main()` function invokes the simulation, initializes the grid from `init.txt`, updates the grid for each generation, and displays the results.

```
- **conway_backend.py:**
```

Handles the simulation logic by reading the initial state, building the sparse neighbor matrix, and updating the grid state for each generation.

```
- **conway_frontend.py:**
```

Contains functions for visualizing the grid (displaying the current generation) and saving the final state to a text file.

```
- **matrix_operations.py:**
```

Implements all matrix and vector operations, including functions for reading a matrix from file, vectorization, matrix multiplication (both standard and sparse/CSR-based), and building the neighbor matrix.

```
- **init.txt:**
```

Contains the initial grid configuration. This file should be present in the same folder as the Python scripts. An example for a 5x5 grid might look like:

```
0 1 1 0 0
```

```
0 1 1 0 0
```

```
0 0 0 0 1
```

```
1 0 0 1 1
```

```
1 1 0 0 0
```

```
## How to Use
```

1. **\*\*Ensure all required files are in one folder:\*\***

Make sure `main.py`, `conway_backend.py`, `conway_frontend.py`, `matrix_operations.py`, and `init.txt` are located in the same directory.

2. **\*\*Prepare your `init.txt` file:\*\***

Edit or create an `init.txt` file with the desired initial grid. Each row should contain space-separated values (0 for dead, 1 for alive).

3. **\*\*Run the Program:\*\***

Execute the main program by running `main.py`. This will automatically invoke the `main()` function:

```
python main.py
```

The simulation will run for 25 generations, displaying each generation in a matplotlib window. After the simulation, the final grid state is saved in `final_state.txt`.

```
## Citation & Authorship
```

```
- **Authors:**
    Faycal Kilali

- Contributions include all code modules and documentation.
- **AI Assistance:**
    Portions of the code and documentation were generated with the help of Claude.AI.
    Example APA citation for Claude.AI:

    > Anthropic. (2023). _Claude_ (Version 1.0) [Large language model]. Retrieved Month Day, Year,
    from https://www.anthropic.com/claude

- **Project Version:**
    1.0

## Additional Notes

- The code is designed to run in a single folder without requiring additional directories or
  external files.
- For collaborative development, please include details about individual contributions in this
  README or in separate documentation.
- For further details about the mathematical modeling and matrix operations used in this project,
  please refer to the accompanying documentation or inline code comments.

Happy simulating Conway's Game of Life!

This **README.md** file provides a clear overview of the project, usage instructions, file
descriptions, and appropriate citations, ensuring that any user or peer can easily run and
understand your program.
```

Given that there was no README.md content before the aforementioned prompt, there's no point of showing the difference in the changes. Afterwards, I improved upon the README.md output and ended up with the following changes;

```
1 # Conway's Game of Life - Matrix-Vector Sparse Implementation
2
3 ## Overview
4
5 This project implements Conway's Game of Life using matrix-vector operations and sparse matrices. The simulation
6 leverages python's efficient numerical operations along with sparse matrix representations (CSR format) to update
7 the state of the grid over a series of generations.
8
9 ## Features
10
11 - "Matrix Operations:" Reads an initial grid from a text file ('init.txt'), vectorizes it, and constructs a sparse
12   neighbor matrix in CSR format.
13 - "Simulation:" Runs 25 generations of Conway's Game of Life, updating the state based on neighbor counts.
14 - "Visualization:" Uses matplotlib to display each generation with a brief pause between updates.
15 - "Output:" Saves the final state to 'final_state.txt'.
```

```
14 ## Requirements
15
```

```
14 ## requirements
```

```
15
16 - Python 3.x
17 - Required Python packages:
18   - 'matplotlib'
19   - 'numpy'
```

```
21 You can install the required packages using pip:
```

```
22 ```bash
23 pip install matplotlib numpy
24 ## File Structure
```

```
26
27 "main.py:"
28 Contains the main entry point. The main() function invokes the simulation, initializes the grid from
29 init.txt, updates the grid for each generation, and displays the results.
30
31 "conway_backend.py:"
32 Handles the simulation logic by reading the initial state, building the sparse neighbor matrix, and updating the
33 grid state for each generation.
34
35 "conway_frontend.py:"
36 Contains functions for visualizing the grid (displaying the current generation) and saving the final state to a
37 text file.
38
39 "matrix_operations.py:"
40 Implements all matrix and vector operations, including functions for reading a matrix from file, vectorization,
41 matrix multiplication (both standard and sparse/CSR-based), and building the neighbor matrix.
```

```
42 "init.txt:"
43 Contains the initial grid configuration. This file should be present in the same folder as the Python scripts.
44 An example for a 5x5 grid might look like:
```

```
45 0 1 0 0
46 0 1 1 0 0
47 0 0 0 0 1
48 1 0 0 1 1
49 1 1 0 0 0
50 ## How to Use
```

```
51
52 1. "Ensure all required files are in one folder:"
53 Make sure main.py, conway_backend.py, conway_frontend.py, matrix_operations.py, and init.txt are
54 located in the same directory.
55
56 2. "Prepare your 'init.txt' file:"
57 Edit or create an init.txt file with the desired initial grid. Each row should contain space-separated values
58 (0 for dead, 1 for alive).
59
60 3. "Run the Program:"
61 Execute the main program by running main.py. This will automatically invoke the main() function:
```

```
1 # Conway's Game of Life - Matrix-Vector Sparse Implementation
2
3 ## Overview
4
5 This project implements Conway's Game of Life using matrix-vector operations and sparse matrices. The simulation
6 custom operations along with sparse matrix representations (CSR format) to update the state of the grid over a
7 series of generations.
8
9 ## Features
10
11 - "Matrix Operations:" Reads an initial grid from a text file ('init.txt'), vectorizes it, and constructs a sparse
12   neighbor matrix in CSR format.
13 - "Simulation:" Runs 25 generations of Conway's Game of Life, updating the state based on neighbor counts.
14 - "Visualization:" Uses matplotlib to display each generation with a brief pause between updates.
15 - "Output:" Saves the final state to 'final_state.txt'.
```

```
16
17 Conway's Game of Life: Code to Mathematical Concept Mapping
18
19 [Mathematical Concept|Code Implementation|Variable Mapping|Constraints & Notes]
20
21 [Grid representation (X)] 'read_matrix()' in matrix_operations.py | X = 2D matrix from init.txt | Each cell contains 0
22 (dead) or 1 (alive)
23
24 [Vectorized grid (x)] 'vectorize()' and 'matrixify()' in matrix_operations.py | x = flattened version of X in row-major
25 order | Length must be MATRIX_ROWS * MATRIX_COLS
26
27 [Neighbor counting matrix (A)] 'build_neighbor_matrix()' in matrix_operations.py | Returns (rows, cols, data) in COO
28 format | Each cell has up to 8 neighbors
29
30 [Sparse matrix storage] 'compressed_sparse_row_format()' in matrix_operations.py | (values, col_index, row_ptr) = CSR
31 representation | Efficient for sparse matrices
32
33 [Matrix-vector multiplication (y = Ax)] 'matrix_multiply_csr()' in matrix_operations.py | neighbor_counts =
34 multiplication result | Calculates live neighbors for each cell
35
36 [State transition rules] 'time_step()' in conway_backend.py | Implements all 4 GoL rules | Cell state depends on neighbor
37 count and current state
38
39 [Iteration counter (k)] 'generation' in main() | Tracks current generation number | Runs for a predefined number of
40 generations
41
42 [Initializing simulation] 'initialize_conways()' in conway_backend.py | Sets up X, x, and A matrix | Starting point for
43 the simulation
44
45 [Converting between representations] 'matrixify()' in matrix_operations.py | Converts vector back to matrix for
46 display | Used after each time step
47
48 [Algorithm step 1] 'build_neighbor_matrix()' in matrix_operations.py | Creates matrix A for counting neighbors | Follows
49 PDF's COO matrix construction approach
50
51 [Algorithm steps 2-3] 'initialize_conways()' in conway_backend.py | Initializes X^(0) and x^(0) | Reads initial state
52 from file
53
54 [Algorithm step 4] 'matrix_multiply_csr()' in matrix_operations.py | y^(k) = Ax^(k) | Calculates neighbor counts vector
55
56 [Algorithm step 5] 'time_step()' in conway_backend.py | x^(k) becomes x^(k+1) | Applies the four GoL rules
57
58 [Algorithm step 6] 'matrixify()' and 'display()' | Reshapes x^(k+1) to X^(k+1) and plots | Visual representation of
59 current state
60
61 [Algorithm step 7] 'loop in main()' | Increments generation counter | Continues simulation process
62
63 [Game of Life rules] 'time_step()' in conway_backend.py | If-else conditions based on cell state and neighbor count |
64 Implements all four rules from the PDF
65
66 Where the mappings corresponds to the variables provided in the GoL document provided by Dr. Lelievre.
```

```
38 ## Requirements
39
```

```
38 ## requirements
```

```
39
40 - Python 3.x
41 - Required Python packages:
42   - 'matplotlib'
```

```
44 You can install the required packages using pip:
```

```
45 ```bash
46 pip install matplotlib
47 ## File Structure
```

```
51
52 main.py:
53 Contains the main entry point. The main() function invokes the simulation, initializes the grid from init.txt,
54 updates the grid for each generation, and displays the results.
55
56 conway_backend.py:
57 Handles the simulation logic by reading the initial state, building the sparse neighbor matrix, and updating the
58 grid state for each generation.
59
60 conway_frontend.py:
61 Contains functions for visualizing the grid (displaying the current generation) and saving the final state to a
62 text file.
63
64 matrix_operations.py:
65 Implements all matrix and vector operations, including functions for reading a matrix from file, vectorization,
66 matrix multiplication (both standard and sparse/CSR-based), and building the neighbor matrix.
```

```
67 init.txt:
68 Contains the initial grid configuration. This file should be present in the same folder as the Python scripts.
69 An example for a 5x5 grid might look like:
```

```
70 0 1 0 0
71 0 1 1 0 0
72 0 0 0 0 1
73 1 0 0 1 1
74 1 1 0 0 0
```

```
73 How to Use
```

```
74
75 Ensure all required files are in one folder:
76 Make sure main.py, conway_backend.py, conway_frontend.py, matrix_operations.py, and init.txt are located in the
77 same directory.
```

```
56 python main.py
57
58
59 The simulation will run for 25 generations, displaying each generation in a matplotlib window. After the simulation,
the final grid state is saved in final_state.txt.
60
61 ## Citation & Authorship
62
63 **Authors:**
64 Faycal Kilali
65
66 Contributions include all code modules and documentation.
67 **AI Assistance:**
68 Portions of the code and documentation were generated with the help of Claude.AI.
69 Example APA citation for Claude.AI:
70
71 > Anthropic. (2023). _Claude_ (Version 1.0) [Large language model]. Retrieved Month Day, Year, from https://
www.anthropic.com/claude
72
73 **Project Version:**
74 1.0
75
76
77 ## Additional Notes
78
79 The code is designed to run in a single folder without requiring additional directories or external files.
80 For collaborative development, please include details about individual contributions in this README or in separate
documentation.
81 For further details about the mathematical modeling and matrix operations used in this project, please refer to
the accompanying documentation or inline code comments.
82
83 Happy simulating Conway's Game of Life!
84
85 This **README.md** file provides a clear overview of the project, usage instructions, file descriptions, and
appropriate citations, ensuring that any user or peer can easily run and understand your program.
86
```

```
77 Prepare your init.txt file:
78 Edit or create an init.txt file with the desired initial grid. Each row should contain space-separated values (0
for dead, 1 for alive).
79
80 Run the Program:
81 Execute the main program by running main.py. This will automatically invoke the main() function:
82
83 python main.py
84
85 The simulation will run for 25 generations, displaying each generation in a matplotlib window. After the
simulation, the final grid state is saved in final_state.txt.
86
87 Citation & Authorship
88
89 Authors:
90 Faycal Kilali
91
92 Contributions include all code modules and documentation.
93 AI Assistance:
94 Portions of the code and documentation were generated with the help of Claude.AI.
95
96 Project Version:
97 1.0
98
99 Additional Notes
100
101 The code is designed to run in a single folder without requiring additional directories or external files.
102 For further details about the mathematical modeling and matrix operations used in this project, please refer to
the accompanying documentation or inline code comments.
103
104
```

original text	Open file	Changed text	Open file
1 # Conway's Game of Life - Matrix-Vector Sparse Implementation		93 AI Assistance:	

Where the right-side is the improvements.