Question 1)

a)

```
(base) bhavithakakkirala@bhavithas-MacBook-Air hw1-Bhavitha88 % du -sh array_004000_asc.ou
320M    array_004000_asc.ou
(base) bhavithakakkirala@bhavithas-MacBook-Air hw1-Bhavitha88 % du -sh array_004000_bin.ou
122M    array_004000_bin.ou
```

Ascii file 320MB

Binary file 122MB

b) Estimation of Array Size in Memory

- **Matrix Dimensions**: n=4000 so the matrix has n*n = 4000*4000 = 16,000,000 elements
- **Size of one double**: 8 bytes
- **Size in memory**: Memory size = 8 × n*n = 8 × 16,000,000 = 128 MB.
- **Memory size** = 8 × n*n = 8 × 16,000,000 = 128MB.

Size in disk for ASCII format : 320 MB
Size in disk for Binary format : 128 MB

As we saw **Binary** format is taking less space than **ASCII** so it is better to store data in **Binary** format as it is space efficient.

Binary Format:

Smaller file size (33% smaller compared to CSV for n = 4000).

Faster read/write performance .

Scales better for very large datasets.

Code Explanation :

This program generates a n×n matrix, populates it with values, and writes it to disk in two formats: **ASCII (text)** and **binary**.

`main()` Function

Matrix Allocation:

Dynamically allocates memory for a 2D matrix `A` of size n×n.

Ensures memory is allocated row by row using `malloc`. If allocation fails, an error message is printed.

Matrix Initialization:

Populates the matrix `A` with values: A[i][j] = i + j

File Writing:

Calls the `print_to_file` function twice:

`format_flag = 0`: Writes the matrix in **ASCII (text)** format.

`format_flag = 1`: Writes the matrix in **binary** format.

Memory Deallocation:

Frees the memory allocated for the matrix to prevent memory leaks.

`print_to_file()` Function

Creates a filename using the matrix size (`n`) and format:

`array_<size>_asc.out` for ASCII format.

`array_<size>_bin.out` for binary format.

File Opening:

Opens the file in either **write mode (**`w`**)** for ASCII or **binary write mode (**`w`**)** for binary format.

Matrix Writing:

ASCII Format (`format_flag == 0`):Writes matrix elements row by row as floating-point numbers with 15 decimal precision.

Binary Format (`format_flag == 1`):Writes each row of the matrix directly using `fwrite` to store raw binary data.

Key Takeaways

Efficiency:

Binary format is faster and takes less storage space but is not human-readable.

ASCII format is easier to inspect and debug but less efficient for large data.

Scalability:

The program scales well for large matrices like n = 4000 , given sufficient memory.

Question 2)

These are the outputs of the given inputfiles

```
vec_000003_000001.in : Yes : -6.0000000000
vec_000003_000002.in : Yes : -6.0000000000
vec_000003_000003.in : Yes : -1.0000000000
vec_000003_000004.in : Not an eigenvector
vec_000005_000001.in : Yes : 0.2680980805
vec_000005_000002.in : Not an eigenvector
vec_000005_000003.in : Yes : 0.9868750245
vec_000005_000004.in : Yes : 1.3990385153
vec_000050_000001.in : Not an eigenvector
vec_000050_000002.in : Yes : 0.4796282347
vec_000050_000003.in : Yes : 1.3378872896
vec_000050_000004.in : Not an eigenvector
vec_000080_000001.in : Yes : 0.3330177549
vec_000080_000002.in : Yes : 0.4931419808
vec_000080_000003.in : Yes : 0.9392745158
vec_000080_000004.in : Not an eigenvector
```

Code Explanation : Finds whether the given vector is a eigen vector for the given matrix.If so it's corresponding eigen value is printed in the screen or else it prints that "Not an eigen vector"

`main`

- Reads the matrix size (`n`) from an input file.
- Dynamically allocates memory for the matrix and vector.
- Loops through all available vector files for the given matrix and processes each one using `is_eigenvector` function.

`read_matrix`

- Reads a matrix from a CSV file.
- Validates dimensions and ensures all data is correctly formatted.

`read_vector`

- Reads a vector from a file, handling whitespace and potential formatting errors.

Purpose

The `is_eigenvector` function determines whether a given vector v is an eigenvector of a matrix A and calculates the corresponding eigenvalue λ if it is.

Process

Compute Av:

1. Multiplies the matrix A with vector v to produce a new vector A*v.

Scaling for Numerical Stability:

1. Identifies the maximum absolute component of A*v to avoid overflow or underflow.
2. Scales both A*v and v by this maximum if it exceeds a large threshold.

Identify the Eigenvalue:

1. Compares corresponding elements of Av and v where $v[i] \neq 0$
2. Computes $\lambda = Av[i] / v[i]$ using the most numerically stable component.

Verification:

1. Checks if $Av \approx \lambda v$ within a small tolerance (`EPSILON`).
2. Considers numerical precision errors and ensures consistency across all components.

Return Result:

1. Returns `1` (true) if v is an eigenvector and stores the eigenvalue.
2. Returns `0` (false) if the conditions are not satisfied.

Strengths

- **Numerical Stability**: Avoids computational errors from large or small values by scaling.
- **Precision Handling**: Uses tolerances to account for floating-point inaccuracies.
- **Efficiency**: Processes the eigenvector equation component-wise, minimizing redundant calculations.