Linear Algebra Assignment 1

Anastasia Riana 邱慧莉 109006234

1.
```
--------Numpy Configuration------------
blas_mkl_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Users/riana.KOESNADI/anaconda3\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:/Users/riana.KOESNADI/anaconda3\\Library\\include']
blas_opt_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Users/riana.KOESNADI/anaconda3\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:/Users/riana.KOESNADI/anaconda3\\Library\\include']
lapack_mkl_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Users/riana.KOESNADI/anaconda3\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:/Users/riana.KOESNADI/anaconda3\\Library\\include']
lapack_opt_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Users/riana.KOESNADI/anaconda3\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:/Users/riana.KOESNADI/anaconda3\\Library\\include']
Supported SIMD extensions in this NumPy install:
    baseline = SSE,SSE2,SSE3
    found = SSSE3,SSE41,POPCNT,SSE42,AVX,F16C,FMA3,AVX2
    not found = AVX512F,AVX512CD,AVX512_SKX,AVX512_CLX,AVX512_CNL
```

```
--------Scipy Configuration------------
lapack_mkl_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Users/riana.KOESNADI/anaconda3\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.117\\windows\\mkl', 'C:\\Program Fil
es (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.117\\windows\\mkl\\include', 'C:\\Program Files (x86)\\IntelSWTools\\com
pilers_and_libraries_2019.0.117\\windows\\mkl\\lib', 'C:/Users/riana.KOESNADI/anaconda3\\Library\\include']
lapack_opt_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Users/riana.KOESNADI/anaconda3\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.117\\windows\\mkl', 'C:\\Program Fil
es (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.117\\windows\\mkl\\include', 'C:\\Program Files (x86)\\IntelSWTools\\com
pilers_and_libraries_2019.0.117\\windows\\mkl\\lib', 'C:/Users/riana.KOESNADI/anaconda3\\Library\\include']
blas_mkl_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Users/riana.KOESNADI/anaconda3\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.117\\windows\\mkl', 'C:\\Program Fil
es (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.117\\windows\\mkl\\include', 'C:\\Program Files (x86)\\IntelSWTools\\com
pilers_and_libraries_2019.0.117\\windows\\mkl\\lib', 'C:/Users/riana.KOESNADI/anaconda3\\Library\\include']
blas_opt_info:
    libraries = ['mkl_rt']
    library_dirs = ['C:/Users/riana.KOESNADI/anaconda3\\Library\\lib']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
    include_dirs = ['C:\\Program Files (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.117\\windows\\mkl', 'C:\\Program Fil
es (x86)\\IntelSWTools\\compilers_and_libraries_2019.0.117\\windows\\mkl\\include', 'C:\\Program Files (x86)\\IntelSWTools\\com
pilers_and_libraries_2019.0.117\\windows\\mkl\\lib', 'C:/Users/riana.KOESNADI/anaconda3\\Library\\include']
```

As we can see in both *numpy* and *scipy* configurations, BLAS/LAPACK information is taken from a library called **mkl_rt**. This is a default when using Anaconda as an IDE. In Anaconda channels, for example: Jupyter Notebook, mkl_rt is a separate package when users install numpy or scipy and include BLAS/LAPACK. This particular library is optimized for users with Intel CPUs. On other IDEs such as Visual Studio Code, numpy or scipy are built with OpenBLAS, hence when we run the same code in Visual Studio Code, information on libraries would not be available.

2.

a) In BLAS/LAPACK, the first letter of a routine describes what data type it would be solved in. If a routine consists of the letter S, it means that the data type will be real, single precision . Likewise for letter D, the data type will be real, double precision. On the other hand, if a routine consists of the letter C and Z, the data type will be complex, single precision and complex, double precision, respectively.

b)

SSPR2 is a routine which performs a symmetric rank two operation whose data type is real, single precision. The formula is:

$$A := \alpha * x * y^T + \alpha * x^T * y + A$$

where A is a matrix, x and y are both vectors, α is a scalar and x^T and y^T are transposes of vectors x and y respectively.

ZGERC is a routine which computes the rank one update of a general matrix, whose data type is a complex, double precision. The formula is:

$$A := A + \alpha * x * y^H$$

where A is a matrix, x is a vector, α is a scalar and y^H is the conjugate transpose of vector y.

DGBSVX is a routine that uses LU factorization, meaning the product of lower and upper triangular matrices, to compute real, double precision system of linear equations Ax=b with a band coefficient matrix A, columns of B as the individual right-hand sides and columns of x as the solutions.

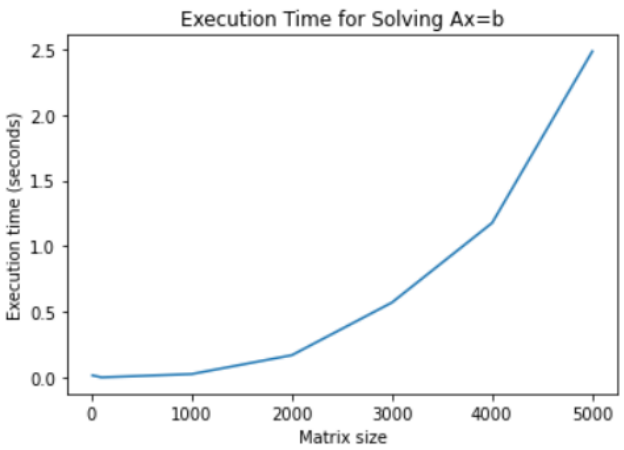
CHEEVR is a routine that computes selected eigenvalues and optionally eigenvectors of complex Hermitian matrix A, stored in either lower or upper storage mode. Both eigenvalues and eigenvectors can be selected by specifying either the range of values or a range of indices for the desired eigenvalues. The formula is:

$$Az = wz \qquad \text{where } A=A^H \text{ or } A=A^T$$

where A is a matrix, A^T is the transpose of matrix A, A^H is the conjugate transpose of matrix A. Eigenvalues and eigenvectors are returned in vector w and matrix Z respectively.

3. The best subroutine to solve the Ax=b linear system is using **?GESV**. As we want to solve this Ax=b problem in double precision, we can use either DGESV or DGESVX which are available only in LAPACK, that is inside the **scipy** library. I chose to use

DGESV as it is a simple driver that solves the system Ax=b by factoring A and overwriting b which results in x as its solution. Both DGESV and DGESVX can compute multiple columns of b. The result of the code is as below:



We can verify the correctness of the code by using a function in the **numpy** library, in particular the *allclose* function. *Allclose* function is used to find if two arrays are element-wise equal to each other within some tolerance. The tolerance is going to be positive and typically very small. The way we use this function is to compare the result of A dot b with x and the function will return either a True or False statement. If the function returns a False statement, we can display a text which will tell us that there are some errors that occured in certain n. If the statement returns a True statement, we append the runtime to an array so it is easier to be made into a plot.

4. Block matrix is a matrix that's defined using smaller matrices called *blocks*. When two block matrices have the same shape, we can just multiply them similarly like the matrix multiplication. To solve the Ax=b system, first A can be rewritten to A_1 and A_2, where A_1 and A_2 are both 2x2 matrices. Then, we multiply A_1 and A_2 to a 2x1 vector of x1 and x2, like the figure below.

$$[[A]_1 \quad [A]_2] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = b.$$

Matrix-matrix multiplication can solve this, since both A_1 and A_2 can be multiplied together with the vector of x1 and x2. A good subroutine to accelerate the performance of computing Ax=b, such as Strassen's algorithm will reduce the time complexity from $O(n^3)$, using the intuitive way to $O(n^{2.807\ldots})$, which is significantly faster.

5. Strassen's algorithm is a method for matrix multiplication where the matrix is divided into four smaller matrices of n/2 x n/2 in each step recursively. For Strassen's algorithm to work, both matrices must be a square matrix and n must be a power of

two. If those restrictions are not satisfied, we must fill it with zeros to meet the conditions.

Algorithm 1: The Naive Matrix Multiplication Algorithm

```
Data: S[A][B], P[G][H]
Result: Q[][]
if B == G then
    for m = 0; m < A; m++ do
        for r = 0; r < H; r++ do
            Q[m][r] = 0;
            for k = 0; k < G; k++ do
                Q[m][r] += S[m][k] * P[k][r];
            end
        end
    end
end
```

To know why Strassen's algorithm is faster, we need to understand the intuitive way of approaching the matrix-matrix multiplication problem. It can be seen from the image, there are three nested for-loops that loop for N times. Then its growth of functions will be $T(n) = 8T(n/2) + cn^2$. By Master's theorem, the total time complexity is $O(n^3)$. As we know, $O(n^3)$ time is not good and it can be improved.

Strassen's algorithm uses the divide-and-conquer algorithm to solve this problem. In the intuitive way, we are using eight multiplications, while Strassen's only uses seven.

Algorithm 3 Strassen's Algorithm

function STRASSEN(M,N)
  if M is 1×1 **then**
    return $M_{11}N_{11}$
  end if
  Let $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ and $N = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$
  Set $S_1 = \text{STRASSEN}(B - D, G + H)$
  Set $S_2 = \text{STRASSEN}(A + D, E + H)$
  Set $S_3 = \text{STRASSEN}(A - C, E + F)$
  Set $S_4 = \text{STRASSEN}(A + B, H)$
  Set $S_5 = \text{STRASSEN}(A, F - H)$
  Set $S_6 = \text{STRASSEN}(D, G - E)$
  Set $S_7 = \text{STRASSEN}(C + D, E)$
  return $\begin{pmatrix} S_1 + S_2 - S_4 + S_6 & S_4 - S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{pmatrix}$
end function

In this implementation, there will be seven recursive calls and both addition and subtractions will take cn^2 time for some constant c. Hence, its growth of function will be $T(n) = 7T(n/2) + cn^2$ and by Master's theorem, $T(n) = O(n \text{ ^ } \log_2 7) = O(n^{2.807\cdots})$ which is in theory, faster than $O(n^3)$.

Strassen's algorithm is not used in high performance computing due to its recursive nature. Recursion uses something called *"call stack"* meaning that when the program runs, functions would be called continuously. When a function is called, that function is going to the top of the call stack. Hence, the computer needs to make extra space to store the call stack which consumes more memory. Other than that, recursive calls add latency to the program. This algorithm is also not suitable for sparse matrices. Hence, for small n, the intuitive approach might be better suited to solve the problem. As for big n, Strassen's algorithm will be better.