Final Project: Matrix Multiplication

Riko Jacob and Matti Karppa

Hand-in date: 2023-01-06 13:59

The assignment is to be solved in groups of 1–3 people.

1 Introduction

This is the final project of the Applied Algorithms course. The project serves as an exam and it is to be solved in groups of 1–3 people. After the hand-in date, the written exam will be followed by an oral exam on January 23–25, 2023. The exact dates and times will be assigned later to individual groups. The oral exam consists of a group part where the group presents their work to the examiner, followed by individual examination of each member of the group. Detailed instructions on the oral examination will be given later.

Problem setting. This project addresses *matrix multiplication.* Assuming the notation presented during lectures, we define (square) matrix product as follows, together with the associated computational problem.

Definition 1 (Matrix Product)**.** Given two operand matrices A and B of shape $n \times n$, we define the *matrix product* $C = AB$ to be the $n \times n$ matrix C that satisfies

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \,, \tag{1}$$

for all $i, j \in \{1, 2, \dots, n\}$.

Definition 1 gives rise to the computational problem of matrix multiplication: Given two $n \times n$ matrices A and B as input, compute their product $C = AB$.

We will assume throughout the project that all matrices are *square* with the side length n a *power of two.* We follow the convention that upper case letters, such as A, B, and C, refer to matrices, and lower case letters, such as a_{ij}, are scalar elements of the matrix. The scalar a_{ij} would be the element on the ith row and the jth column. When we use subindices with capital letters, we denote *block matrices*, for example, A_{11} would be the upper-left corner submatrix of A; the precise size of A_{11} depends on context. For example, we could say that the $n \times n$ matrix A consists of four submatrices of size $\frac{n}{2} \times \frac{n}{2}$:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} ,$$

so A can be obtained by concatenating the submatrices appropriately. For example, if

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} ,$$

then

$$A_{11} = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} , \qquad A_{12} = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix} ,$$
$$A_{21} = \begin{bmatrix} 9 & 10 \\ 13 & 14 \end{bmatrix} , \qquad A_{22} = \begin{bmatrix} 11 & 12 \\ 15 & 16 \end{bmatrix} .$$

Your task will be to implement and evaluate several different matrix multiplication algorithms using floating-point arithmetic. It is recommended that you use double-precision IEEE 754 floating-point numbers. These correspond to the data type `double` in Java and C.

Tools. While you are free to use any programming language you wish, we have provied you with a stub of such a linear algebra library in Java that satisfies the needs of the project. The file `Matrix.java` defines a class called `Matrix` with the basic functionality that is required to operate with dense row-major double-precision floating-point matrices. However, the logic of the library is partially missing, and you need to fill it in. The library file also includes function declarations of the necessary procedures you need to implement during this project, so filling the stub file is sufficient for the implementation part of this project, although you will need to verify the correctness of your implementations separately.

The library stub serves to provide you with an Application Programming Interface (API) that is sufficient for the needs of this course and assist you in implementing the necessary functionality. It is, however, completely optional, and all tasks can be performed with any other tools that you may wish, and you may change or extend the library as you see fit.

If you choose to use external libraries directly, please only use elementary operations (such as matrix addition, scalar multiplication, submatrix access) and maintain full control over the inner workings of the routines you are explicitly asked to implement; that is, it is not sufficient to implement matrix multiplication or transpose as a simple function call to an external library, you need to implement the internal logic yourself.

Note. Your task is to apply what you have learned in class to practice, not to craft a perfect, fully-fledged, competitive implementation of these algorithms. Be ready to identify things that did not go as expected, identify problems, address them as you can, but most importantly, be prepared to document it if something goes wrong.

Caveat. Floating-point numbers may behave in surprising ways. One such problem relates to performance if we operate on numbers that are close to the *machine epsilon* that is, if we encounter numbers that are small enough to be close to what can be represented. In such cases, *subnormal numbers* may arise, and as they need specific treatment, this may have a surprisingly drastic impact on performance.

Another problem relates to the correctness of the results. Since floating-point numbers have a finite precision, even when correctly *rounded*, the results may depend on the order of operations, and may not match the expectations derived by applying mathematics on real numbers. Therefore, when testing the algorithm for correctness, the input data must be carefully chosen such that comparisons to a ground truth are meaningful.

We recommend that you test the correctness of your algorithm with sufficiently small integer-valued elements that can be represented exactly, including all intermediate products. For example, we have 24 bits of mantissa in 32-bit floating points, so for $n = 256$, having all elements as positive integer values less than x yields the equation

$$nx^2 < 2^{24}\,,$$

which implies $x < 256$, so having such small values will prevent any rounding errors. A similar approach can be followed for performance experiments as well to avoid running into complications arising from rounding errors; please document carefully how you generated your input.

Tasks. Tasks that need to be completed are presented in a specific environment marked by the word **Task** in bold. In cases where the task number is followed by a number in parentheses, this indicates that the task is independent of tasks with a higher number and needs only be completed if you want a grade equal to or higher than the number in parentheses. In cases where there is no grade in parentheses, the task is required for passing. Completing the tasks is not a guarantee for the grade in parentheses, however, as the final grade depends on your achievement of the intended learning outcomes as a whole. For implementation tasks, you are required to test your implementation for correctness.

2 The Elementary Algorithm

We call the naïve algorithm implementing the definition from Equation (1) the *elementary algorithm.* That is, the elementary algorithm is the immediate algorithm that one obtains by three nested loops. A correct implementation of the elementary algorithm will be useful for checking the correctness of more complex algorithms.

Task 1. Implement the elementary matrix multiplication with three nested loops.

Task 1 amounts to implementing the Algorithm 1 in the functions called `elementaryMultiplication` in the Java library stub. The three-argument variant should implement the operation $C \leftarrow C + AB$, such that it can be used to accumulate results in-place. It may make sense to implement that function first, and then only write `elementaryMultiplication` as a thin wrapper around it.

Algorithm 1 Pseudocode for the elementary matrix multiplication.

```
1: Input: $n \times n$ matrix $A$, $n \times n$ matrix $B$
2: Output: $n \times n$ matrix $C = AB$
3: procedure MatMul($A, B, C$)
4:     Set $c_{ij} \leftarrow 0$ for all $i \leftarrow 1, 2, \ldots, n$ and $j \leftarrow 1, 2, \ldots, n$
5:     for $i \leftarrow 1, 2, \ldots, n$ do
6:         for $j \leftarrow 1, 2, \ldots, n$ do
7:             for $k \leftarrow 1, 2, \ldots, n$ do
8:                 $c_{ij} \leftarrow c_{ij} + a_{ik}b_{kj}$
9:             end for
10:        end for
11:    end for
12: end procedure
```

3 Matrix Transpose

We discussed in class the possibility of making the elementary algorithm more efficient by transposing one of the operands. Before we can make use of that, we need to be able to transpose our matrices. Recall that the transpose $A^\top$ of the matrix A is the result of an operation that switches the rows and columns with one another, that is, the elements of the transposed matrix satisfy

$$a_{ij}^\top = a_{ji}\,.$$

Task 2. Implement the simple transpose of a matrix that creates a copy using two nested loops.

Task 2 amounts to implementing the function `transpose` in the library stubs, as per Algorithm 2. The two-argument static version stores the transposed version of the source matrix in the target matrix, and as with Task 1, it may be useful to implement this function first and the non-static version as a thin wrapper.

Algorithm 2 Pseudocode for the elementary transpose.

```
1:  Input: $n \times n$ matrix $A$.
2:  Output: $n \times n$ matrix $A^\top$
3:  function TRANSPOSE($A$)
4:      Initialize $A^\top$
5:      for $i \leftarrow 1, 2, \ldots, n$ do
6:          for $j \leftarrow 1, 2, \ldots, n$ do
7:              $a_{ji}^\top \leftarrow a_{ij}$
8:          end for
9:      end for
10:     return $A^\top$
11: end function
```

We also talked in class how to use recursion to make the transpose more cache-efficient by applying recursion on the $\frac{n}{2} \times \frac{n}{2}$ block matrices.

Task 3 (Grade 4)**.** Implement the recursive transpose algorithm with the parameter controlling the minimum submatrix size s specified as a parameter to the subroutine you implement.

Task 3 amounts to implementing Algorithm 3 in the `transposeRec` function in the library stub. You will need to use the elementary transpose routine as a subroutine for cases below the threshold size s which is the lower-bound on the problem size that is solved recursively. This is because the recursive calls present overhead. To figure out a proper size for the parameter s, you should evaluate your algorithm experimentally.

Algorithm 3 Pseudocode for the elementary transpose.

```
1: Input: $n \times n$ matrix $A$, minimum block size $s$.
2: Output: $n \times n$ matrix $B = A^\top$ ($B$ is assumed to have been initialized
   in the correct shape before calling this procedure)
3: procedure TransposeRec($A, B$)
4:     if $n \leq s$ then
5:         $B \leftarrow$ Transpose($A$)
6:     else
7:         Let $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$
8:         Let $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$
9:         TransposeRec($A_{11}, B_{11}$)
10:        TransposeRec($A_{12}, B_{21}$)
11:        TransposeRec($A_{21}, B_{12}$)
12:        TransposeRec($A_{22}, B_{22}$)
13:    end if
14: end procedure
```

Task 4 (Grade 4)**.** Fix a suitable size for the matrix n, and evaluate your implementation of the recursive transpose with different values of s. Use this information to determine the optimal value for s.

To obtain meaningful results in Task 4, you will need to use sufficiently large input matrices, such that they cannot fit in your L3 cache. You should have $n \geq 2048$, and preferably even larger.

4 Transposed Matrix Multiplication

Assuming the matrices are stored in row-major order, it makes sense to transpose the right-hand operand matrix, such that when evaluating the innermost loop of the elementary algorithm, both matrices are accessed sequentially. For column-major order, conversely, the left-hand operand should be transposed.

Task 5. Implement a variant of the Elementary Matrix Multiplication (see Task 1) such that first creates a transposed copy of its right-hand operand, and then computes the matrix product using three nested loops that access both operands sequentially in the chosen memory layout.

Task 5 amounts to implementing the function `elementaryMultiplication-Transposed` in the Java library stub. The algorithm is otherwise the same as in Algorithm 1, except you need to swap the order of the right-hand indices, and you need to apply the transpose beforehand.

5 Tiled Matrix Multiplication

We discussed the ideal-cache model and the tiling algorithm for matrix multiplication in class. The idea is to break the operands and the output matrix into tiles of size $s \times s$, and perform $(\frac{n}{s})^3$ submatrix multiplications of size $s \times s$. You may assume s divides n. There are a few ways this algorithm can be implemented; the most obvious ones are one where the elementary algorithm is used as a subroutine that is called for each tile triple, or one can expand the subroutine call and construct the routine directly as six nested loops.

Task 6 (Grade 7)**.** Implement the tiled Matrix Multiplication, with the parameter controlling the submatrix size s to be specified as a parameter to the subroutine you implement.

Task 6 amounts to implementing the function `tiledMultiplication` in the Java library stub, following Algorithm 4. The task also requires it to be possible to experiment with different values of s. This is precisely what you will be doing in the next task.

Task 7 (Grade 7)**.** Fix n to a suitable, sufficiently large power of two. Justify your choice of n. Try different powers of two for the parameter s of the tiling matrix multiplication algorithm from Task 6 from 2 up to n, and determine which choice of the parameter is the most beneficial in terms of runtime.

In order to be able to see any interesting effects in Task 7, you will need to have sufficiently large matrices as operands such that they do not fit in the L3 cache of your computer. You should use at least $n \geq 2048$, the larger the better.

Algorithm 4 Pseudocode for the tiled matrix multiplication.

```
Input: $n \times n$ matrix $A$, $n \times n$ matrix $B$, tile size $s$
Output: $n \times n$ matrix $C = AB$
function MatMulTiled$(A, B)$
    Initialize $n \times n$ matrix $C$ as all zeros.
    for $i_0 = 1, 2, \ldots, \frac{n}{s}$ do
        for $j_0 = 1, 2, \ldots, \frac{n}{s}$ do
            for $k_0 = 1, 2, \ldots, \frac{n}{s}$ do
                $C_{i_0 j_0} \leftarrow C_{i_0 j_0} + A_{i_0 k_0} B_{k_0 j_0}$    ▷ Operands are of shape $s \times s$.
            end for
        end for
    end for
    return $C$
end function
```

6 Recursive Matrix Multiplication

We also discussed the recursive matrix multiplication. Your task is to implement it. The idea is simple: let us represent the $n \times n$ input matrices A and B and the output matrix C as four $\frac{n}{2} \times \frac{n}{2}$ block matrices each:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}, \qquad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}.$$

Since we have

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}\,, \quad & C_{12} &= A_{11}B_{21} + A_{12}B_{22}\,, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}\,, \quad & C_{22} &= A_{21}B_{21} + A_{22}B_{22}\,, \end{aligned} \tag{2}$$

we can perform the $n \times n$ matrix multiplication as a recursive procedure of computing eight $\frac{n}{2} \times \frac{n}{2}$ submatrix multiplications using the same routine, and then combining the results according to Equation (2), with scalar multiplication in the base case of $n = 1$. This can be formulated as a straight-line program by defining

$$\begin{aligned} &P_1 = A_{11}\,, \quad Q_1 = B_{11}\,, \quad P_2 = A_{12}\,, \quad Q_2 = B_{21}\,, \\ &P_3 = A_{11}\,, \quad Q_3 = B_{12}\,, \quad P_4 = A_{12}\,, \quad Q_4 = B_{22}\,, \\ &P_5 = A_{21}\,, \quad Q_5 = B_{11}\,, \quad P_6 = A_{22}\,, \quad Q_6 = B_{21}\,, \\ &P_7 = A_{21}\,, \quad Q_7 = B_{12}\,, \quad P_8 = A_{22}\,, \quad Q_8 = B_{22}\,, \end{aligned} \tag{3}$$

followed by eight recursion calls computing the $\frac{n}{2} \times \frac{n}{2}$ product $M_i = P_i Q_i$ for $i = 1, 2, \ldots, 8$, followed by computing

$$\begin{aligned} C_{11} &= M_1 + M_2\,, & C_{12} &= M_3 + M_4\,, \\ C_{21} &= M_5 + M_6\,, & C_{22} &= M_7 + M_8\,. \end{aligned} \tag{4}$$

A direct implementation of the algorithm from Equations (3) and (4) would imply creating intermediate copies at each stage. Although the intermediate copies are $\Theta(n^2)$ operations and do not contribute to the asymptotic analysis as such, common sense tells us that it might be slowing us down unnecessarily since the Equations (3) and (4) show that the intermediate results are trivial. The variant introduced in lecture does not perform any operations on intermediate results; all additions are deferred to the base case. We will call this variant the *write-through* version of the recursive algorithm.

Task 8 (Grade 10)**.** Implement the write-through variant of the recursive Matrix Multiplication according to the pseudocode presented on lecture slides. Include a parameter that determines the subproblem size such that the routine falls back to elementary algorithm. Allow the possibility that the elementary algorithm is not used at all.

Task 8 amounts to implementing the `recursiveMultiplication` in the Java library stub, following Algorithm 5. Note that the algorithm takes a minimum base case s as an argument; this determines the minimum problem size at which the algorithm falls back to the elementary algorithm. Note that the elementary algorithm is assumed to work *additively*, that is, the result is added to the output matrix. Be sure to use a *shallow* view of the submatrices when performing the operations, that is, you should not copy data in any intermediate steps, but only create an appropriate matrix header.

It will be interesting to analyze the behavior of the recursive algorithm with respect to its parametrization, much like the previous case of the tiled matrix multiplication.

Task 9 (Grade 10)**.** Fix n to a suitable value, and evaluate the best subproblem size between 0 and $\frac{n}{2}$ that the recursive algorithm of Task 8 should fall back to elementary algorithm. Justify your choice of n. Discuss your findings. How did the algorithm behave with respect to the parameter, and why do you believe that is?

As before, you need to fix n to be sufficiently large in order to see any interesting behavior. The choice of a sufficient n depends on your L3 cache size.

Algorithm 5 Pseudocode for the recursive matrix multiplication.

```
Input: $n \times n$ matrix $A$, $n \times n$ matrix $B$, minimum problem size $s$
Output: $n \times n$ matrix $C = AB$
procedure MatMulRecursive($A, B, C, s$)
    if $n = 1$ then
        $c_{11} \leftarrow c_{11} + a_{11}b_{11}$                ▷ Scalar multiplication
    else if $n \leq s$ then
        Matmul($A, B, C$)                                         ▷ Non-trivial base case
    else
        MatMulRecursive($A_{11}, B_{11}, C_{11}$)
        MatMulRecursive($A_{12}, B_{21}, C_{11}$)
        MatMulRecursive($A_{11}, B_{12}, C_{12}$)
        MatMulRecursive($A_{12}, B_{22}, C_{12}$)
        MatMulRecursive($A_{21}, B_{11}, C_{21}$)
        MatMulRecursive($A_{22}, B_{21}, C_{21}$)
        MatMulRecursive($A_{21}, B_{12}, C_{22}$)
        MatMulRecursive($A_{22}, B_{22}, C_{22}$)
    end if
end procedure
```

7 Strassen's algorithm

We also very quickly introduced Strassen's algorithm in class. Unlike all of the other algorithms presented here which have an $\Theta(n^3)$ runtime in the number of arithmetic operations, Strassen's algorithm only requires $O(n^{\log_2 7}) = O(n^{2.81})$ operations. Your task is to implement Strassen's algorithm. Note that unlike Task 8, Strassen's algorithm will require some intermediate results that need to be stored, as the intermediate steps are nontrivial. Pay special attention to make sure that your implementation is correct.

Strassen's algorithm is based on the observation that we can reformulate Equations (3) and (4) in the following equivalent way that only requires seven recursive calls. We first compute the following intermediate $\frac{n}{2} \times \frac{n}{2}$ matrices:

$$\begin{aligned}
&P_1 \leftarrow A_{11} + A_{22} && Q_1 \leftarrow B_{11} + B_{22} \\
&P_2 \leftarrow A_{21} + A_{22} && Q_2 \leftarrow B_{11} \\
&P_3 \leftarrow A_{11} && Q_3 \leftarrow B_{12} - B_{22} \\
&P_4 \leftarrow A_{22} && Q_4 \leftarrow B_{21} - B_{11} \\
&P_5 \leftarrow A_{11} + A_{12} && Q_5 \leftarrow B_{22} \\
&P_6 \leftarrow A_{21} - A_{11} && Q_6 \leftarrow B_{11} + B_{12} \\
&P_7 \leftarrow A_{12} - A_{22} && Q_7 \leftarrow B_{21} + B_{22}\,,
\end{aligned} \tag{5}$$

which will be followed by computing $M_i = P_i Q_i$ for each $i = 1, 2, \ldots, 7$ by invoking the Strassen's algorithm recursively on each pair of intermediate matrices. Finally, we compute

$$\begin{aligned}
C_{11} &\leftarrow M_1 + M_4 - M_5 + M_7 \\
C_{12} &\leftarrow M_3 + M_5 \\
C_{21} &\leftarrow M_2 + M_4 \\
C_{22} &\leftarrow M_1 - M_2 + M_3 + M_6\,.
\end{aligned} \tag{6}$$

Here, the submatrices A_{ij}, B_{ij}, C_{ij} should be shallow, but you will need to construct the intermediate matrices as copies.

Task 10 (Grade 12)**.** Implement Strassen's algorithm. Include a parameter s such that the algorithm falls back to a $\Theta(n^3)$ algorithm when the subproblem size is sufficiently small.

Task 10 amounts to implementing the function `strassen` in the Java library stub, following the Equations (5) and (6). A somewhat abstract pseudocode is shown in Algorithm 6. You can choose the exact $\Theta(n^3)$ algorithm together with the associated parametrization yourself; remember to justify your choices.

As before, since the algorithm has a parameter, we should explore its behavior with respect to this parameter.

Task 11 (Grade 12)**.** Fix n, and evaluate the best subproblem size where fallback from Strassen's to a $\Theta(n^3)$ algorithm should happen as implemented in Task 10. Justify your choice of n. Repeat the experiment with at least two other values of n. What kind of runtimes do you see? How does the algorithm behave with respect to the parameter? Why? Discuss your observations.

Algorithm 6 Pseudocode for Strassen's algorithm.

```
1: Input: $n \times n$ matrix $A$, $n \times n$ matrix $B$, minimum problem size $s$
2: Output: $n \times n$ matrix $C = AB$
3: procedure STRASSEN($A, B, C, s$)
4:     if $n = 1$ then
5:         $c_{11} \leftarrow c_{11} + a_{11}b_{11}$                ▷ Scalar multiplication
6:     else if $n \leq s$ then
7:         Compute $C \leftarrow AB$ with a $\Theta(n^3)$ algorithm    ▷ Non-trivial base case
8:     else
9:         Construct the $\frac{n}{2} \times \frac{n}{2}$ auxiliary matrices $P_1, P_2, \ldots, P_7$ and $Q_1, Q_2, \ldots, Q_7$ following Equation (5)
10:        Initialize the $\frac{n}{2} \times \frac{n}{2}$ auxiliary matrices $M_1, M_2, \ldots, M_7$ as zeros
11:        for $i \leftarrow 1, 2, \ldots, 7$ do
12:            STRASSEN($P_i, Q_i, M_i, s$)
13:        end for
14:        Store the appropriate linear combinations of the product matrices into the $\frac{n}{2} \times \frac{n}{2}$ submatrices of $C$ following Equation (6)
15:    end if
16: end procedure
```

8 Horse Race

We return back to the topic of McGeoch's methodology. Since we have several different algorithms, it will make sense to compare them against one another, try to determine their scaling, and whether it makes sense to pick one over another in different problem size ranges. That is, we should be able to make an educated choice of algorithm, depending on n. It will also be useful to include comparison to a third party implementation to see if the implementations are even in the same ball park with respect to runtime.

Task 12. Perform a horse race between the algorithms that you have implemented: the elementary algorithm of Task 1, the elementary algorithm that transposes the right-hand side operand matrix of Task 5 together with the best choice of the parameter s from Task 4, the tiled matrix multiplication of Task 6 with the best choice of parameter s from Task 7, the recursive matrix multiplication of Task 8 with the best parameter choice from Task 9, Strassen's algorithm of Task 10 with the best choice of parameter from Task 11. If you chose not to implement some algorithms, you can obviously leave them out.

For the transposed variant of the elementary algorithm, also measure and report the time required for the transpose of the operand (without performing the multiplication) for completeness.

Pay attention to how you generate your input, and remember to report how you did that as well.

Show how the different algorithms scale with respect to the side length of the input operands n in the form of a plot. Think how to normalize the plots so as to visualize the properties of the scaling between the different algorithms clearly.

Discuss your the results. Include discussion about the choice of the algorithm in different ranges of the problem size parameter n. Also discuss the choice of the parameters for each algorithm: what was the best choice, and why do you think that was the case.

9 Report

Write a report that includes a sufficiently detailed discussion of the work you have done, and that answers the tasks as laid out in this document. Use your judgement in how to report your results concisely but sufficiently accurately. Write your report as if writing to a peer who has not taken this class; you may assume the reader has basic data structures and algorithms knowledge, but is not particularly familiar with the topic.

Think also in terms of reproducability: the reader should be able to reproduce your results (including, but not limited to, the possible tables and plots that you have in your report) without consulting you for advice. Describe the environment where you ran the experiments in sufficient detail, including but not limited to the hardware, programming language, compiler or interpreter versions, libraries you used etc.

Use LaTeXto write a report in good style that does not exceed **6 A4 pages** in length. If you cite external literature, remember to include appropriate references. Extra tables, figures, or references after the main text do not count towards the page limit. You may also include an appendix (see the command `\appendix`) if you have results that you think may be helpful, but would otherwise obstruct the flow of the text. This can include lengthy tables, for example. If you do decide to include an appendix, mention it in your report, and refer to the results there.

Return all your work as a single **zip file** that includes your report in PDF format and also includes all of the code for reproducing your experiments.