HOMEWORK #1

SUBMISSION DUE DATE: 23/4/2020 23:00
SUBMISSION IN PAIRS

INTRODUCTION

Principal Component Analysis (PCA) is a statistical procedure for dimensionality reduction, enabling us to identify correlations and patterns in a data set so that it can be transformed into a data set of significantly lower dimension with minimal loss of important information.

In this assignment, you are asked to implement two C programs operating on matrices. The first program receives an input matrix and outputs its *covariance* matrix. The second program receives a covariance matrix and outputs an estimate of its *eigenvector* with the highest eigenvalue. Using these two, we can easily compute the first principal component, i.e., a single-dimension vector such that projecting the data onto it maximizes the variance of the projected points.

The purpose of this assignment is to practice all you have learned in C thus far, and to get familiar with Linux and the development environments.

1. Remotely connect to the Nova server and get familiar with basic shell commands
2. Create your first C programs
3. Compile, debug and basic use of makefile
4. Check your code according to the guidelines given in class

An executable tester is provided to you, and your submission will be graded accordingly by running the tester with your outputs on various inputs. The tester determines whether the output is correct, and you can use it to test your code. It is detailed later in this document.

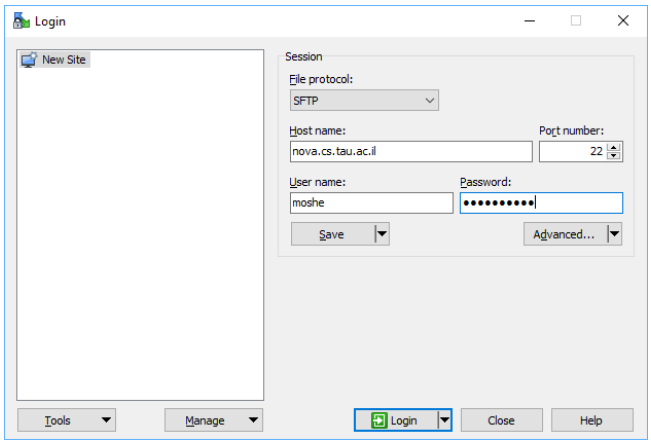
This assignment requires connecting to the faculty server, Nova. Do this as soon as possible, to ensure everything works. If you have issues connecting to Nova, please contact system: http://cs.tau.ac.il/system

DEVELOPMENT ENVIRONMENT

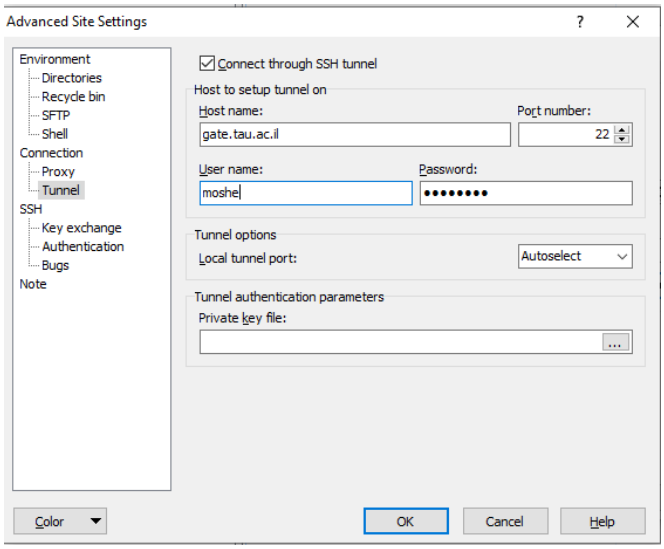
Students may work in any development environment they like. However, we encourage you to use eclipse as your development tool (we will not support issues regarding other IDEs or operating systems other than Linux or Windows). Your submission will be automatically checked by a script, so your code should run properly on **Nova** - the faculty server. You can use the computers in the PC farm or remotely connect to the server as described below.

FILE TRANSFER

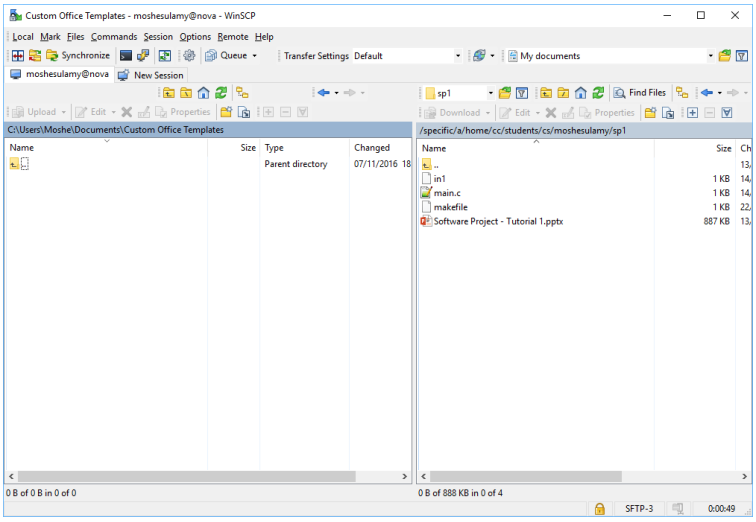
To transfer files to/from Nova, use WinSCP:



Before pressing Login, press "Advanced" to define a tunnel to *gate.tau.ac.il*:

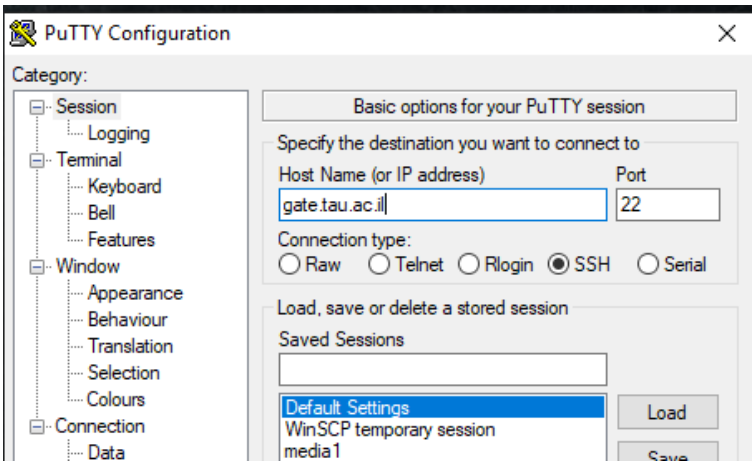


Once connected, drag files left (your computer) and right (Nova):



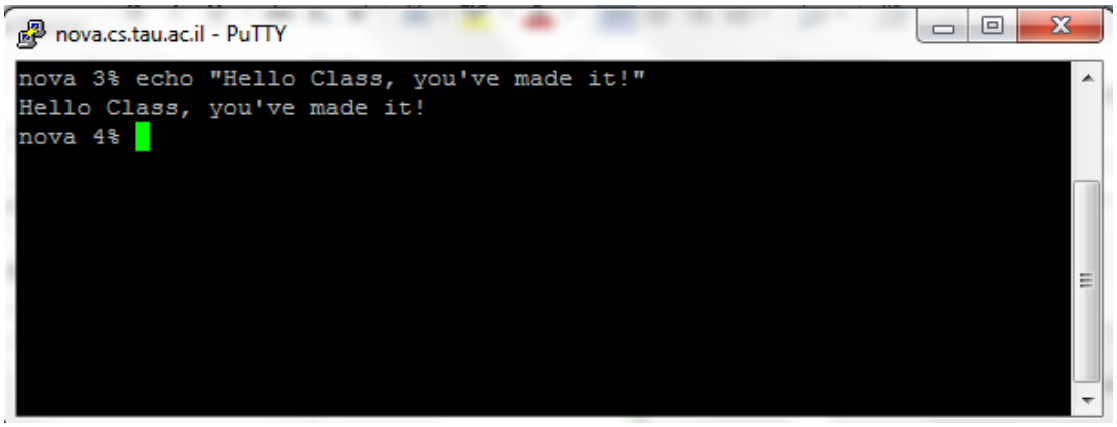
REMOTE CONNECTION TO NOVA

To connect to Nova, use SSH or PuTTY (*gate.tau.ac.il*):



After pressing open, enter your username and password for authentication (the username and password are the same as in your moodle authentication). Once connected, you should enter the following command: ***ssh nova.cs.tau.ac.il***

When successfully connected to Nova you will be able to see the terminal as in the following picture:



Now you can enter shell commands and execute them by pressing enter. Read the section below to find out more about shell commands.

(Note: echo is a shell command that is used to print strings to the standard output.)

USEFUL LINKS

You can download PuTTY and WinSCP by clicking on the following link. Recall that we recommend that you work with eclipse; please follow the installation guidelines for eclipse in moodle.

BASIC SHELL COMMANDS

After connection to Nova is established, you can type in shell commands and execute them by pressing enter. The results will be shown on the terminal (If there's any).

A few useful shell commands:

>> pwd

Prints the full pathname of the current working directory to the standard output. (The pathname is relative to the root directory which is the first directory in Linux).

>> ls [dir]

Lists the content of the directory "*dir*" (Both files and directories). If no parameters are given, the result is the content of the current working directory.

>> cd [dir]

Changes the current directory to be *dir.*
Use the following shortcuts:

. – This is a shortcut for the current directory.
.. – This is a shortcut for the parent directory in the hierarchy.
~ – this is a shortcut for your home directory.

>> mkdir [dirName]

Creates a new directory with the name dirName.

>> cp [file1] [file2] … [fileK] [dir]

Copies *file1,file2,…,fileK* to the directory *"dir".*
Note: To copy an entire directory (recursively copy a directory) use –r flag.

>> rm [file1] [file2] … [fileK]

Removes *file1,file2,…,fileK.*

>>man [command]

The man command is used to display the manual page of the command "*command*".
Note: you can navigate through the manual page using the arrows in the keyboard. To exit the manual page press "*q*"

>> diff [file1] [file2]

Prints the difference between the two files (file1 and file2)
Note: Use the *man* command to see the manual page of the command *"diff".*

>> chmod [options] [file]

Changes the permissions of [file] according to [options].
Note: When transferring executable files to *nova*, you need to execute *"chmod 777 <file>"* to be able to execute it.

MATRIX FILES

In this assignment, the input and output consist of **binary files** containing matrices.

Recall that binary files are not "human-readable" and cannot be edited in a text editor. This section will detail the exact file format, as well as provide sample C code for binary files.

FILE FORMAT

The start of each file contains two integer values, each with a size of 4 bytes (the size of *int* on most machines, including Nova). The 1st value is the number of columns in the matrix, and the 2nd value is the number of rows in the matrix.

The rest of the file are floating-point numbers (*double*), each with a size of 8 bytes, of the matrix values from left to right, then top to bottom.

Use **`sizeof(int)`** and **`sizeof(double)`** to properly get variable sizes in your code.

Throughout this assignment you may assume that all files are formatted correctly. However, you should still check for errors in opening and reading files (as described here). You may use **`assert`** to exit if any errors occur.

BINARY FILES IN C

Binary files in C are represented with a variable of type **`FILE*`**.

To open a binary file, we use the function **`fopen`**, which receives the filename as the 1st argument and the mode as the 2nd argument. Use mode **`"r"`** to open a binary file for reading, and **`"w"`** to open it for writing. Note that **`"w"`** creates a new file if it doesn't exist, and overwrites it if it does. If **`fopen`** fails it returns **`NULL`**, otherwise it returns **`FILE*`**, pointing to the file opened.

When done with a file, close it by calling **`fclose`**, which receives a single **`FILE*`** argument (the file to close). Note that it is important to close all files!
There is no need to check the return value of **`fclose`**.

Here is a code snippet that opens a binary file for reading, ensures it is opened correctly, and immediately closes it:

```
FILE* file = fopen("c:/testfile.arr", "r");
assert(file!=NULL);
fclose(file);
```

READING AND WRITING

To read and write binary files, we use the functions **`fread`** and **`fwrite`**, respectively.

Both functions work with elements. That is, we read and write an array (represented by a pointer) of elements, and specify the number of elements in the array and the size of each element. The functions return the number of elements read/written. A negative return value indicates an error. A return value smaller than the number of elements we attempted to read (including 0) indicates EOF – the "End of File" was reached.

Both functions receive the following arguments:

1. A pointer to where we wish to read data into, or write data from
2. The size (in bytes) of each element we read/write
3. The number of elements to read/write
4. The binary file (*FILE** variable)

The following example writes a single integer into the file, guaranteeing the write succeeds:

```
int val = 5;
n = fwrite(&val, sizeof(int), 1, file);
assert(n==1);
```

The following example reads 4 doubles from the file into *arr*, guaranteeing exactly 4 values were successfully read:

```
double arr[4];
n = fread(arr, sizeof(double), 4, file);
assert(n == 4);
```

Note: Use pointers and dynamic allocation in your code instead of constant-sized arrays.

REWIND

When reading or writing, we advance in the file so future reads are performed on the next data in the file, and further writes are written beyond what was already written.

Sometimes, we wish to return to the start of the file. In order to do so, we use **`rewind`**, which receives **`FILE*`** and resets its position to 0. It has no return value.

```
rewind(file);
```

COVARIANCE MATRIX

This section will detail the first program, creating a covariance matrix from an input file. Implement this program in the source file **`cov.c`**.

The covariance matrix is a square matrix giving the covariance between each pair of elements (rows) of a given input matrix. That is, element [x,y] of the covariance matrix is the covariance value (dot product) of rows x and y in the original input matrix. The diagonals are variances, i.e., the covariance of each element with itself.

The size of the covariance matrix is thus the number of rows in the input matrix, squared. The number of columns in the input has no effect on the covariance matrix size.

Note that the covariance is symmetric, i.e., elements [x,y] and [y,x] are equal.

The program receives two filenames as command-line arguments. The 1st refers to the input file, and the 2nd is the output file. You may assume both arguments are provided. The input file is a matrix, and the output file produced should contain its covariance matrix.

Important: the covariance matrix can be large, and we do not store it in memory. Instead, we create and write it to the output file one row at a time. You should, however, read the entire input matrix into memory and operate on it accordingly.

MAIN

The main function is the entry point of any C program. Its signature should be:

```
int main(int argc, char* argv[])
```

You can access the 1st argument (input filename) with **`argv[1]`**, and the 2nd argument (output filename) with **`argv[2]`**. You may assume both are always provided.

The covariance program should read the entire input matrix into memory, standardize it, and then write its covariance matrix into the output file row by row. Here is the flow of the program:

1. Read the entire input matrix into memory
2. Standardize the input:
    a. Calculate the mean of each row of the input matrix
    b. Subtract the mean of the corresponding row from each cell of the input matrix
3. Create the covariance matrix:
    a. Cell `[i,j]` is the dot product of row `i` with row `j` (both from the input matrix)
    b. Create the covariance matrix and write it to the output file one row at a time

Note: to calculate each row of the covariance matrix you need to iterate over the entire input matrix **once**. Try to be as efficient as possible in these iterations!

POWER ITERATION

This section will detail the second program: power iteration. Implement this program in the source file **`eigen.c`**.

Power iteration is an eigenvalue algorithm: given a symmetric matrix (covariance matrix, in our case), the algorithm produces the eigenvector corresponding to its greatest eigenvalue.

The program receives two command-line arguments. The 1st is the input file, the 2nd is the output file. You can access the 1st argument (input filename) with **`argv[1]`**, and the 2nd argument (output filename) with **`argv[2]`**. You may assume both are always provided. The input file is a symmetric matrix, and the output file produced should contain its eigenvector obtained using power iteration.

The power iteration algorithm starts with a random vector b_0, and in each iteration uses the current vector to produce a new vector, used in the next iteration. When done, the vector produced in the final iteration is the desired eigenvector.

To create b_0, fill it with random values. Randomization is detailed later in this document.

In the power iteration algorithm, in each iteration we obtain the next *normalized* vector according to the following equation:

$$b_{k+1} = \frac{Ab_k}{\|Ab_k\|}$$

Where *A* is our input covariance matrix, and the denominator is the vector's magnitude (i.e., square root of the vector's dot product with itself). To obtain the square root, use **`sqrt`**.

Recall that, as before, we do not store the entire covariance matrix in memory. Thus, we read the input file one row at a time. In each iteration of the algorithm, we also perform an iteration of the entire input file. Instead of opening and closing the file each time, use **`rewind`** (described above).

We are done iterating when the change in the new vector is small enough. When **all** of the differences (between each corresponding pair of values in the previous vector and the new one) are smaller than epsilon, we stop iterating and use the new vector as the eigenvector.

Use the function **`fabs`** to get the absolute value of a **`double`**. Use epsilon=0.00001.

RANDOMIZATION

Include **`<time.h>`** and add this call at the beginning of your program (only once!): **`srand(time(NULL))`**. This initializes the random number generator. To generate a random number, call **`rand()`**, which returns an integer between 0 and **`RAND_MAX`**.

ASSUMPTIONS AND REQUIREMENTS

You may make the following assumptions, and must follow the following requirements. Note that this list applies to both programs in this assignment:

- You may assume the input file is in the correct format (but may not assume that you succeed in opening it or reading/writing).
- You may use **`assert`** to exit on any error in accessing files or allocating memory.
- You may not change the supplied makefile or use functions from *math.h* except **`fabs`** and **`sqrt`**.
- You may assume input files do not lead to math errors, e.g., division by zero.
- Any further questions should be answered by using the supplied tester.

COMPILE, DEBUG AND MAKEFILE

As stated above, students may work in any development environment they choose. However, your code should compile and run on Nova. Please upload your source code and the makefile provided in the assignment to Nova (all the files in the same directory).

After connection is established, set your current directory as the source code's directory. Type in the following command and press enter to build your program, replacing *main.c* with the name of the file you are compiling (e.g., *cov.c*):

```
>>  gcc -ansi -Wall -Wextra -Werror -pedantic-errors  main.c -o hw1
```

The following command will compile your program and create a binary file called **hw1**. Note that the compilation flags used in the command above will be our default flags, so you should configure your eclipse installation with these flags (more info in moodle).

Now you can run your program by executing the following line on Nova:

```
>> ./hw1
```

Check your programs using the tester attached to the assignment (detailed below).

Before submitting your code make sure your code compiles using the makefile provided with the assignment. To do so, follow these instructions:

1. Copy your source code along with the file "makefile" into the Nova server. Place the files in the same directory.
2. Go the directory where you copied the files to, and simply write the following command: `>> make`
3. You can run your programs by executing **cov** and **eigen**, similar to **hw1** above.

Note: Please look at the makefile and read the guidelines in the course material for more information on makefiles. You will need to implement one on your own in future assignments.

TESTER FILE

Attached to this assignment is an executable file **tester**. It can generate random matrix files and check your output files. This tester is used by the graders! Using it is an integral part of the assignment. To use it, copy it to Nova and give execution permissions (see above).

- To generate a random matrix, run the tester with three arguments:
  1. Matrix filename (into which the matrix is written)
  2. Number of columns
  3. Number of rows

  For example: >> **./**tester input.arr 1000 2000
  Will create a matrix file named "input.arr" in the current directory with 1000 columns and 2000 rows.
- To check your output, run the tester with two or three arguments:
  1. Matrix filename (the input matrix)
  2. Covariance filename (your output from the 1st program)
  3. Eigenvector filename (your output from the 2nd program, optional)

  In case of errors, the tester specifies the first error that it detected and terminates.
  For example: >> **./**tester input.arr cov.arr eigen.arr
  Will check that the file "cov.arr" is the correct covariance matrix created from "input.arr" by the 1st program, and that the file "eigen.arr" is the correct eigenvector created from "cov.arr" with power iteration by the 2nd program.

TIME MEASUREMENTS

Your code is graded for performance! When your files are correct, the tester outputs the time each check took. Your code should exhibit roughly similar performance.

Include **<time.h>**. The **clock()** function returns a **clock_t** variable. Call it when you start and finish measuring time, and store both results. The timespan (in seconds) is:

```
((double)(end-start) / CLOCKS_PER_SEC)
```

FIX FOR ECLIPSE

When working with Eclipse on Windows, output (with **printf**) is not displayed correctly. To fix it, a file **SPBufferset.h** is attached to this assignment.

Place the file in the same directory as your source code, and add the following: the 1st below all other include statements, and the 2nd as the **first** call in your `main` method.

```
1. #include "SPBufferset.h"
2. SP_BUFF_SET();
```

You may leave these lines in your code when submitting. Do **not** submit the file SPBufferset.h.

SUBMISSION

Please submit a zip file named **id1_id2_hw1.zip** replacing *id1* and *id1* with the actual **9-digit** ID of both partners. If you submit alone, name it **id_hw1.zip**. Only a **single** partner should submit the assignment!

The zipped file must contain the following files (at the root, **no folders**):

- cov.c – Your source code for the 1st program.
- eigen.c – Your source code for the 2nd program.

Submit **only these files! Don't** add the makefile, SPBufferset.h, or any other files or folders.

MAC users: create (or check) your zip files under Linux, as otherwise hidden auxiliary files are automatically added and will reduce your grade.

You may leave any debug output (e.g., printf) in your code, but be careful – too many outputs will affect your performance.

REMARKS

- For questions regarding the assignment, please post in the forum.
- Late submissions are not acceptable unless you have the lecturer approval.
- Borrowing from others' work is unacceptable and bears severe consequences.

GOOD LUCK