INSTITUTE FOR
MICROELECTRONICS

2022W

nssc@iue.tuwien.ac.at

Numerical Simulation and Scientific Computing I

Exercise 1

General Rules and Tips

1. The group representative must submit the solutions by **November 3th, 2022 at 08:00** to TUWEL as **one compressed ZIP file per task** (max ZIP file size is 10MB) **containing one folder per task**.
2. Each task's folder must include all source codes, `Makefiles`, I/O files, plots in `.png` format, and answers in a plain `.txt` file.
3. The TUWEL course contains an "Exercises" section and in it an "Exercise 1" section. You will be able to submit the ZIP file there. Only one member of the group is required to submit the files. All group members will get the same points.
4. The code examples must be developed in a GNU/Linux development environment: You can find on TUWEL a guide how to install a virtual GNU/Linux operating system suitable for the exercises.
5. List the members of your group by name and student ID number in the `.txt` file.
6. Each task should be submitted in a self-contained way so that the code compiles and executes without any file dependencies from other tasks submitted with your compressed files.
7. Some tasks require you to use a specific filename, be aware of that!
8. Your C++ binary should execute all subtasks when invoked from console without additional parameters.
9. All C++ floating-point numbers should be `double`.
10. Use of modern C++ standards will be appreciated.
11. You are only allowed to use C++ standard libraries unless otherwise indicated.
12. Do feel encouraged to reuse code between tasks (but copy the code; don't introduce inter-task dependencies, see above.)
13. You may write additional functions in addition to the ones required for the tasks.
14. Google, Stack Overflow, and other platforms can be useful resources and using those is in principle not forbidden. But, you must let us know in the `.txt` if you did so and provide the URL to the source. However, do not copy code without thinking (keep the challenging exam in the end in mind!)
15. Should there be any questions: ask before guessing!

Task 1 – Basic Exercise (3 points)

Develop a small C++ program which operates on text files as described below.

a) Create a datatype (e.g. `struct` or `class`) to store a string holding the CPU vendor and another string for the CPU model. Additionally add integer numbers to store the number of cores, bit-architecture, and number of transistors.

b) Write a function which reads a plain text file containing per line `vendor, model, number of cores, bit-architecture, and number of transistors`. All input data from a file should be stored in an STL container (e.g., `std::vector`). The element type of the container should be your struct/class datatype from a).

c) Write a function which returns the CPU model with the largest core count for the data stored in one container populated in b).

d) Write a function which returns the average transistor count for the data stored in one container populated in b).

e) Write a function which returns the transistor/core ratio of a single element stored in one container populated in b).

f) Read the three input files `1999_CPUs.txt`, `2006_CPUs.txt` and `2022_CPUs.txt` and store the data in separate containers (one container per file), see b), and store the year for each container (extract the year from the input file names with string operations). The filenames should be provided via command line arguments.

g) Save to a single plain text file (`CPU_analysis1.txt`) the CPU model with the largest core count and its transistor/core ratio over all three datasets, see f). Separate the data with a comma. The output file should thus contain only a single line.

h) Save to a single plain text file (`CPU_analysis2.txt`) the year and the average transistor count for each data set in a new line. The output file should thus contain three lines.

i) Calculate (preparation for subtask j)) the average increase in transistor count per year (absolute and relative) from 1999-2006, 2006-2022, and 1999-2022.

j) Save to a different plain text file (`CPU_analysis3.txt`) a list with the following **three** columns separated by comma and output the data accordingly: `from-to, absolute change, relative change.`
For instance: `"1999-2006, 100000, 34.13"`
The output file should thus contain three lines.

Additional information:

-Regarding subtask j): The relative change must be outputted in percent with fixed floating-point notation and with at most 2 digits after the decimal point.

-Together with the above-named output files, also provide your source code, and the `Makefile`.

Task 2 – Plotting with Matplotlib/NumPy (2 points)

Use Python to generate the following plots:

a) Using NumPy's `genfromtxt`, read the text file you generated from item h) of Task 1 (`CPU_analysis2.txt`). Using Matplotlib, make a *bar chart* showing the growth of the average transistor count per year. Each bar should correspond to a year. The x-axis should show the year for each bar. Save the result as `transistor_growth.png`.

b) Using the Python library `pandas` and its `read_csv` function, read the text file you generated from item j) of Task 1 (`CPU_analysis3.txt`). Use `pandas`'s `plot.bar` function and `matplotlib` to create a **colored** *grouped bar chart* of the absolute and relative growth (the grouping should be for the two types of change) for all three "year-from-year-to" pairs. Ensure that you have two different y-axes and **deal with the different orders of magnitudes** of the data sets (ensure proper y-axes scaling). Assign all axes labels (in particular, the x-axis should show the "year-from-year-to" label for each group.) **and** add a color legend. Save the result as `transistor_change.png`. In total, the graph should show three 2-column groups.

Additional information:

-Also submit the scripts `transistor_growth.py` and `transistor_change.py` you created to generate the plots!
-Do not forget to submit your input text files `CPU_analysis2.txt` and `CPU_analysis3.txt` for this task too (copy it from the results of Task 1). Should you not be able to finish Task 1, you can manually create your own input files.
-You can take inspiration from Matplotlib's Gallery.

Task 3 – Basic Linear Algebra (3 points)

Develop a small C++ program which performs basic linear algebra operations as described below.

a) Write a function which reads a matrix in plain text format and stores it in a container (e.g. `std::vector` or array).
b) Write a function which reads a vector in plain text format and stores it in a container (e.g. `std::vector` or array).
c) Write a function which returns the result of the *outer* product of two vectors (result is a matrix).
d) Write a function which returns the result of the *dot* product of two vectors (result is a scalar).
e) Write a function which returns the result of a matrix-matrix multiplication.
f) Write a function which returns the result of a scalar-matrix multiplication.
g) Read in a matrix A: `MatrixA.txt`
h) Read in a vector v: `VectorV.txt`
i) Calculate and save to different plain text files the following results:
   a. *outer* product: $v \otimes v$, file name: `vxv.txt`
   b. *dot* product: $v \cdot v$, file name: `vdv.txt`
   c. $(v \cdot v)(A(v \otimes v))$ file name: `vdvAvxv.txt`

Additional Information:

-You are not allowed to use any libraries except for C++ standard libraries.

-Together with the above-named files, also provide a `Makefile` you generated.

-Each line of the matrix files contains a row, with elements separated by a single space.

-Each line of the vector file contains a single vector entry.

-All matrices are square.

-As a reminder, the elements of the matrix $C = AB$ are $C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$

Task 4 – Compiler-Driven Optimization (2 points)

You will run vector-matrix multiplication and evaluate the runtimes based on different compiler flags.

a) Generate a matrix A such that: $A_{ij} = (i + 1) \cdot j$. The matrix must have 5 000 elements in each dimension.
b) Generate a vector v such that: $v_i = (i + 2)$. The vector must have 5 000 elements.
c) Implement a function which calculates the matrix-vector multiplication.
d) Use C++'s `<chrono>` library to calculate the runtime in seconds of the matrix-vector product. Use `std::chrono::steady_clock;`
e) Read the `g++` compiler options. Try to identify optimization flags other than the ones already included in `-O3`. Choose 2 different flag combinations for compilation.
f) Compile your code using `GCC` and the following options separately:
   a. `-O0`
   b. `-O1`
   c. `-O2`
   d. `-O3`
   e. Your flag combination 1
   f. Your flag combination 2
g) Run each version of the program 8 times and record the runtimes.
h) Plot the average runtime as a function of the optimization flag and save it as `optimization_flag.png` using Matplotlib.

Additional Information:

-Together with the above-named files, also provide your source code and the `Makefile` you generated. Also, provide the Python script `optimization_flag.py` you used to generate the plot.

-Only the runtime of the matrix-vector multiplication should be recorded and not the whole program!