CME 213, Introduction to parallel computing
Eric Darve
Spring 2023

Stanford University

Homework 1

Total number of points: 100.

Deliverables: Turn in all your code and a brief writeup (as a PDF on Gradescope). Detailed instructions for creating the submission can be found at the end of this document. When writing a test, print appropriate outputs showing the test results to the standard output. Run your code with the test and make sure to include the output of the test in your writeup.

You will find the starter code on Canvas under `Assignments`.

Please do not modify the file names or the Makefile.

```
$ make
```

will make all the files

```
$ make main_q1
```

will only make problem 1, etc.

Because of missing pieces of code that you need to fill in, the starter code for Problem 2 will not successfully compile.

Have fun coding!

Problem 1 C++ derived classes and matrix class

25 points. Assume that we want to create a `C++` library for matrices. Since matrices often have structure, it doesn't make sense to create one class for all matrices. For example, a diagonal and a dense matrix have very different storage requirements and using a dense matrix class to store both would be very inefficient. A better approach is to define an interface, to which all matrix classes must adhere and then create different implementations for different matrix structures.

We want to explore the use of inheritance by implementing a `C++` class diagonal matrices and another class for symmetric matrices. This class should have at least the following properties:

- Inherit from a *pure abstract* base class for general matrices.
- The classes use a template argument for the type of matrix entries. Assume that the type supports all arithmetic operations: `+`, `-`, `*`, `/`, such as `float` or `double`. For simplicity, you may consider that the templated type is either `float` or `double`.
- It should accept a constructor with input argument `n`, the size of the matrix. The matrix then has `n` rows and `n` columns.
- Design the storage of the matrix elements such that for diagonal matrices, the storage of the elements should be $n + O(1)$ and for symmetric matrices, the storage should be $n(n + 1)/2 + O(1)$ where n is the matrix size.
- You should define an `size()` function to return the dimension of the symmetric matrix (number of rows or number of columns of the square matrix).
- You should define an `operator()` to access and modify entries in the matrix. The operator should take as input a row `i` and a column `j`.

- You should define an operator << to pretty print the entire matrix. When the << operator is used on any matrix, the elements of the matrix should be printed in order as a string. Consecutive columns should be separated by some amount of whitespace, and consecutive rows should be separated by a newline.
- You should define a method to calculate the ℓ_0 "norm" (the number of non-zero elements).

Implement the `Matrix`, `MatrixSymmetric`, and `MatrixDiagonal` classes in the given `Matrix.hpp` file. Turn in your code.

Testing your code It's an important skill to know how to test code. We want you to write test cases for your code to pass in the `main_q1.cpp` file. This is an example of a good coding practice. We would like to see tests for different matrix sizes, getting and setting matrix entries and verifying symmetry, pretty printing your matrix, as well as the ℓ_0 "norm". Recall that the ℓ_0 "norm" is the number of nonzero elements in the matrix.

In this homework, the testing framework that we will use for writing our tests is GoogleTest. The hyperlinked GitHub page is your best source of information about this test suite. Don't worry; we have provided all the necessary GTest files along with the starter code, correctly set it up for you, and provided a sample test case in `main_q1.cpp` to get you started.

The tests your code needs to pass are described in `main_q1.cpp`; please implement them all for both the `MatrixSymmetric` and `MatrixDiagonal` classes. Does your code pass all these tests? Please turn in the output of your code in your write-up.

Problem 2 Elementwise broadcasting

15 points. One of the interesting features of standard linear algebra libraries like numpy is the possibility of operand broadcasting. Broadcasting allows performing element-wise operations between matrices even if their dimensions do not match. This can be done under certain assumptions for the matrix sizes. See
`https://numpy.org/doc/stable/user/basics.broadcasting.html`
for information about operand broadcasting.

Given to you is a code skeleton in `matrix_rect.hpp` which implements rectangular matrices which need not be square in shape. To start off, copy your code from Problem 1 to fill out the following functions in `matrix_rect.hpp` (marked by `TODO`): the two constructors for `Matrix2D`, `size_rows`, `size_cols`, `operator()`, `Print` and `operator<<`. The implementations of these functions will closely follow your solution for Problem 1. The only difference will be how the `data_` vector is sized and organized for 2D rectangular matrices, and how to index into it. Please follow the comments and hints given in the starter code to implement this.

In this problem, we will implement operand broadcasting. We only consider 2D rectangular matrices and the operation of elementwise multiplication of matrices of the same shape. Given to you is a code skeleton in `matrix_rect.hpp` which implements rectangular matrices which need not be square in shape. When the two operand matrices have the exact same shape, elementwise multiplication can be performed directly with no need for broadcasting. However, say that A has shape $(3, 3)$ and B has shape $(1, 3)$, then B will be broadcast to perform elementwise multiplication and the result will have shape $(3, 3)$. In this problem, given two matrices that do not have the same shape, we want you to write code that identifies whether broadcasting is possible, and if it is, to compute the final result. In `matrix_rect.hpp`, you must fill in the blank parts of the code marked by `TODO` to correctly implement broadcasting for elementwise multiplication of two matrices of shapes (m_1, n_1) and (m_2, n_2). You can read about the constraints on the shapes of the two matrices that allow for broadcasting in the link above. These constraints will be implemented by you in the function `Broadcastable` and the actual operation will be implemented inside the function `dot`.

Just like in Problem 1, you should test your broadcasting code thoroughly by implementing test cases in the `main_q2.cpp` file. The comments in that file describe a few test cases that you should consider writing with the Google Test suite.

Problem 3 C++ polymorphism

10 points. Assume you completed the matrix library in Problem 1 and are writing a function that needs to use a sequence of matrices as input. The input argument should be a `std::vector` of matrices. Since all matrices implement the same `Matrix` interface, and therefore support the same basic operations such as addition and multiplication, we would like to use that `Matrix` interface rather than the matrix classes for specific cases. This will allow appending different kinds of matrices to the same `std::vector`. Please complete the code given in `main_q2.cpp` and demonstrate how this can be done.

Problem 4 C++ set using the standard library

15 points. You are running a Monte Carlo simulation and would like the ability to quickly query the number of samples in a range given by [`lb`, `ub`]. There are many ways to achieve this, but one possibility is to store all samples in an `std::set`. Although `std::set`s have many similarities to the mathematical notion of a set, they have the additional property that they are sorted.[1] For simplicity, we assume that each data point is unique.[2] Write a function that takes the following parameters:

- A set containing the data, `std::set data`,
- A range given by `double lb` and `double ub`.

And returns the number of data points within the range. You need to use the `std::set::lower_bound` and `std::set::upper_bound` functions.

Please complete the code given in `main_q3.cpp`. Turn in your code and the output.

Problem 5 C++ standard library

The following are short problems involving the C++ standard library. The code skeleton has been provided for you, but you will need to implement the test code, which is marked with `TODO`s in the code. Finally, you are not allowed to use any loops anywhere outside of your tests. Instead use `std::for_each`, `std::transform`, `std::sort`, `std::all_of` from `algorithm`.

(a) 10 points. Implement DAXPY, where DAXPY is a shorthand for $ax + y$ where `x` and `y` are vectors containing doubles, and `a` is a double. Your DAXPY function should return a new vector with the result of this operation. Implement this function, verify its correctness with a test, and turn it in.

(b) 10 points. You are a professor, and you need to compute your students' grades. You want to see if everyone has passed or not. For your class, you have determined the following weights: homework is 20%, midterm is 35%, and the final exam is 45%. To pass, a student must be equal to or above 60%. Implement the `all_students_passed` function, verify its correctness with a test, and turn it in. Assume all values are percentages, and are in the range `[0, 1]`. Use the C++ standard library for this.

[1] This is a consequence of the fact that they are implemented as binary search trees

[2] Otherwise we would need to use the `std::multiset`, which lifts the requirement that each entry be unique

(c) 5 points. Sort a list of integers such that the odd numbers come first, and then the even numbers. The numbers within each odd and even number section should also be sorted ascending. For example, given the vector `[4, 2, 5, 3, 0, 1]`, your function should output `[1, 3, 5, 0, 2, 4]`. Implement the `sort_odd_even` function, verify its correctness, and turn it in.

(d) 10 points. One way to implement a sparse matrix is to use a linked list, where each node holds the tuple (i, j, val), where i is the row, j is the column, and val is the nonzero value at that location. To improve random access times, it is best to keep this list sorted. To sort this list, we want elements with smaller row numbers to be toward the head of the list. If we have two nonzero elements on the same row, the one with the smaller column index will come first. To visualize this, imagine flattening the matrix into a 1D array. Each nonzero value in the sorted linked list will thus point to the next nonzero value in the sparse matrix. Implement the `sparse_matrix_sort` function, verify its correctness with a test, and turn it in. You may add public members to the `SparseMatrixCoordinate` struct.

6 C++ Standard Library Functions

This homework uses quite a few functions from libraries such as `algorithm`, `functional`, and `numeric` and containers such as `vector` and `list`. We have compiled a complete list of all C++ Standard Library functions that may be helpful in this assignment along with its associated header and a brief description.

Function	Header	Description
std::all_of	algorithm	Returns true if all elements fulfill the given predicate, false otherwise.
std::distance	algorithm	Returns the number of "hops" between two iterators.
std::for_each	algorithm	Applies the given predicate for each element.
std::sort	algorithm	Sorts all elements given a predicate or the default < operator.
std::transform	algorithm	Applies a predicate to each element in a src container and places the result in a dst container.
std::accumulate	numeric	Given an initial value v, predicate f, and element x from container X, compute $v + \sum_{x \in X} f(x)$. Similar to Python's reduce.
std::iota	numeric	Fills a container with [i, i+1, i+2, ...] given some initial i.
std::list<T>	list	A doubly linked list container.
std::list<T>::sort	list	Specialized sort for linked lists. Not all sorts work on linked lists.
std::vector<T>	vector	A resizable array-backed list.
std::set<T>	set	A sorted container.
std::set<T>::lower_bound	set	Gets an iterator to the first element not less than the given value.
std::set<T>::upper_bound	set	Gets an iterator to the first element larger than the given value.
std::default_random_engine	random	Create an instance to use for anything that needs a PRNG.
std::normal_distribution	random	Generates normally distributed values with given mean and standard deviation.
std::cout	iostream	Use this to print out stuff to the console.
std::ostream	ostream	Use this when overloading the << operator.
std::stringstream	sstream	Useful when you want to save the << operator output into a string.
std::stringstream::str	sstream	Retrieves the string inside stringstream.
std::exception	stdexcept	Base exception class. Useful to catch for tests.
std::runtime_error	stdexcept	Thrown during a runtime error. Useful to catch for tests.
std::invalid_argument	stdexcept	Throw this when you encounter an invalid argument.
std::out_of_range	stdexcept	Throw this when you encounter something out of range.

7 Submission instructions

To submit:

1. For all questions that require explanations and answers besides source code, put those explanations and answers in a separate PDF file. Upload this file on Gradescope.

2. The rest of the files (Makefile, code, etc) should be submitted using a submission script on cardinal. The submission script must be run on cardinal.stanford.edu. It will not work from rice.

3. Copy the directory containing all your submission files to cardinal.stanford.edu. You can use the following command in your terminal:

```
scp -r <directory to be submitted> <your SUNetID>@cardinal.stanford.edu:<your directory>
```

   Here is the list of files we are expecting:

```
main_q1.cpp
```

```
main_q2.cpp
main_q3.cpp
main_q4.cpp
main_q5.cpp
matrix.hpp
matrix_rect.hpp
```

4. Make sure your code compiles on `rice` and runs. To check your code, we will run:

   ```
   $ make
   ```

   This should produce 5 executables: `main_q1`, `main_q2`, `main_q3`, `main_q4` and `main_q5`.

5. Install `python-dateutil`. Type:

   ```
   $ pip3 install python-dateutil
   ```

   This is a one-time operation that is required to run the Python submission script below.

6. Type:

   ```
   $ /afs/ir.stanford.edu/class/cme213/script/submit.py hw1 <directory with your submission files>
   ```

   The `submit.py` script will copy the files listed above to a directory accessible to the CME 213 staff. Only files in the list above will be copied. Make sure these files exist and that no other files are required to compile and run your code. In particular, do not use external libraries, additional header files, etc, that would prevent the teaching staff from compiling the code successfully. The script will fail if one of these files does not exist.

7. You can submit at most 10 times before the deadline; only the last submission will be graded.

You may review your submission by typing in the following command in your terminal while you are on `cardinal`:

```
ls /afs/ir.stanford.edu/class/cme213/submissions/hw1/<your SUNetID>/<submission number>
```

In this directory, because of the ACL permissions,[3] you are only authorized to list and create new files. You cannot read, move or change the content of the files inside those directories. It is a violation of the honor code to submit your homework files without using the script provided by the CME 213 staff.

[3] https://uit.stanford.edu/service/afs/sysadmin/userguide/filepermissions