CME 213, Introduction to parallel computing
Eric Darve
Winter 2020

Stanford University

Homework 1

Total number of points: 100. Late day policy: 2 late days with a 10% grade penalty. Please do not modify the file names or the Makefile.

```
$ make
```

will make all the files

```
$ make main_q1
```

will only make the first problem, etc.

Enjoy!

Problem 1

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 for symmetric matrices. This class should have at least the following properties:

- Inherit from a *pure abstract* base class for general matrices.
- Use a template argument for the type of the matrix entries. Assume that the type supports all arithmetic operations: `+`, `-`, `*`, `/`, such as `float` or `double`.
- It should accept a constructor with input argument `n`, the size of the matrix.
- The storage should be $n(n+1)/2 + O(1)$ where n is the matrix size.
- 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.
- You should define a method to calculate the ℓ_0 "norm" (the number of non-zero elements).

(a) 15 points. Implement the `Matrix` and `MatrixSymmetric` class in the given `Matrix.hpp` file. Turn in your code.

(b) 10 points. We want you to demonstrate that you know how to write correct code, which includes knowing how to test your code. 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.

Describe the operations that you want your class to correctly support and implement. Explain which tests you want to write to check whether your class correctly implements the features you want.

(c) 10 points. Write and turn in code that implements these tests. Run your code. Did your class pass all your tests?

Problem 2

15 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 3

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.

Use the following code to generate some test data:

```
#include <random>
#include <set>
  ...
std::set<double> data;
std::default_random_engine generator;
std::normal_distribution<double> distribution(0.0, 1.0);
for (unsigned int i = 0; i < 1000; ++i)
    data.insert(distribution(generator));
```

and report the number of points in the range `[lb, ub] = [2, 10]`. Turn in your code.

Problem 4

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 are 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.

[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

(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 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.

(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 sections 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 towards 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.

5 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 not 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.

6 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
matrix.hpp
```

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

   ```
   $ make
   ```

   This should produce 4 executables: `main_q1`, `main_q2`, `main_q3`, and `main_q4`.

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`