The Sparse Matrix Class

COMP3171/COMP9171 11s2 Deliverable 2 Specification

Version 1.0, Sun Aug 14 09:00:46 EST 2011

1 Introduction

Matrices, usually two-dimensional tables of scalar values, are an important tool in mathematics, engineering and science. You will be constructing a C++ class that is able to encode a matrix, together with a series of matrix operations.

The horizontal lines of a matrix are called *rows* and the vertical lines are called *columns*. A matrix with m rows and n columns is referred to as an m-by-n ($m \times n$) matrix. The entry at row i and column j is the (i, j)-th entry. For example, a 4×3 matrix:

$$\mathbf{A} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \\ a_{3,0} & a_{3,1} & a_{3,2} \end{bmatrix}$$

Note that matrix rows and columns are zero-indexed. Various operations can be performed on matrices. Addition and subtraction, defined for matrices of identical dimensions and achieved by element-wise addition and subtraction, respectively. Multiplication, defined for an $m \times n$ matrix multiplied with an $n \times p$ matrix and achieved via row-by-column multiplication. Transposition, achieved by turning rows into columns and vice-versa, and more...

Sparse matrices are matrices with relatively few non-zero elements (as opposed to 'regular', *dense* matrices that have few zero elements). Working with complex partial differential equations typically involves very large sparse matrices. It quickly becomes impractical to store such matrices in a dense format, e.g., a $10,000 \times 10,000$ matrix of integers would require $\sim$ 400MB of space. Instead, you will use a much more compact sparse representation, together with appropriately optimised operations.

Such a sparse matrix class might appear in a C++ high-performance mathematics library.

2 Aims

The idea of this deliverable is to provide you with first-hand experience implementing a complex C++ class from scratch. You will employ the following C++ concepts:

- constructors
- copy control
- destructor
- data and function members, including static members
- overloaded operators
- class friends
- simple iterators

- exception handling
- explicit dynamic memory management
- separation of interface and implementation

3 Requirements

You are required to write a C++ class for sparse matrices with integer (built-in `int`) values. The name of the class will be `SMatrix`. Following are the requirements for the internal representation, constructors, operations, overloaded operators, friends, the destructor and error handling.

3.1 Internal Representation

Various schemes exists for efficiently storing sparse matrices. Even the naïve approach of storing (*row, column, value*) triples is a vast improvement over dense storage. For the purposes of this deliverable you are required to use the following sparse storage format for an $m \times n$ matrix with k non-zero entries:

1. a pointer, `vals`, to an array (size k) of `int` representing the non-zero matrix entries
2. a pointer, `cidx`, to an array (size k) of `size_type` representing the matrix column indices corresponding to each of the entries above
3. a `map`, `ridx`, of row indices to `pair`s of indices (into `vals` and `cidx`, corresponding to the first non-zero entry in each matrix row) and non-zero entry counts

Consequently, your `SMatrix` class will include:

```
private:
  int *vals_;
  size_type *cidx_;
  std::map< size_type, std::pair<size_t, unsigned int> > ridx_;
```

Consider the following 8×7 matrix with 15 non-zero entries:

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & 0 & 2 & 0 & 0 & 3 \\ 4 & 5 & 0 & 0 & 0 & 0 & 0 \\ 0 & 6 & 7 & 0 & 0 & 8 & 0 \\ 9 & 0 & 0 & 10 & 11 & 12 & 0 \\ 0 & 13 & 0 & 0 & 14 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 15 \end{bmatrix}$$

We would encode it as:

$$\begin{array}{lcl} \texttt{vals} & \rightarrow & \{\,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\,\} \\ \texttt{cidx} & \rightarrow & \{\,0,3,6,0,1,1,2,5,0,3,4,5,1,4,6\,\} \\ \texttt{ridx} & \rightarrow & \left\{ \begin{array}{ccc} 0 & \mapsto & (0,3) \\ 1 & \mapsto & (3,2) \\ 2 & \mapsto & (5,3) \\ 3 & \mapsto & (8,4) \\ 4 & \mapsto & (12,2) \\ 7 & \mapsto & (14,1) \end{array} \right\} \end{array}$$

The map `ridx` stores an entry for each row that contains at least one non-zero element. This encoding scheme allows very large matrices with few non-zero elements to be stored compactly, since entries are not required for all-zero rows.

Thus the non-zero values in the i-th row are:

```
vals[ridx[i].first]      @ (i, cidx[ridx[i].first]),
vals[ridx[i].first + 1]  @ (i, cidx[ridx[i].first + 1]),
vals[ridx[i].first + 2]  @ (i, cidx[ridx[i].first + 2]),
... ,
vals[ridx[i].first + (ridx[i].second - 1)]
                         @ (i, cidx[ridx[i].first + (ridx[i].second - 1)])
```

It is not necessary in general to keep the range of values in `vals` and `cidx` (corresponding to each row) ordered; however, you are required to keep them in increasing column-index order for the purposes of this deliverable. This allows implementation uniformity and further optimisations.

Make sure that you fully understand the details of the encoding before you proceed with writing the code of your implementation.

Once you read the interface requirements, you will notice that, while the size of the matrix is fixed upon construction, the number of non-zero values may fluctuate, requiring possible resizing of the representation arrays. You should use the following simple amortisation technique, assuming that you are working with an $m \times n$ matrix:

- the `vals` and `cidx` arrays should be sized to $min(\lceil (m \times n) \div 5 \rceil,\ 1000)$ upon construction (with the constructor that only specifies an initial matrix size)
- the `vals` and `cidx` arrays should be doubled in size upon reaching capacity

Intuitively, we assume that only a small number of matrix entries will be non-zero and resize as necessary. Better amortisation techniques exist, but this suffices for our purposes.

When non-zero matrix values are set to zero (i.e., via `setVal`) you are required to reorganise the internal representation to remove the new zero value from `vals` and `cidx`, and update `ridx` as appropriate (this includes potentially removing an entry from `ridx` if a row no longer has any non-zero entries). However, you are *not* required to downsize the representation arrays.

You *absolutely must* use dynamically allocated arrays to store the matrix representation. (Please make sure you use `new` and `delete`!) Therefore, the default copy constructor, copy assignment operator and destructor will not provide the correct semantics. You must provide your own implementations for these operations.

NB: Failure to follow the prescribed internal representation for your matrix will result in a total mark of 0 for this deliverable.

3.2 Constructors

Your design should allow matrices to be defined in three ways, illustrated below:

Constructor	**Side-Effects**
`SMatrix a(3, 5);`	Matrix `a` is constructed as a 3 × 5 matrix (3 rows, 5 columns), with all entries zero.
`SMatrix b(cin);`	Matrix `b` is constructed based on a serialised representation read from the input stream `cin`.
`SMatrix c = a;`	Matrix `c` is constructed and copy-initialised (value semantics) with matrix `a`. Matrix `c` is a copy of matrix `a`; it has the same dimensions and values.

Note, in the definition of `b` above, the constructor argument is an input stream containing a serialised representation of a sparse matrix. This representation is the same as the one produced by the output operator, see §3.5.1.

For all constructors, you may assume that the arguments supplied by the user are *correct.* This means that all row and column sizes given are positive (> 0), and the serialised representation is well-formed and has the correct number of values.

3.3 Operations

The (public) operations shown below should be supported by matrices.

Operation	Preconditions	Side-Effects	Return Value
`a = b` (copy assignment)	none	`a` becomes a copy of `b` (value semantics)	a reference to `a`
`a += b`	`a` and `b` have the same dims.	`a` is replaced with the sum of `a` and `b`	a reference to `a`
`a -= b`	`a` and `b` have the same dims.	`a` is replaced with the difference of `a` and `b`	a reference to `a`
`a *= b`	`a.cols() == b.rows()`	`a` is replaced with the product of `a` and `b`	a reference to `a`
`a(i,j)`	values of `i` and `j` are in bounds	none	the value of the (i, j)-th element of `a`
`a.rows()`	none	none	the number of rows in `a`
`a.cols()`	none	none	the number of cols. in `a`
`a.setVal(i,j,v)`[1]	values of `i` and `j` are in bounds	the value of the (i, j)-th element of `a` is set to `v`	`true` if additional memory must be allocated and `false` otherwise
`a.addRows(i,j)`	values of `i` and `j` are in bounds	row `i` of `a` is replaced by the sum of rows `i` and `j`	none
`a.addCols(i,j)`	values of `i` and `j` are in bounds	column `i` of `a` is replaced by the sum of columns `i` and `j`	none
`a.subRows(i,j)`	values of `i` and `j` are in bounds	row `i` of `a` is replaced by the difference of rows `i` and `j`	none
`a.subCols(i,j)`	values of `i` and `j` are in bounds	col. `i` of `a` is replaced by the difference of cols. `i` and `j`	none
`a.swapRows(i,j)`	values of `i` and `j` are in bounds	row `i` of `a` is swapped with row `j`	none
`a.swapCols(i,j)`	values of `i` and `j` are in bounds	column `i` of `a` is swapped with column `j`	none
`a.begin()`	none	set an internal pointer to the first element of `a`	none
`a.end()`	none	none	`true` once internal pointer goes past the last element of `a` and `false` otherwise
`a.next()`	none	move the internal pointer to the next element of `a`	none
`a.value()`	none	none	the element value pointed to by the internal pointer
`identity(n)` (`static` member)	`n` > 0	none	a new matrix, the `n` × `n` identity matrix

You should *not* modify or augment the public interface provided.

[1] This operation runs in something like $O(k)$ time for matrices with k non-zero elements. Not particularly efficient, but reasonable when a matrix requires only a few tweaks.

3.4 Iterators

The four member functions, `begin()`, `end()`, `next()` and `value()`, provide a *fake* iterator for enumerating all the values (including zero values) in a matrix. The values should be enumerated in row-major order. You might use these functions like this:

```
SMatrix m(3, 4);
for (m.begin(); !m.end(); m.next())
  std::cout << m.value() << " ";
std::cout << std::endl;
```

This 'iterator' is not nearly as powerful as a proper iterator, but it should suffice to give you an idea of what iterators are all about. You must make sure that these member functions can be invoked correctly on both `const` and non-`const` `SMatrix` objects.

The user is expected to guard against invalidated iterators and no error checking is required. *Hint:* All these functions have short implementations. Review the `mutable` qualifier.

3.5 Friends

In addition to the operations indicated earlier, the following operations should be supported as friend functions. Note that these friend operations don't modify any of the given operands.

Operation	**Preconditions**	**Side-Effects**	**Return Value**
`a == b`	none	none	`true` if `a` and `b` have the same dimensions and $\forall\ i,j\ \ \mathtt{a}_{i,j} = \mathtt{b}_{i,j}$
`a != b`	none	none	`true` if `a` and `b` have different dimensions or $\exists\ i,j$ s.t. $\mathtt{a}_{i,j} \neq \mathtt{b}_{i,j}$
`a + b`	`a` and `b` have the same dimensions	none	a new matrix equal to the sum of `a` and `b`
`a - b`	`a` and `b` have the same dimensions	none	a new matrix equal to the difference of `a` and `b`
`a * b`	`a.cols() == b.rows()`	none	a new matrix equal to the product of `a` and `b`
`transpose(a)`	none	none	a new matrix equal to the transpose of `a`
`os << a`	none	prints `a` to output stream `os`	a reference to `os`

3.5.1 Output Operator and Serialisation

Since it is not feasible to output a dense representation for very large sparse matrices, you will instead output a serialised[2], sparse representation. An $m \times n$ matrix with k non-zero total entries and with $k_0, k_1, \ldots, k_t$ non-zero entries in rows $i_0, i_1, \ldots, i_t$, each at some matrix location (i, j) and with some value v, is serialised as:

(m, n, k)
$(i_0, j_0, v_0)␣(i_0, j_1, v_1)␣\cdots␣(i_0, j_{k_0-1}, v_{k_0-1})$
$(i_1, j_{k_0}, v_{k_0})␣(i_1, j_{k_0+1}, v_{k_0+1})␣\cdots␣(i_1, j_{k0+k1-1}, v_{k0+k1-1})$
$\ldots$
$(i_t, j_{k-k_t}, v_{k-k_t})␣(i_t, j_{k-k_t+1}, v_{k-k_t+1})␣\cdots␣(i_t, j_{k-1}, v_{k-1})$

[2] Serialisation usually yields a binary representation, but we just use plain text.

All values are parentheses-enclosed, integer triples, separated by single spaces as shown. Rows should appear in order and on a separate lines. Values are in column-index order. No leading or trailing whitespace should be output on each line. A newline should not be output for the last output line. It is C++ convention that the output operator performs minimal formatting and does not emit a trailing newline. Sample matrix **B** would be output as:

```
(8,7,15)
(0,0,1) (0,3,2) (0,6,3)
(1,0,4) (1,1,5)
(2,1,6) (2,2,7) (2,5,8)
(3,0,9) (3,3,10) (3,4,11) (3,5,12)
(4,1,13) (4,4,14)
(7,6,15)
```

NB: If you do not follow this format exactly you are likely to fail most automarking tests.

3.6 Destructor

Due to the use of dynamic memory allocation in your constructors, you must provide a destructor that deallocates the memory acquired by the constructors. You should ensure that your implementation does not leak memory. You will be penalised for an improperly implemented destructor.

3.7 Private Members

Remember you are required to store the matrix values in dynamically allocated arrays. Otherwise you may introduce whatever private members you feel are necessary for your implementation. This includes private member functions.

3.8 Exception Handling

Some of the matrix operations are required to detect runtime errors. The C++ exception handling infrastructure provides similar facilities to Java. Section §6.13 of your textbook introduces `try` blocks and exception handling. Further details can be found in Section §17.1.
A class, `MatrixError`, for exception handling is supplied:

```
#include <string>
#include <exception>

class MatrixError : public std::exception {
 public:
  MatrixError(const std::string& what_arg) : _what(what_arg) { }
  virtual const char* what() const throw() { return _what.c_str (); }
  virtual ~MatrixError() throw() { }
 private:
  std::string _what;
};
```

This class will be used for exceptions thrown by matrix operations. You can ignore this class and design your own, however the following requirements *must* be met:

- The name of your class must be `MatrixError`.
- Your class must provide the public operation `what()`, which, when called on an exception object, should print a meaningful error message for the exception thrown. (It doesn't have to be `virtual`.)
- The source code for this class must be placed in the files `SMatrix.h` and/or `SMatrix.cpp`.

There are two possible errors, size errors (e.g., multiplying a 2 × 3 matrix by a 5 × 3 matrix) and bound errors (e.g., `a(4,2)`, where `a` is 2 × 3. Some meaningful messages in both cases are:

```
Matrix size error: (2 x 3) * (5 x 3)
Matrix bound error: (4, 2) entry of 2 x 3 matrix
```

For marking purposes, the text before ':' must be exactly the same as indicated above.

The member declarations in the stub provided indicate which operations throw exceptions. Operators `+`, `-`, `*`, `+=`, `-=`, and `*=`, produce matrix size errors, the rest produce bound errors.

4 Getting Started

Carefully read this specification several times or until you are satisfied that you have not missed anything important. It is very easy to overlook small but critical details about what is required. Remember that there is no such thing as a "small error", incorrect results are incorrect results.

You are provided with a stub that contains all the files you will need to complete this deliverable, together with some test files. The stub is available on the subject account. Login to CSE and then type something like this:

```
% mkdir -p cs3171/del2
% cd cs3171/del2
% cp ~cs3171/soft/del2/del2.zip .
% unzip del2.zip
% ls
```

Spend some time familiarising yourself with the content of the stub files. The supplied code will require work before it compiles. You should first get your matrix class to compile by adding dummy implementations for each operation. You will then be able to develop your test suite as you write the code, testing each new addition.

You must organise your C++ code in two files: `SMatrix.h`, which contains the interface of your class for your clients, and `SMatrix.cpp`, which contains the implementation details of your class. *The `main` function should not be contained in either of these two files.*

5 Testing

Your code will be compiled together with a test file containing the `main` function, as follows:

```
% g++-4.3 -Wall -Werror -O2 SMatrix.cpp testx.cpp
```

If no binary is produced due to compilation errors you will receive a mark of 0 for that test. If a binary is produced, it will be run and its output compared to the expected output.

What output? Each test file will contain some code that will test various features of your implementation, e.g., create two matrices, add them together and print the result. If your result is correct then you've passed that test.

You are expected to test your code thoroughly. Please develop your own exhaustive test suite. A sample solution will not be provided for this deliverable, however the expected output should be evident. The proof of the pudding is in the eating!

Your favourite mathematics package (Maple, Mathematica, MATLAB) should support sparse matrices if you want to get really serious with your testing.

Since you are working with explicit memory management, part of your testing should consist of checking for memory leaks using a profiler (e.g., `valgrind` on the school machines, Shark on Mac OS X, etc.). You don't want your beautiful code to leak memory! What next? Become a multinational computer software giant!? A tool like `valgrind` can also detect other memory faults, like out-of-bounds access, uninitialised access, etc.

5.1 Performance

The astute and resourceful (=lazy) reader will quickly realise that, once `operator()` and `setVal` are correctly coded, most matrix operations can be implemented as they would be for dense matrices. This will produce a painfully slow, if correct, solution that is unlikely to score a total mark greater than pass.

The whole idea behind sparse matrix representation is peformance, which cannot be achieved if the operations do not take advantage of this sparse representation. For example, dense matrix multiplication runs in $O(n^3)$ (or slightly better at the cost of implementation complexity), so for two $10^9 \times 10^9$ matrices, that's $c \times 10^{27}$ operations. Ouch!

Please implement all operations as efficiently as possible, using the underlying sparse matrix representation. We will test your performance using fiendishly large test cases and your tutors will also check your implementation when marking the submissions.

Your mark may be capped at 50/100 if you do not follow the guidelines in this section!

6 Marking

This deliverable is worth **10%** of your final mark.

Your submission will be given a mark out of 100 with a 80/100 automarked component for output correctness and a 20/100 manually marked component for code style and quality.

As this is a third-year course we expect that your code will be well formatted, documented and structured. We also expect that you will use standard formatting and naming conventions. However, the style marks will be awarded for writing C++ code rather than C code.

We have found good use for some of the STL algorithms, e.g., `copy`, in our implementation. Remember that these algorithms work just as well on arrays as they do on the STL containers.

As always, a perfectly working submission can, and will, be penalised for failing to follow the implementation guidelines, in particular, failing to use dynamic memory allocation (total mark of 0). Note that you *must*, and should, use `new` and `delete` in favour of `malloc()` and `free()`.

7 Submission

DELIVERABLE 2 IS DUE: Sunday, 4 September 2011 at at 23:59:59.

Copy your code to your CSE account and make sure it compiles without any errors or warnings. Then run your test cases. If all is well then submit using the command (from within your `del2` directory):

```
% give cs3171 del2 SMatrix.cpp SMatrix.h
```

Note that you may only submit two specific files. Your entire submission *must* be contained in these two files.

If you submit and later decide to submit an even better version, go ahead and submit a second (or third, or seventh) time; we'll only mark the last one. Be sure to give yourself more than enough time before the deadline to submit.

Late submissions will be penalised unless you have legitimate reasons to convince the LIC otherwise. Any submission after the due date will attract a reduction of 10% per day to the maximum mark. A day is defined as a 24-hour day and includes weekends and holidays. Precisely, a submission x hours after the due date is considered to be $\lceil x/24 \rceil$ days late. *No submissions will be accepted more than five days after the due date.*

Plagiarism constitutes serious academic misconduct and will not be tolerated. CSE implements its own plagiarism addendum to the UNSW plagiarism policy. You can find it here:

`http://www.cse.unsw.edu.au/~chak/plagiarism/plagiarism-guide.html`.

Further details about lateness and plagiarism can be found in the Course Outline.