This is the design doc for the Matrix library created by Ian Feekes (ianfeekes@gmail.com)

Table of Contents:

1: Files Included

-Makefile: File containing the script for generating a matrixTest binary file that runs the unit tests
-Matrix.h: File containing all necessary definitions for the MATRIX namespace library. Including this file in your program allows you to access the matrix class and all of its functionality.
-Matrixutils_test.cpp: File with data structures holding literal values to be passed into matrices for unit testing of various facets of their functionality. The matrix library is by no means dependent on this file's existence: it only exists to prove the correctness of the solution, and to demonstrate to library users the functionality of the Matrix.h library.

If no arguments are specifed, it will silently run the unit tests, else if "talk" is specified after running eg

```
$ ./matrixTest talk
```

The program will output to users the results from each individual unit test
-UnitTest.h: File containing the specifications of the unit test structure that is used to make testing the matrices a more automated process, where data only needs to be written to Matrixutils_test.cpp and passed into these structures, which will then do all the heavy lifting and can output results.

Note that this automated testing could be more streamlined, however there are limitations due to template form of matrices causing them to fail being stored in conventional containers such as vectors, so much of the testing has code redundancies unfortunately. If the client were to want to continue development of automated unit testing and adding functionality to the matrix library, a good initial approach would be to create a container that holds the templatized matrix class, and to create a file parser in Matrixutils_test.cpp to remove large amounts of code while retaining functionality and usage.

2: Matrix Implementations

The matrix class is a template class with values of M columns, N rows, and a data type of T. Note that since matrices have purely mathematical intentions, the data type has only been tested for ints, doubles, and floats, and that errors will be thrown if a user attempts to initialize a matrix of string or char types.

The underlying representation of a matrix is a c-style array named ‘arr’ with [M*N] entries holding the matrix’s data type. This choice was made because matrices remain static in their number of rows and columns, and as such, it is best to hold memory contiguously (when many vectors may not be contiguous, especially those with dynamically-expanding size). A comparison of performance between matrices holding a single dimensional array and matrices holding one and two dimensional vectors can be found in the benchmarking to prove the efficiency of this underlying representation (see results). If users desire an array with dynamic row and column sizes, a one dimensional vector would be the a better data structure for the matrix’s underlying representation.

Matrices can be constructed with template parameters, and are designed to take in various data structures to initialize their values through overloading the constructor arguments and parameters. Matrices can be constructed with a single value (which may or may not be differently-typed) such that each element from the resulting matrix will be that value. Matrices can also be constructed with an argument of another matrix (with same template parameters) or constructed with an argument of a vector (which can be differently-typed).

Matrices have a built-in ‘==’ operator that allows them to check their data against constant values, vectors, other matrices, etc that may or may not hold different data types. This was not required to be implemented as part of this sample code, however it comes in useful for passing in expected values during unit testing.

Matrices also have a ‘=’ operator that allows them to change their data entries with vectors, single values, and other matrices, which is used in certain transposition and multiplication methods to instantiate new matrices and return them (doing the heavy lifting so the user doesn’t have to).

Matrices have a ‘*’ operator which will call the multiply method. This library follows the iterative approach as it shares time complexity with all other approaches, while avoiding the extra stack space, and high cache miss rates that are outcomes of other approaches such as the divide and conquer. This approach sames the same runtime as the multiplication operator for google’s matrix library and webgl functions. For an attempt to be clever, the algorithm originally held an ‘if’ statement to detect if an entry was 0 to avoid bothering to multiply and increment the sum for the given current value. However, after timing this and running it against a library that did not bother checking for 0’s, it was outperformed with a speedup of ~.92. The ‘*’ operator has been overloaded to perform static casting when users attempt to multiply different-typed matrices.

Matrices also have the option to perform scalar multiplication with different-typed values. The algorithm simply multiplies each matrix entry by the scalar.

Transposition is implemented with the transpose method and the createTranspose method. Transpose() requires a new matrix to be passed into its argument to fill with the new values, whereas createTranspose() creates a new matrix and calls the transpose function to

save users from having to initialize a new matrix manually each time they want to transpose. Transposition is done in O(M*N) time with careful attention paid to maximizing the cache hit rate.

3: Testing/Correctness of solution:

Transposition simply reverses the rows and columns of matrices, which is exactly how the algorithm performs through setting [i][j] to [j][i] (or as the user might imagine it: with a one dimensional array it's [i*N+j] to [j*M+i]). There are two sets of unit tests that call transposition on various matrices (different types, rows, columns, and values) with the intent to find edge cases, and they all return true for each run of the unit test. This shows that transposition is implemented correctly.

Matrix multiplication's algorithm is fairly straightforward with intents to reduce cache miss rate, and its following of the typical approach to the solution. There are about two sets unit tests that call multiplication on various matrices (different types, rows, columns, and values) that are compared with expected solutions to prove correctness of the solution. On all cases of unit testing, the multiplication results in expected values.

NOTE: as the == operator in c++ does not often work as desired for comparing floating point types, this library considers two floating point data types to be "equal" if they are different by a degree < EPSILON (which is .001).

Testing for correctness of expected values of initialization, '=', '*', transposition, and other various built-in matrix operators was all performed using the matrix test binary. With all the code in one directory, this can be done through invoking 'make' (which is just a script invocation of standard c++ compile command) to create the matrixTest executable file. This can then be executed with ./matrixTest to show the results of the unit tests.

4: Comparison and Analysis

1 Underlying Representation:

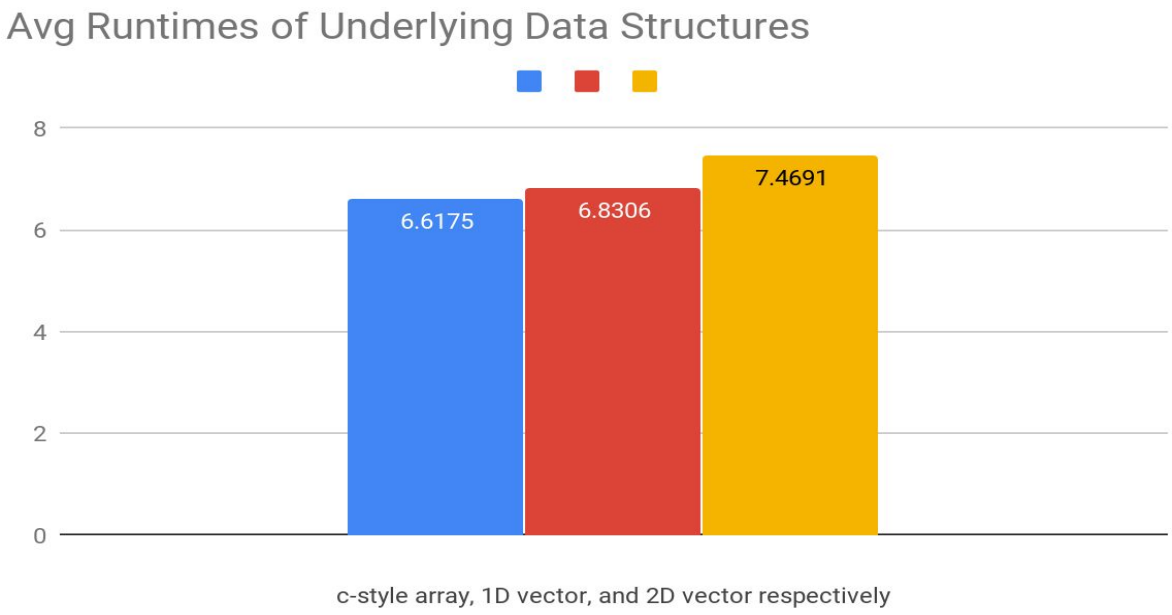
Given the functionality of the matrix library being solely multiplication and transposition, a single-dimensional c-style array works well for storing a matrix's values contiguously (which is not always the case with vectors) which is especially useful in speeding up multiplication and transposition with avoiding cache misses. Running this on a benchmarking program which initializes several very large floating-point matrices (which use enough data to attempt to increase I1 cache miss rate) with the time command allows us to quantify the outperformance compared to 1 and 2 dimensional vector representation as the following graph illustrates.

The matrixTest that tested these multiplication and translation functions looped 4000 times calculating the transpose and performing transposition under about a dozen <50,50,double> matrices in an attempt to maximize stressing an ALU workload.

The standard deviation for the c-style array representation was .0014 seconds, which is fairly consistent given that the test was timed for each data structure 30 times.

The standard deviation for 1-dimensional array representation is .0072 seconds, a fairly consistent performance as well, which makes sense given c++ vector library optimizations, and a good likelihood of contiguous block allocation seeing as matrices have their data initialized in their constructor.

The 2-dimensional vector representation, however, had a standard deviation of .071 seconds, which illustrates much higher performance inconsistencies compared to the previously mentioned data structures and their standard deviations. This may be attributed to higher cache miss rates due to the data more likely not being allocated contiguously within a 2D vector.

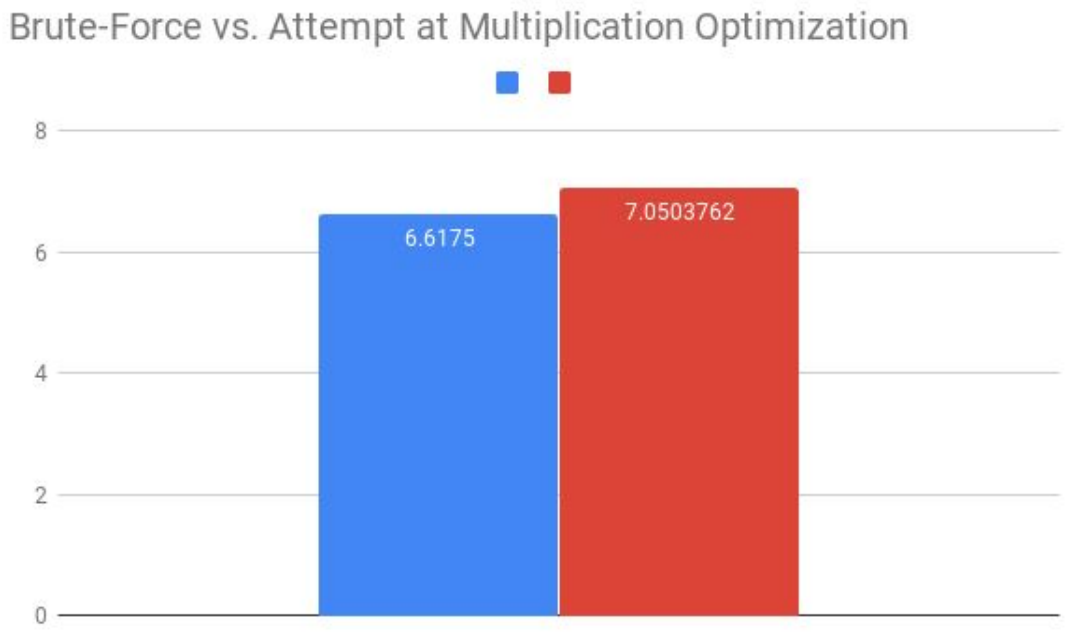


2: Multiplication Optimization:

As mentioned briefly under "Matrix Implementations," an attempt to speedup performance of the matrix multiplication algorithm originally was to check to see if the given element of either the multiplier or multiplicand matrix was zero, and if so, to simply move onto

the next element (incrementing the appropriate loop iterator). The idea behind this is to save multi-cycle ALU and branch instructions when necessary. However, when matrices running this corner-cutting technique were benchmarked against a standard brute-force multiplication approach without checking for zeros, the brute force outperformed the attempted optimization as the graph below illustrates.

This may be attributed to branch prediction failure. With this if statement leading to higher rates of failure in branch prediction, the pipeline will have to be emptied and reset, which can significantly reduce performance on an instruction-level of program execution. Also, it is possible the the c++ compiler my linux uses self-optimizes this multiplication algorithm in such a way that adding this extra step increases workload and complicates the optimization handled by the compiler.



(brute force solution) (attempt to cut corners with 0's)