UNIVERSITY OF WASHINGTON
Department of Applied Mathematics

AMATH 483/583 **Assigned:** 4/12/2017
Problem Set #3 **Due:** 4/18/2017

PYTHON GRADING SCRIPT

We will be using a new python grading script for all future homework. You will be able to download the scripts for each HW assignment from the canvas page. Make sure to download the tar file for AMATH 483 or for AMATH 483.

When you download and untar the file you will find a folder called student_grader_amath483 or student_grader_amath583. This folder will contain the following directories:

- *candidate*: a directory where you should place all your c++ code and makefile.
- *input_files*: a folder storing input files (do not modify)
- *key_output_files*: a folder storing reference output files (do not modify)
- *setting.json*: a file containing information for grading (do not modify)
- *grader.py*: the python grading script (do not modify)

To run the grader, copy your code and your makefile in the candidate directory, then cd to the directory student_grader_amath483 (or student_grader_amath583) and execute the command:

```
$ python grader.py
```

The output from the grader will look something like this

```
== RUNNING CANDIDATE ===============

1. Loading Settings.json
  PASS
2. Compilation
$ make all
  PASS
3. Copying Input Files to Program
copyied input1.txt
copyied input2.txt
4. Running Programs
$ ./main_1.exe : EXIT CODE CORRECT
$ ./main_2.exe : FAIL (returned exit code -9, expected 2)
5. Verifying Output Files Exist
file_1.txt: Exists
file_2.txt: Exists
file_3.txt: Exists
6. Comparing Output To Key
file_1.txt : PASS
file_2.txt : PASS
file_3.txt : PASS

== END TEST ========================

Estimated Score: 72/72
```

OUTPUTING SOLUTIONS

Instead of outputting results to the terminal using cout, scripts should now produce text files that contain answers. We will be providing you with all the functions for producing these output files. For now the functions we will provide are

1. writeVector(const Vector& x, const char * file_path)
2. writeNumber(T number, const char * file_path)   [where T can be int, float, double, etc]

We will also provide you with the following functions for reading input files

1. double readDouble(const char * file_path)
2. double readVector(const char * file_path)

For example, let us consider a main file that:

- loads two vectors whose paths are passed in through the argv array
- produces an output file contain the sum of the vectors

Ignoring argument verification, the program would look something like

```
int main (int argc, char *argv[])
{
     Vector x = readVector(argv[1])
     Vector y = readVector(argv[2])
     writeVector(x + y, "vec_sum.txt")
}
```

and would be executed using

```
./main.exe input_vector1.txt input_vector2.txt
```

COMPILING C++ CODE

The grader will run the command "make all" to compile your c++ code. This means that you will have to add an additional line to your makefile.

```
all: executable_1.exe executable_2.exe ... executable_n.exe
```

where you should replace executable_1.exe executable_2.exe ... executable_n.exe with a list of all the executable your must produce for the homework. An example makefile for producing main1.exe and main2.exe will look something this

```
all: main1.exe main2.exe

main1.exe: main1.cpp
     c++ main1.cpp -o main1.exe
main2.exe: main2.cpp
     c++ main2.cpp -o main2.exe
```

In this case, the command make all will produce both main1.exe and main2.exe

1 Preliminaries

1.1 Pre-Processing

Although not part of the C++ language per se, the pre-processor allows you to programmatically manipulate the text of the program itself. That is, you can make changes to the program text – in an automated fashion – before it is actually compiled. One example we touched on was in the use of `assert()` for defensive programming.

When `assert()` is enabled, it executes a check of the expression that is passed to it – that is, it executes a small amount of code. In performance critical applications, this can steal cycles as well as prevent certain compiler optimizations. For production (release) versions of a program, we need to be able to remove the assertions.

One way to remove the assertions, of course, would be to go to all of your files and either delete or comment out all of the calls to `assert()`. This is a anti-solution: during development we need to be able to switch between debug and release versions of the code. We need to be able to switch *all* of the calls to `assert()` on and off in one fell swoop each time. To support this, calls to `assert()` can be "turned off" by the pre-processor if the macro `NDEBUG` is defined.

The pre-processor works in the following basic way. It takes your program as input and provides program text as an output. How this output is produced is controlled by the pre-processors own macro language – which is also embedded in the program text. Some of the statements you have already seen in your programs, such as `#include` are pre-processor statements. In fact, all statements beginning with `#` are pre-processor language commands.

Beyond just filtering input and output, the pre-processor has some sophisticated features for text manipulation. However, we are just going to look at some basic capabilities for including or excluding specific parts of your program before being passed to the compiler. Other traditional uses for the pre-processor such as defining compile time literals have been superseded by **`const`** and **`constexpr`**.

The command you have already seen from the pre-processor is `#include`, which is used to pull the text of one file into another file. For example, if your file `Vector.cpp` includes the file `Vector.hpp`

```
#include "Vector.hpp"

int main() {

  return 0;
}
```

the text that is passed to the compiler is the concatenation of the two files. That is, what is compiled when you invoke the compiler on `Vector.cpp` is the complete text of `Vector.hpp` with `Vector.cpp`, with the text of `Vector.hpp` inserted at the location of the `#include ``Vector.hpp''`. If you would like to see all of the text that is passed to the compiler after pre-processing, you can add the option "`-E`" to your compilation command. Be careful with this though – if you have included anything from the standard library you will get an enormous amount of text back. The statements `#include <iostream>` are not special – there is a file named `iostream` that is part of the standard library and its text is pulled in with that pre-processor directive.

Now, as we mentioned above, the pre-processor can be programmed by the user. To do this, we need to be able to do expected programming tasks, such as defining and testing variables and branching based on the tests. Since the pre-processing text and the program text are combined – and since the pre-processor manipulates the program text – it is important to distinguish between pre-processor commands and variables, and the program text and variables. The convention is to use `ALL_CAPS` for any pre-processor macros or variables.

Variables in the pre-processor are defined with the following syntax:

```
#define MY_VARIABLE sometext
```

This creates the macro (pre-processor variable) with the name `MY_VARIABLE` in the pre-processor namespace. Two things happen when a pre-processor macro is defined. First, whenever the pre-processor encounters the text `MY_VARIABLE` in the program text, it substitutes the defined text for that variable. In other words, the following transformation occurs.

```
#include <iostream>
using namespace std;                              using namespace std;

#define MY_GREETING "Hello World"
#define OP *

int main() {                                      int main() {

  cout << MY_GREETING << endl;                      cout << "Hello World" << endl;
  cout << "7 x 6 = " << 7 OP 6 << endl;             cout << "7 x 6 = " << 7 * 6 << endl;

  return 0;                                         return 0;
}                                          ⇒      }
```

Note that the macros `MY_GREETING` and `OP` are replaced by *exactly* the text they are define to be. In this case, that even includes the quotation marks in `MY_GREETING`.

Branching in the pre-processor is controlled by a family of `#if` directives: `#if`, `#ifdef`, and `#ifndef`, which test the value of a compile-time expression, whether a macro is defined, or whether a macro is not defined, respectively. In response to evaluating an `#if` the pre-processor doesn't execute one branch of pre-processor code or another, rather it sends one stream of your program text or another to the compiler.

Defensive Programming One standard use of the branching capabiliites of pre-processor is to make sure that the text from any give header files is only placed once into the text stream to the compiler. This can easily happen when multiple headers include each other and/or include multiple headers from the standard library. In that case, even though the headers might be `#include`d, we only insert their text once. The following is standard technique. For any header file, the following pre-processor directives are used (assume the header file is `Matrix.hpp`):

```
#ifndef MATRIX_HPP   // if the macro MATRIX_HPP is not defined, include the following text
#define MATRIX_HPP  // First, define the macro


#endif  // The program text up to the matching #endif is what is included
```

You should make a habit of always protecting your header files in this way.

As you might expect, there is an `#else` to go along with `#if`.

```
#include <iostream>

int main() {

#ifdef BAD_DAY
  std::cout << "Today is a bad day'' << std::endl;
#else
  std::cout << "Today is a good day" << std::endl;
#endif

  return 0;
}
```

In this example, the compiler will get the first branch of text if the macro `BAD_DAY` is defined, otherwise it will get the second branch. **NB:** With `#ifdef`, it does not matter what the value of the macro is. The test is only whether the macro exists or not. It is perfectly acceptable to `#define` a macro with no value.

In fact, when we want to disable `assert()`, we just need to `#define` the macro `NDEBUG`, we don't need to give it any particular value. But, since the pre-processor just processes your program text and sends it to the compiler, the `NDEBUG` macro *must* be defined before the `#include <cassert>` statement.

But this raises almost the same scalability issue we mentioned before. If we want to globally remove assert, we need to have the `NDEBUG` macro defined when processing our program files. One way to do this would be to edit each of the files and insert (or remove) `#define NDEBUG` in every one. This is impractical for all but the smallest of programs (and maybe even not then).

There is an essential feature of the C++ compiler that solves this problem, namely the `-D` option. This option passes a macro (with or without a defined value) to the prep-processor. In particular, you can pass `NDEBUG` to your programs this way (without ever having to change the program):

```
c++ -DNDEBUG main.cpp -o main.exe
```

You can give the macro a value by using `=`

```
c++ -DNDEBUG=1 main.cpp -o main.exe
```

but for `NDEBUG` that the definition exists, not any particular value, is what turns on or turns off `assert()`.

Automation In keeping with the course philosophy of "automate anything repetitive", the place to take advantage of incorporating (or not) `NDEBUG` definitions is in your `Makefile`. However, a problem with a familiar feature appears. Namely, if we want to globally turn on or turn off `assert()`, we have to make a sweep through the `Makefile` and add it or remove it from every production rule. If we truly want to be able to enable or disable `assert()` with the flip of a switch (as it were), we need to just be able to change it in *one place* but have the effect be global.

To programmatically control the automation that is introduced by `make`, the `make` program also has its own macro language. Fully using those capabilities can quickly turn into "Deep Magic" and we want to avoid that, but there are some basic features that enable the "edit once - change everywhere" behavior we are looking for.

In particular, `make` allows you to define macros that are expanded within the body of the `Makefile`. Two very common macros that you might use are to define which compiler you want to use, and to define which flags you want to pass to the compiler. For example, to make the programs `dot5893` and `vectorNorm` we might have the following intermediate rules in the `Makefile`:

```
dot583.o: dot583.cpp amath583.hpp
        c++ -Wall -g -std=c++11 -c dot583.cpp -o dot583.o

amagth583.o: amath583.cpp amath583.hpp
        c++ -Wall -g -std=c++11 -c amath583.cpp -o amath583.o

vectorNorm.o: vectorNorm.cpp amath583.hpp
        c++ -Wall -g -std=c++11 -c vectorNorm.cpp -o vectorNorm.o
```

And one can easily imagine having many more production rules for any number of files. Now, suppose we wanted to introduce `NDEBUG`, or change the compiler we are using, or switch between release levels of C++? Manually, we would have to edit every line, doing it completely consistently, and not introducing any errors. Or, we can use a macro and do it once. Using a `Makefile` macro is straightforward. You define it with `=` and use it with `\$`. If we used the standard approach in the above `Makefile`, it would look like this:

```
CXX      = c++
CXXFLAGS = -Wall -g -std=c++11

dot583.o: dot583.cpp amath583.hpp
        $(CXX) $(CXXFLAGS) -c dot583.cpp -o dot583.o

amagth583.o: amath583.cpp amath583.hpp
        $(CXX) $(CXXFLAGS) -c amath583.cpp -o amath583.o

vectorNorm.o: vectorNorm.cpp amath583.hpp
        $(CXX) $(CXXFLAGS) -c vectorNorm.cpp -o vectorNorm.o
```

Now, if we wanted to add `NDEBUG` to disable `assert()`, we simply change the definition of `CXXFLAGS`:

```
CXXFLAGS = -Wall -g -std=c++11 -DNDEBUG
```

We can do the same with switching from debug mode to release mode

```
CXXFLAGS = -Wall -O3 -std=c++11 -DNDEBUG
```

(Note the use of `-O3` rather than `-g`.) There are a number of conventions used in naming and use of `Makefile` macros – and a number are pre-defined for you. Those wanting to further leverage the automation capabilities of `make` are encouraged to consult the on-line references given on the course web site.

Just for Ninjas For the truly lazy (in a good way), you will quickly notice that there is *still* alot of repetition in the `Makefile`. All of the production rules have the same pattern

```
        $(CXX) $(CXXFLAGS) -c <something>.cpp -o <something>.o
```

Compile "something.cpp" to create "something.o". In large programs with large `Makefile`s, keeping these all consistent could benefit from automation. To automate pattern-based production rules, `make` has some "magic" macros that pattern match for you.

```
CXX      = c++
CXXFLAGS = -Wall -g -std=c++11

dot583.o: amath583.hpp
dot583.o: dot583.cpp
        $(CXX) -c $(CXXFLAGS) $< -o $@

amath583.o: amath583.hpp
amath583.o: amath583.cpp
        $(CXX) -c $(CXXFLAGS) $< -o $@

vectorNorm.o: amath583.hpp
vectorNorm.o: vectorNorm.cpp
        $(CXX) -c $(CXXFLAGS) $< -o $@
```

A few things to note here before we get to our final `Makefile`. First, dependencies don't have to all be on one line. In the above, we express the dependencies for the object files in multiple lines, one dependency per line. The lines that then have production rules are executed when any of the dependencies are not met.

```
        $(CXX) -c $(CXXFLAGS) $< -o $@
```

In this production rule, the macro `$<` means the file that is the dependency – the .cpp file – and `$@` means the target.

There is one last thing to clean up here. We *still* have a repetetive pattern – all we did was substitute the magic macros in. But every production rule still has:

```
something.o : something.cpp
        $(CXX) -c $(CXXFLAGS) $< -o $@
```

There is one more pattern matching mechanism that `make` can use – implicit rules. We express an implicit rule like this:

```
%.o : %.cpp
        $(CXX) -c $(CXXFLAGS) $< -o $@
```

This basically the rule with "something" above – but we only need to write this rule *once* and it covers all cases where something.o depends on something.cpp. We still have the other dependencies (on headers) to account for – but this can also be automated – google for "makedepend".

At any rate, the Makefile we started with now looks like this:

```
CXX      = c++
CXXFLAGS = -Wall -g -std=c++11

%.o : %.cpp
	$(CXX) -c $(CXXFLAGS) $< -o $@

dot583.o: amath583.hpp
amath583.o: amath583.hpp
vectorNorm.o: amath583.hpp
```

This will handle *all* cases where something.o depends on something.cpp.

NB: Unless otherwise instructed, you do *not* have to use any of the advanced features of `make` in your assignments as long as your programs correctly build for the test script. However, keep in mind that these mechanisms were developed to save time and decrease mistakes while programming.

2 Warm Up

2.1 make

To verify, and gain some familiarity with, the operation of `Makefile` and pre-processor macros, work through the following examples.

Create the following program:

```cpp
#include <iostream>
#include <cassert>

int main() {

  assert(1 == 0);

  std::cout << "Hello World" << std::endl;

  return 0;
}
```

Notice that the assertion `(1 == 0)` will always be false and so the assertion will fail. On failure, the program will abort, and will abort before it prints the hello message. Try compiling and running that program with no particular compiler options and verify that it aborts before printing the message.

Next compile the program with the `-DNDEBUG` option.

```
$ c++ -DNDEBUG fail.cpp
```

What happens when you run the resulting `a.out`?

Finally, create a test `Makefile` that looks like the following:

```
CXX      = c++
CXXFLAGS = -Wall -g -std=c++11

fail:  fail.cpp
	$(CXX) $(CXXFLAGS) -o fail

clean:
	/bin/rm -f fail fail.o a.out
```

Preliminary question – when you type `make`, what gets created? What happens when you run that program?

Finally, modify the above `Makefile` so that it compiles a program that ignores the `assert()`. Do that without changing the production rule (the compilation line after the dependency rule for `fail`).

2.2 Timing

As discussed in lecture – if we are going to achieve "high performance computing" we need to be able to measure performance. Performance is the ratio of how much work is done in how much time – and so we need to measure (or calculate) both to quantitatively characterize performance.

To measure time, in lecture we also introduced a `Timer` class (also available as `Timer.hpp` in the example code directory).

```
class Timer {
private:
  typedef std::chrono::time_point<std::chrono::system_clock> time_t;

public:
  Timer() : startTime(), stopTime() {}

  time_t start()   { return (startTime = std::chrono::system_clock::now()); }
  time_t stop()    { return (stopTime  = std::chrono::system_clock::now()); }
  double elapsed() { return std::chrono::duration_cast<std::chrono::milliseconds>(stopTime-startTi

private:
  time_t startTime, stopTime;
};
```

To use this timer, you just need to `#include "Timer.hpp"`. To start timing invoke the `start()` member, to stop the timer invoke the `stop()` member. The elapsed time between the start and stop is reported with `elapsed()`.

To practice using the timer class, write and compile the following program.

```
#include <iostream>
using namespace std;

#include "Timer.hpp"

int main() {
  long loops = 1024L*1024L*1024L;

  Timer T;
  T.start();
  for (long i = 0; i < loops; ++i)
    ;
  T.stop();

  cout << loops << " loops took " << T.elapsed() << " milliseconds" << endl;

  return 0;
}
```

First, to get a baseline, compile it with no optimization at all. On my laptop, the 1G loops above took about 2 seconds. If your computer takes too long or too short, you can adjust the loop value (multiply it by 2 for example, or change one of the 1024 values into 512). What value does your computer give when timing this loop? How many milliseconds per loop? Note that the empty statement ";" in the loop body just means "do nothing."

Second, let's look at how much optimizing this program will help. Compile the same program as before, but this time use the `-O3` option. How much did your program speed up? If it executes too quickly, again, try increasing the loop count. How much time per iteration is spent now? Does this make sense?

If you are unsure about the answer you are getting here, start a discussion on Piazza. Try to have this discussion sooner rather than later, as you will need some of the information gained for later in this assignment.

2.3 Abstraction Penalty and Efficiency

One question that arises as we continue to optimize, e.g., matrix multiply is: how much performance is available? The performance gains we saw in class were impressive, but are we doing well in an absolute sense? To flip the question around, and perhaps make it more specific: We are using a fairly deep set of abstractions to give ourselves notational convenience. That is rather than computing linear offsets from a pointer directly to access memory, we are invoking a member function of a class (recall `operator()()` is just a function. Then from that function we are invoking another function in the `vector<double>` class – `operator[]()`. And there may even be more levels of indirection underneath that function. Calling a function involves a number of operations, saving return addresses on the stack, saving parameters on the stack, jumping to a new program location – and then unwinding all of that when the function call has completed. When we were analyzing matrix-matrix product in lecture, we were assuming that the inner loop just involved a small number of memory accesses and floating point operations. We didn't consider the cost we might pay for having all of those function calls – calls we could be making at every iteration of the multiply function. If we were making those – or doing anything extraneous – we would also be measuring those when timing the multiply function. And, obviously, we would be giving up performance. The performance loss due to the use of programing abstractions is called the *abstraction penalty*.

One can measure the difference between achieved performance vs maximum possible performance as a ratio – as *efficiency*. Efficiency is simply

$$\frac{\text{Achieved performance}}{\text{Maximum performance}}$$

Measuring maximum performance. Let's write a short program to measure maximum performance – or at least measure performance without abstractions in the way.

```
size_t N = 1024L;
double a = 3.14, b = 3.14159, c = 0.0;
Timer T;
T.start();
for (size_t i = 0; i < N*N*N; ++i) {
  c += a * b;
}
T.stop();
```

To save space, I am just including the timing portion of the program. In this loop we are multiplying two doubles and adding to doubles. And we are doing this N^3 times – exactly what we would do in a matrix-matrix multiply.

Time this loop with and without optimization. What happens? How can this be fixed? Again, this is a question I would like the class to discuss as a whole and arrive at a solution.

2.4 The Matrix Code

For this assignment you will be provided with a number of source code files that you should put into a subdirectory `ps3`. The code for the `Matrix` class will be in `Matrix.cpp` and `Matrix.hpp`. The files wil contain most of the examples that we showed in lecture. There will also be a driver program `bench.cpp` that when built (see the Makefile) will create an executable that will run performance tests for a variety of matrix sizes and algorithms (as specified on the command line). You should be able to deduce how it functions based on inspection of the code as well as simply building it and running it. Try some different sizes and different algorithms. In general you should keep the `-O3` flag turned on or you will be waiting a long time for the programs to complete.

For creating performance plots as shown in lecture, there is also a small python script `plot.py`. This program takes a list of names. For each name it looks for a file called name.txt, which should contain the output from running the `bench` program. It will then create a plot in a pdf file that contains performance plots from the data in each file.

Note that the `bench.cpp` program sends its output to `std::cout`. To capture this into a file for use by `plot.py`, you can use the I/O redirection capability in the `bash` shell:

```
$ ./bench 2x2 > 2x2.txt
```

This will send the output of the run with the 2×2 kernel to the file `2x2.txt`. You can then generate a pdf file with plot.py

```
$ python plot.py 2x2
```

Note that no matter what input you give the program it will always create an output `matplot.pdf`. Note also that you can put multiple lines on the same plot

```
$ python plot.py 2x2 4x4 4x2
```

Disclaimer: The `plot.py` script has not been fully tested or debugged. Contributions and improvements are welcomed. I am especially trying to find a solution to make plots with a 9×4 aspect ratio rather than square.

Note that in the `Matrix.cpp` file we have provided a function for filling a matrix with random numbers as well as a function for zeroing out a matrix.

NB: I recommend that you take a quick tour through the code and familiarize yourself with some of the contents.

3 Exercises

3.1 Floats and Doubles

In the first problems set, several students noticed that if they used a **`float`** instead of **`double`** that the newton iteration would only converge to about the first 8 digits or so. Most microprocessors today support two kinds of floating point numbers: single precision (**`float`**) and double precision (**`double`**). The IEEE 754 floating point standard specifies how many bits are used for representing **`float`**s and **`double`**s, how **`float`**s and **`double`**s are represented, what the semantics are for various operations, and what to do when exceptions occur (divide by zero, overflow, denormalization, etc.). This standard, and the hardware that supports it (notably the original 8087 chip) is one of the most significant achievements in numerical computing. (William Kahan won the Turing award for his work on floating point numbers.)

Since **`double`** has so much better precision than **`float`**, why would one ever use a **`float`**? Floating point computation represents quite a bit of hardware real estate in a CPU. When CPUs were much more expensive than they are today (and might have been mainframes rather than microprocessors), single precision required less efforts (and cost to implement). Memory was similarly limited and single precision numbers used less memory. Finally, the computation process itself was not as blazingly fast as it is today – single precision had higher performance. As CPUs have become immensely more powerful (and economical), as memory became incredibly inexpensive, and as floating point techniques have advanced, some of these tradition concerns aren't really of concern anymore.

In this exercise we are going to investigate some operations with **`float`** and **`double`** and see what may or may not still be true (and hypothesize why).

Write a program that has two (very similar) sections: one for single precision and one for double precision. The functionality of each section is the same, only the datatype is different. In each section, you are to time two things: the time it takes to allocate and initialize three arrays and the time it takes to multiply two of them and set the third to the result.

The section for double might look like this, for instance (with the versions for float being very similar:

```
{
  Timer t;    t.start();
  double_vector x(dim, 3.14);
  double_vector y(dim, 27.0);
  double_vector z(dim, 0.0);
  t.stop();
  std::cout << "Construction time: " << t.elapsed() << std::endl;

  t.start();
  for (size_t i = 0; i < dim; ++i)
    z[i] = x[i] * y[i];
  t.stop();
  std::cout << "Multiplication time: " << t.elapsed() << std::endl;
}
```

For this exercise, generate four sets of numbers. Single precision with optimization off, double precision with optimization off, single precision with optimization on and double precision with optimization on.

Deliverable. In the text file ps3.txt include the following:

1. What is the difference between single and double precision for construction with no optimization? Explain.
2. What is the difference between single and double precision for multiplication with no optimization? Explain.
3. What is the difference between single and double precision for construction with optimization? Explain.
4. What is the difference between single and double precision for multiplication with optimization? Explain.
5. What is the difference between double precision multiplication with and without optimization? Explan.

Your explanations should refer to the simple machine model for single core CPUs with hierarchical memory that we developed in lecture 5.

3.2 Efficiency

For thie problem we want to investigate how to properly measure a loop with just a multiply and an add in the inner loop. In particular, we want to see if that cant tell us the maximum FLOPS count you can get on one core. We would like to relate that to the clock rate of the CPU on your computer and estimate how many floating point operations being done on each clock cycle. See the paragraph marked ”Measuring maximum performance” above.

Once you have a meaningful answer for the maximum floating point rate of your computer, we can use this result as the denominator in computing efficiency (fraction of peak).

Deliverable. In the text file ps3.txt include the following:

1. Description of what changes you made to the above snippet of timing code, if any
2. Explanation of why you made those changes
3. Clock rate of your computer
4. Max achieved floating point rate of your timed code (with and without optimization)

Note that since we are interested in maximum efficiency, we are interested in the rate possible with fully optimized code. **NB:** Give some thought to the numbers that you generate and make sure they make sense to you. Use double precision for your experiments.

3.3 operator+=

When we defined operator+ in lecture, we saw that it allocates and returns a Vector.

```
  Vector operator+(const Vector& x, const Vector& y) {
  Vector z(x.num_rows());
  for (size_t i = 0; i < x.num_rows(); ++i) {
    z(i) = x(i) + y(i);
  }
  return z;
}
```

One common operation in numerical linear algebra has the form $y = y + \alpha x$ where x and y are vectors and α is a scalar. Let's consider a simpler case, where there is no α, so we just have $y = y + x$. If we write that operation using our Vector class and operator+ we would write

```
Vector x(1024), y(1024);
// fill vectors, compute, etc
y = y + x;
```

Now, if we expand the operators we see that we are (essentially) doing

```
Vector z(x.num_rows());
for (size_t i = 0; i < z.num_rows(); ++i) { // operator+ expanded
  z[i] = x[i] + y[i];
}
for (size_t i = 0; i < z.num_rows(); ++i) {  // operator= expanded
  y[i] = z[i]
}
```

But this isn't what we want at all. If you were just going to write a "**hand written loop**" for $y = y + x$, you would just write

```
for (size_t i = 0; i < z.num_rows(); ++i)
  y[i] = y[i] + x[i];
```

The case with the operator+ and subsequent assignment requires allocating a new Vector as well as looping twice over x and y.

If you look at the instruction set for modern microprocessor, the addition operation actually is typically of the form $y = y + x$, as a single instruction. To help compilers pick that instruction rather than creating a temporary (as we saw above in the Vector example), C (and C++) have special assignment operators. In particular,instead of writing `y = y + x;` one can simply write `y += x;`. Using this operator helped simpler compilers generate the right instructions for what the programmer intended – and avoided making unnecessary copies. Modern compilers are more powerful and can usually deduce that `y = y + x;` should compile to the equivalent of `y += x;`.

That's fine for built-in types, the compiler knows about those. But what about user defined types? For the Vector example above, even a very powerful compiler would not be able to turn the expression $y = y + x$ into the second, more efficient, loop. Here is where the rich set of assignment operators can come to the rescue. That is, we can define operator+= for Vector and obtain the more efficient loop construct.

Deliverable. Implement operator+= for Vector. Put the declaration in amath583.hpp and the implementation in amth583.cpp. It should have a prototype as follows:

```
Vector& operator+=(Vector& y, const Vector& x);
```

It should be implemented in the form of the efficient loop above. Write a test program `vector_add.exe` to verify that using `operator+=` is as efficient as the hand written loop.

Your program should take a first command line argument specifiying the size of the `Vector` you will be taking and a second that specifies how many times the loop will be run inside of start and stop of a timer. The program should produce two output files:

1. *timed_add_vec_ppe.txt*: contains a double equal to the ratio of time taken to run the overloaded + operator (i.e. y = y + x) over the time taken to run vector addition using overloaded += operator (i.e. y += x). You should time 1000 trials of vector addition for both these cases. Use the function *writeNumber* to produce the output file.
2. *timed_add_vec_hpe.txt*: contains a double equal to the ratio of time taken to run the hand-written loop (i.e. y(i) = y(i) + x(i)) over the time taken to run vector addition using overloaded += operator (i.e. y += y). You should time 1000 trials of vector addition for both these cases. Use the function *writeNumber* to produce the output file.

3.4 operator* for scalar time vector (583 ONLY)

For this exercise, write an `operator*` function that applies a scalar to a `Vetor` and returns a new `Vector` that is the result of that product.

```
Vector operator*(const double a, const Vector& x);
```

Declare the function in amath583.hpp and implement it in amath583.cpp.

Deliverable. As above, but with `a * x` in place of `x` (and a * x(i) in place of x(i) in the hand written loops). Does the `operator+=` match the hand-written variant? If not, explain. Extra credit: can you think of a way to circumvent the problem (if there was a problem)?

4 Turning in The Exercises

Create a tarball ps3.tgz with all the files necessary to run your code, the your written exercise responses ex2.t, and a text file ref2.txt that includes a list of references (electronic, written, human) if any were used for this assignment.

As with the previous assignment, before you upload the ps3.tgz file, it is **very important** that you have confidence in your code passing the automated grading scripts.

IMPORTANT:

Before submitting homework, check that the command "make all" generates the executables:

add_vec.exe, timed_add_vec.exe, st_vec.exe.

The proper usage for each of these commands is as follows:

1. add_vec.exe [vectorFilename] [vectorFilename]
2. timed_add_vec.exe [vector_size - int] [num_trials - int]
3. st_vec.exe [doubleFilename] [vectorFilename]

The python grader will automatically produce all input files so you do not write these by hand! The grader will give you an estimated grade by running each of the following command(s)

1. `$ add_vec.exe input_vector_1.txt input_vector_2.txt`

   which should generate the following output file:

   (a) *add_vec.txt*: contains a Vector object corresponding to the sum of the Vector objects saved in vector1.txt and vector2.txt. You should use the function *readVector* to read in the vectors and the function *writeVector* to produce the output file.

2. `$ timed_add_vec.exe 1024 1000`

   which should generate the following output file:

   (a) *timed_add_vec_ppe.txt*: contains a double equal to the ratio of time taken to run the overloaded + operator (i.e. y = y + x) over the time taken to run vector addition using overloaded += operator (i.e. y += y). For these specific arguments The vector should be of size 1024 and 1000 trials of vector addition should have been performed. Use the function *writeNumber* to produce the output file.

   (b) *timed_add_vec_hpe.txt*: contains a double equal to the ratio of time taken to run the hand-written loop (i.e. y(i) = y(i) + x(i)) over the time taken to run vector addition using overloaded += operator (i.e. y += y). For these specific arguments The vector should be of size 1024 and 1000 trials of vector addition should have been performed. Use the function *writeNumber* to produce the output file.

3. `$ st_vec.exe input_double_1.txt input_vector_1.txt`

   which should generate the following output file:

   (a) *st_vec.txt*: contains a Vector object corresponding to the double saved in input_double_1.txt multiplied by the Vector saved in input_vector_1.txt. Use the functions *readDouble* and *readVector* to load the double and the vector and the function *writeVector* to produce the output file.

NOTE: st_vec.exe is only required for AMATH 585 students

5 Learning Outcomes

At the conclusion of week 3 students will be able to

1. Describe the interface and implementation of the `Matrix` class, including the `operator()()` member function.
2. Implement basic matrix-vector product and matrix-matrix product functions using the external interface of the `Matrix` class.
3. Include or exclude code in a source code file using statements from the `#if` family.
4. Explain the high-level functionality of the `-O3` compiler flag.
5. Derive the mapping from two `Matrix` indices to one `std::vector<double>` index.
6. Write a simple C++ class.
7. Correctly use initializer in the constructor for a C++ class.
8. Write a simple test harness for measuring performance of matrix-vector and matrix-matrix multiply routines.
9. Explain the hardware mechanisms that are leveraged by the hoisting, tiling, blocking, and copy-transpose matrix-matrix optimizations.
10. Explain the difference between column-major and row-major ordering.
11. Derive the (basic) computational complexity of matrix-matrix product and matrix-vector product.