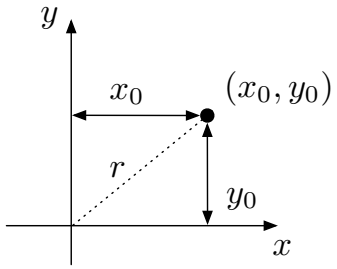
UNIVERSITY OF WASHINGTON
Department of Applied Mathematics

AMATH 483/583 **Assigned:** 4/4/2018
Problem Set #2 **Due:** 4/11/2018

1 Preliminaries

In Lectures 3 and 4 we introduced vector spaces – sets of objects that obeyed certain properties with respect to addition and with respect to scalar multiplication. What that definition did not include was any way to measure the distance between two vectors (or, equivalently to measuring the distance between a vector and the origin, the magnitude of a vector). In plane geometry, points in the plane are represented as pairs of numbers, and it is fairly easy to verify the vector space properties with pairs and see that pairs of numbers can represent a vector space.

The figure below illustrates how we customarily think about measuring distances in the plane.



That is, for a vector represented as the pair (x_0, y_0), the length of that vector, its distance from the origin, is $r = \sqrt{x_0^2 + y_0^2}$.

We can generalize that notion to N-tuples of numbers (N-dimensional vectors) in the following way. Let the vector $\boldsymbol{x}$ be the N-tuple of real numbers

$$\boldsymbol{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix}.$$

We use the shorthand for saying that $\boldsymbol{x}$ is an N-tuple of real numbers with the notation $\boldsymbol{x} \in \mathbb{R}^N$. Then, the distance of the vector $\boldsymbol{x}$ to the origin is

$$||\boldsymbol{x}||_2 = \sqrt{\sum_{i=0}^{N-1} x_i^2} = \left(\sum_{i=0}^{N-1} x_i^2 \right)^{\frac{1}{2}}.$$

Distance is expressed as $||\cdot||$ is called a "norm" and the 2-norm above is also called the Euclidean norm (in analogy to the plane geometry interpretation of distance). There are two other common norms in numerical linear algebra that can also provide useful notions of distance, respectively the 1-norm and the infinity (or max) norm:

$$||\boldsymbol{x}||_1 = \sum_{i=0}^{N-1} |x_i| \qquad \text{and} \qquad ||\boldsymbol{x}||_\infty = \max_i |x_i|.$$

A vector space with a norm is called a normed vector space; if the vector space is complete in the norm, it is called a Banach space. As with the definition we had in lecture about vector spaces, any function $f : V \to \mathbb{R}$ can be a norm on a vector space V, provided it satisfies certain properties:

1. $f(\boldsymbol{x}) \geq 0$ for all $\boldsymbol{x} \in V$
2. $f(\boldsymbol{x} + \boldsymbol{y}) \leq f(\boldsymbol{x}) + f(\boldsymbol{y})$ for all $\boldsymbol{x}, \boldsymbol{y} \in V$

3. $f(\lambda \boldsymbol{x}) = |\lambda| f(\boldsymbol{x})$ for all $\lambda \in \mathbb{C}$ and $\boldsymbol{x} \in V$

4. $f(\boldsymbol{x}) = 0$ if and only if $\boldsymbol{x} = 0$

The interested reader can verify that the 1-norm, 2-norm, and infinity norm defined above satisfy these properties.

2 Warm Up

Starting with this assignment, we will be building up a small library of high-performance linear algebra functionality – both performance and functionality will be built up in successive assignments. We will be using files as data input and output mechanisms for the computational capabilities we will be developing. As is customary with large software libraries, we will factor the software it into multiple files. Regardless of the size of a software project, we will want to start automating the compilation process. (As the course progresses we will also be introducing more advanced features of make.)

Create a new subdirectory named "ps2" to hold your files for this assignment. Using the example from the first problem set, create a source code file "hello.cpp" in this subdirectory. Create a file "Makefile" with the following contents:

```
hello: hello.cpp
	c++ hello.cpp -o hello
```

Note that you *must* use a tab as the leading whitespace on the second line. Use of spaces (or no space) will result in an error along the lines of `missing separator`.

Now, with this Makefile and your hello world source code file in your ps2 subdirectory, issue the following command:

```
$ make -n hello
```

The `-n` option tells make to print out what it would do, but not actually do it. In this case it should print out the compilation command on the second line of the Makefile. Next, invoke make without giving it a target name:

```
$ make -n
```

What does it print out? In general (look this up in your favorite resource) what is the default behavior of make if you don't pass in a target name on the command line? Finally, proceed with the actual make:

```
$ make hello
```

Verify that the compilation took place by running the program.

One feature of the rule consequent commands that we did not discuss in lecture is that you can have a sequence of commands that run (in order) if a rule is triggered. Execution of the sequence of commands will halt if a command fails – an example of a program (make) using the return value from another program (one of the rule consequents). Also note that further rules will not be triggered if a rule in the make chain fails.

Create the following program in your ps2 directory:

```
#include <iostream>
using namespace std;
int main() {
  cout << "Goodbye World" << endl;
  return 1;
}
```

Note that this program returns a value of 1 rather than 0 to the calling program. Augment your Makefile as follows:

```
hello: hello.cpp
        c++ hello.cpp -o hello

goodbye: goodbye.cpp hello
        c++ goodbye.cpp -o goodbye

test1: hello goodbye
        echo "say hello then goodbye"
        ./hello
        ./goodbye
        echo "done"

test2: hello goodbye
        echo "say goodbye then hello"
        ./goodbye
        ./hello
        echo "done"
```

What happens when you "make test1"? What happens when you "make test2"? Again, note that you can have an arbitrary number of commands in the sequence of commands in the consequent. See also the instructor provided Makefile.inst for a much more involved sequence of commands for each rule.

2.1 Defensive Programming and Assertions

Maurice Wilkes was one of the founders of modern computing and, in some sense, of debugging. One of his most poignant quotes is:

> It was on one of my journeys between the EDSAC room and the punching equipment that "hesitating at the angles of stairs" the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs.

Over the years, *defensive programming* has evolved as a set of techniques that can be used to help you find your own errors. One fundamental part of the process of defensive programming is to develop a program two support two different modes of compilation and execution: debug mode and release mode. In debug mode, a program is first of all, compiled to enable use of a symbolic debugger (using the -g flag to the compiler). In addition, the program itself includes self-checks, or *assertions*, (inserted by the programmer) to insure that necessary program invariants hold.

Assertions Note that these self-checks (assertions) are not the same as error-checking. Most importantly, assertions are removed from the release mode of the program. In the first problem set, we tested for some errors that might occur because a user incorrectly creates input for the main() or readVector() functions. Handling user errors is part of normal program logic for a correctly functioning program. However, there are other kinds of errors due to flaws in the program logic (aka "bugs").[1] Correct program logic depends on invariants holding during the course of execution. During development and debugging it can be useful to check for invariants and to terminate the program at the point where an invariant is violated. C and C++ provide a checking facility for asserting such invariants in the ⟨assert⟩ header. There is a concise description of the principles of using `assert` here: `http://bit.ly/2o9THxq` . Exactly how and where to use `assert` when you are programming will largely be up to you, but you should add it to your arsenal of tools and techniques for programming in this course (and beyond) so that the remainder of your life can be spent doing other things than finding errors in your own programs.

An assertion statement will print a message and cause a program to halt if the assertion fails, that is, if the expression passed to it evaluates to false or, equivalently, to zero (false and zero are essentially the same value in C/C++). As mentioned above, `assert` statements are removed from your program for its release

[1] One kind of program flaw, of course, is not catching a user error that results in invalid data and subsequent undefined behavior in the program.

mode. This removal is done functionally rather than physically – you don't actually go through the code and remove the `assert` statements. Rather, they are turned into empty statements by the pre-processor if the macro `NDEBUG` exists in the environment prior to inclusion of the header file `<cassert>`. Try the following three programs:

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

int main() {
  assert(1 == 1 && "This is true");        // will not be triggered
  assert(1 == 0 && "This is never true"); // will be triggered
  cout << "Hello World" << endl;
  return 0;
}
```

```
#define NDEBUG

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

int main() {
  assert(1 == 0 && "This is never true"); // will be triggered
  cout << "Hello World" << endl;
  return 0;
}
```

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

#define NDEBUG


int main() {
  assert(1 == 0 && "This is never true"); // will be triggered
  cout << "Hello World" << endl;
  return 0;
}
```

Which version prints "Hello World"? The technique of using the logical "and" operation (&&) in addition to a string lets you include a helpful message when the assertion line is printed when there is a failure. The string is not necessary: `assert(1==0)` would be sufficient to trigger a failed assertion.

NB: What you pass to `assert` is something you expect to *always be true* for correct operation of the program and, again, is a check that will be removed for the release mode of your program. For example in the `sqrt` example we have been using in lecture you might include an assertion that the input value is non-negative:

```
double sqrt583(const double& y) {
  assert(y >= 0);
```

```
  double x = 0.0, dx;
  do {
    dx = -(x*x-y) / (2.0*x);
    x += dx;
  } while (abs(dx) > 1.e-9);

  return x;
}
```

Compiler Pickiness Since a compiler is built to translate a program written in a given program language, it can also be used to analyze how programs are written. Both clang (llvm) and g++ use the flag "-Wall" to enable pickiness, meaning the compiler will issue warnings for just about anything in your program that might be suspect. Warnings are not fatal, your program will still compile and run if warnings are issued. But, as part of defensive programming, your programs should always compile cleanly with -Wall enabled. For maximal pickiness you can also use "-Wextra" and "-pedantic".

Language Level Support C++ is still being actively extended and improved upon by the ISO standards committee responsible for it. There are four versions of the language that represent major milestones in the evolution of the language: C++98, C++11, C++14, C++17. Although standards are developed (and named) by the standards committee, it does take some time for compilers as well as programs themselves to catch up to the standards. Although it is currently 2017, the default level of language support for clang and g++ is still C++98. This is a reflection more that the vast majority of extant C++ code is written in C++98 than it is a reflection of the timeliness of the compiler writers.

To specify a given level of language support (to enable certain features that are in one that are not in an earlier one), we can pass one of the following flags to the compiler: "-std=c++11", "-std=c++14", or "-std=c++17". Since C++17 is still very very new, you should not expect it to be well supported at this time (but neither will you need to use many of its features, if any at all).

For this course you are welcome to use any level of the C++ language that you like, but it is recommended to use at least C++11.

Leveraging Automation So now we have some number of flags that we would like to always be able to send to the compiler: "-g" (to enable debugging), "-std=c++11" (for C++11 language support), and "-Wall" (for pickiness). Using all of these flags consistently and correctly would be infeasible if we had to remember to type them by hand every time for every compilation. However, automation (with make) can make this almost transparent. You just need to add the flags once and for all to the consequent compilation rules in your Makefile. Consider the following example that consists of main.cpp, amath583.cpp, and amath583.hpp:

```
main.exe:    main.o amath583.o
        c++ -g main.o amath583.o -o main.exe

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

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

With the "good kind of laziness" we seek to replace repetitive tasks with automation. Even in the above Makefile, we can see some repetition. In subsequent assignments we will use more features of make to further automate the program build process. (One thing we will want to automate, for instance, is switching between debug and release modes of a program.)

2.2 Program Organization

In lectures 3 and 4 we presented a basic class for representing a Vector. In your ps2 subdirectory, create two files: Vector.hpp and Vector.cpp to hold the declarations and implementations of the Vector class. At this point the Vector.hpp declarations should just be what we presented in lecture.

Functions that operate on Vectors using just their external interfaces are neither declarations nor implementations of the Vector class. For now, we will put such functions (and we will be writing several in this assignment) into amath583.hpp and amath583.cpp (declarations and implementations, respectively).

Even though we haven't created any contents yet for amath583.cpp nor Vector.cpp, they should each still have their corresponding declaration file included.

3 Exercises

3.1 Reading Vectors from Files

In future programs we will be reading and writing our program data to text files, using the format presented in the first problem set. To support that we need some utility functions for reading and writing vectors.

Deliverable Add a function to `amath583.cpp`, `readVector(std::string ...)`, that takes a `std::string` specifying a filename, that opens that file for reading, and reads the contents into a Vector object and returns that object. If the function encounters and error, the program calling it should return a non-zero error code. For now, the function can call `std::exit()` (with appropriate exit code) upon error. Create the build rule for amath583.o in your Makefile.

Input The input will consist of lines of text, consisting of a header, an integer n, a sequence of n numbers, and a trailer. As above, lhe header string is "AMATH 583 VECTOR" and the trailer string is "THIS IS THE END". The numbers will be in integer or floating point format. Your program should so some basic checks for incorrect input. If either the header or the trailer are incorrect, your program (your main() function) should not print anything and should return a value of -1. If the value of n is less zero your program should not print anything and should return a value of -2. If the value of n is equal to zero or n does not exist (there are only the header and the trailer), your program should print 0 and return 0 (correct execution). For now we can assume all of the input numbers are well formed and that there are a correct number of them.

Output You don't need to deliver an executable as part of this exercise. However, you should think about how you might want to test this function. Your `readVector()` function should return a `Vector`.

Example The following is a sample input file for this problem.

```
AMATH 583 VECTOR
6
0.4
-1.3
2.141
3.14159
4.0
5
THIS IS THE END
```

Your function may be invoked in the following way (where foo.in contains data in the format shown directly above):

```
std::ifstream inputStream("foo.in");
Vector x = readVector(inputStream);
```

3.2 Generating Random Vectors

Deliverable The following function (implemented in `amath583.cpp` and declared in `amath583.hpp`) :

```
Vector randomVector(int N);
```

This function should create and return a `Vector` of length `N` and filled with random double-precision numbers. You don't need to write/deliver a test program for this function. But, again, you should think about how to test this function.

Example

```
Vector x = randomVector(N);
```

will create a Vector x of length N filled with random values.

3.3 Writing Vectors to Files and to cout

Deliverable Create two functions in `amath583.cpp`, that put formatted vectors onto `cout` or to an output file. You should have a function
`void writeVector(const Vector& ..., ostream& ...)`
as well as a `void writeVector(const Vector& ..., std::string ...)`.

In C++ the prototype of a function (its name that the compiler uses) includes the input types as well as the function name. Accordingly, you can use the same function name for different functions as long as there are different input types. The output type is not part of the function signature and can't be overloaded on.

Input. Write a test program that writes vectors to standard output or to a file. You can put the testing code in `testWrite.cpp`. You should add a target to your Makefile such that when "make testWrite" is executed, an executable called `testWrite` is generated. If your `testWrite` is invoked with a single command-line parameter, n, it should print a random Vector to cout (with the appropriate header and footer). If it is invoked with two command-line parameters, the first will be n and the second will be a file name to which the Vector should be written to instead of the standard output.

Output. The output will consist of a header string, the integer n, a sequence of n random floating numbers and a trailer string. The header string is "AMATH 583 VECTOR" and the trailer string is "THIS IS THE END".

Error Handling. You may assume that only integer values will be supplied to your program, but not that the value will be greater than zero. If the value of n is less than zero, your program should not print anything and should return a value of -2. If the value of n is zero, your program should print the header and trailer with nothing between them. If no value is passed to the program, your program should print a message:

```
Usage: ./testWrite n [ filename ]
```

Example. The following is sample invocation for this problem:

```
./testWrite 5
```

The following is sample output for this problem.

```
AMATH 583 VECTOR
5
1.3
9.2
```

```
3.8
5.4
5.9
THIS IS THE END
```

Suggested Test. If your readVector and writeVector are working properly you should be able to write a (simple) program that verifies that a Vector that you read from a file can be written back out – with the resulting file being identical to the file that was read in.

3.4 Vector Norm

Deliverable Create a function (in `amath583.cpp`) with the following prototype:

```
size_t infNormIndex(const Vector&  x);
```

Note that since `x` is not changed – and since we don't want to copy it – we pass it in as a const reference. The function should return the index of the variable with the largest absolute value in vector `x`. If there are multiple entries with the same maximum value, the function should return the lowest index (the first entry).

Next, reate a function (in `amath583.cpp`) with the following prototype:

```
double infNorm(const Vector&  x);
```

This function should use `infNormIndex()`.

Output Next, create a test program `infRandom` (with `main()`) that does the following. If the program is invoked with one argument, n, it should generate a random vector and print the index of the maximum value as well as the maximum value itself. If the program is invoked with two arguments n and a name, the program should print the index and maximum value to standard out (cout) and the vector itself (using writeVector()) to the specified file. Invoking the program with zero or more than two arguments should result in a printed usage statement and a non-zero return value. If the specified file name is "cout" your program should print the Vector to cout instead of to a file and the Vector should be printed after the index and max value are printed. You should add a target to your Makefile such that when "make infRandom" is executed, an executable called `infRandom` is generated.

Example

```
$ ./infRandom 5 cout
3 14.1
AMATH 583 VECTOR
5
5.5
3.6
4.2
14.1
8.2
THIS IS THE END
```

3.5 Inner Product (583 Only)

Deliverable Create a function (in amath583.cpp) with the following prototype:

```
double dot583(const Vector&  x, const Vector&  y);
```

Input Next, create a test program called `dot583` (with `main()`) that does the following.

```
Vector x = readVector(argv[1]);
Vector y = readVector(argv[2]);
double n1 = dot583(x, y);
cout << n1 << endl;
```

Add a target to your Makefile such that when "make dot583" is executed, an executable called `dot583` is generated.

Output Your program should compile to a program named `dot583` and should have its build rules incorporated into your Makefile. Assume that the filenames given for argv[1] and argv[2] are valid. Your program should check to make sure the right number of arguments are given and if they aren't it should return with a non-zero error code. Your program should check that the Vectors specified by argv[1] and argv[2] are of the same length. If not, it should print an error message and return with a non-zero error code.

Example

```
$ ./dot583 foo.in bar.in
7.015
```

4 Turning in The Exercises

Create a tarball ps2.tgz with all the files necessary to run your code 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 ps2.tgz file, it is **very important** that you have confidence in your code passing the automated grading scripts. Make use of the python script test_ps2.py by using the command `python test_ps2.py` in your ps2 directory. Once you are convinced your code is producing the correct output for the given input, upload ps2.tgz to Canvas.

Below are files that might be included in your tarball file: Makefile, Vector.hpp, Vector.cpp, amath583.hpp, amath583.cpp, testWrite.cpp, infRandom.cpp. For AMATH583 student, your tarball file might also include dot583.cpp.

5 Learning Outcomes

At the conclusion of week 2 students will be able to

1. Read and write data files into `Vector`.
2. Write basic C++ programs that take command line arguments in `argc` and `argv`
3. Create a Makefile to automate compilation of a single file program
4. Create a Makefile to automate compilation of a multiple file program
5. Explain "defensive programming"
6. Explain why you would use `assert()` in your programs