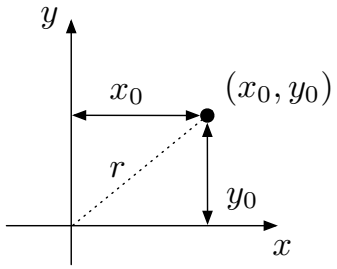
UNIVERSITY OF WASHINGTON
Department of Applied Mathematics

AMATH 483/583 **Assigned:** 4/4/2017
Problem Set #1 **Due:** 4/11/2017

1 Preliminaries

In Lecture 3 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. As discussed in Lecture 3, we will be using files as data input and output mechanisms for the computational capabilities we will be developing. As the capabilities of this library grow, we will factor it into multiple files. But, even in our initial organization of one source code file and one header file, 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 mode. This removal is done functionally rather than physically – you don't actually go through the code

[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.

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:

```
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.)

3 Exercises

3.1 Reading and Interpreting (Parsing) Command Line Arguments

In Lecture 3 we discussed how the main() program receives command-line arguments from the calling environment. As we also mentioned in class, we will only be doing basic command line processing. Nonetheless there is a design process to use that can be helpful in processing command line arguments in your programs.

First, define how you want the program to be invoked, what command line arguments you will pass, what flags, in what order, and what flags might be optional. If you try "man tar" in your docker container you will see an example of how a command line might be specified. Note also that if you issue "tar" without any arguments, you will get a usage message.

Consider a simple program that we want to print a sequence of number from 0 to $N-1$. To invoke it, we require that the value of N be passed into the program. If no value of N is passed in the program should exit with a non-zero error code. We also optionally take in a flag "-o" that will also have an argument indicating a file to write the output to. If there is no file specified, we send the output to cout. That is, the usage for the program would be:

```
$ ./a.out N [ -o outputfile ]
```

Thus, for processing the command line, we need to make sure there is at least one argument beyond the command itself, and then check for a -o flag (making sure there is a following argument).

Here is the (incomplete) skeleton of the argument processing logic for such a program:

```
if (argc < 2) {
  cout << "Usage: " << argv[0] << " N [ -o outputfile]" << endl;
}

int N = stoi(argv[1]);

string outputFilename = "";
for (int opt = 2; opt < argc; ++opt) { // skip argv[0] and argv[1]
  if (string(argv[opt]) == "-o") {
    ++opt;                          // skip to next token
    if (opt >= argc) {
      cout << "No output file specified" << endl;
    }
    outputFilename = argv[opt];
  }
}
```

Deliverable A source code file `seq.cpp` implementing a program for printing a sequence of numbers, along with a Makefile that defines the `seq` target (compiles `seq.cpp` into the executable `seq`)

Input Input to this program will be specified on the command line. When `seq.cpp` is compiled into an executable it should support the following calling convention:

```
$ ./seq N [ -o outputfile ]
```

Output If no output file is specified, the program should print a sequence of number from 0 to N-1 to the screen. Otherwise it should print the sequence to the specified output file. If no value of N is given the program should return a nonzero errorcode. If the -o option is given but no outputfile is specified, the program should also return a nonzero errorcode. If the program completes successfully it should return a zero errorcode.

Example

```
$ ./seq
Usage: ./seq N [ -o outputfile]

$ ./seq 4
0
1
2
```

```
3

$ ./seq 4 -o foo.txt
$ cat foo.txt
0
1
2
3

$ ./seq 4 -o
No output file specified
```

3.2 Reading Vectors from Files and from cin

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. As we have seen with some of our examples from lecture, file streams and standard I/O streams (cin and cout) have essentially the same semantics and same interfaces. The semantics are so similar in fact, that we can substitute one for the other in situations where that makes sense.

For example, if we would like to be able call readVector() with either cin or a file stream, we can define it with the following interface:

```
vector<double> readVector(istream& inputStream);
```

With this interface we can invoke readVector() in either of the following ways:

```
vector<double> x = readVector(cin);
// OR
ifstream inputStream("foo.in");
vector<double> x = readVector(inputStream);
```

The cin input stream and the file ifstream are not exactly the same type, they do share the same interfaces. This common interface is provided through a C++ mechanism known as "inheritance" (a fundamental property for object-oriented programming). One way of realizing *polymorphism* (using the same function on different types) is to use inheritance to create new types, but invoke the function on the base class on which the other types are based.

Deliverable Create a file named `amath583.cpp`. We will use this file to hold the HPC library that we will be building up during this course. The interfaces in `amath583.hpp` and the file itself (`amath583.hpp`) should be included in any other file that calls functions in `amath583.cpp`. (There was an example of this in the slides of Lecture 2 and Lecture 3.) You should also include `amath583.hpp` in `amath583.cpp` as a defensive programming technique to insure the interface declarations also match the function definitions. This file should contain two functions (to begin with). For the first, create readVector(istream& ...) that has the same functionality as defined in the last assignment, only now takes an istream& object. You may use your own function, a rewritten version of your function, or the instructor solution for this. For now, the function can call std::exit (with appropriate exit code) upon error. For the second, create a readVector(std::string ...) function that takes a std::string specifying a filename, that opens that file for reading, and then invokes the previous readVector(istream& ...) function. Finally, the target `amath583.o` should be defined in your Makefile.

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 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 The same as the previous assignment for vectors. (whether read from a file or fron cin).

Output You don't need to write an executable for this exercise. Your readVector() function should return a vector<**double**>. The program should exit with the same exit codes as before.

Example Your function may be invoked in either of the following ways.

```
vector<double> x = readVector(cin);
// OR
ifstream inputStream("foo.in");
vector<double> x = readVector(inputStream);
```

3.3 Generating Special Vectors

Deliverable The following function (in `amath583.cpp`):

```
vector<double> randomVector(int N);
```

It should create and return a vector<**double**> of length N and filled with random numbers. Write a test program that takes in an integer as an argument and outputs the elements of the random vector. The executable, when compiled by an appropriate make target, should be named `genRandomVector`. You can assume the integer, if present, is non-negative.

Output An appropriate list of carriage-return separated numbers.

Example

```
$ ./genRandomVector 5
```

3.4 Vector Norms

In Lecture 3 we introduced the formal definition of vector spaces as a motivation for the interface and functionality of a Vector class. But besides the interface syntax, our Vector class needs to also support the semantics. One of the properties of a vector space is associativity of the addition operation, which we are going to verify experimentally in this assignment.

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

```
double twoNorm(const vector<double>& 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 Euclidean (2) norm of the vector x.

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

```
vector<double> x = randomVector(N);
sort(x.begin(), x.end());   // sort vector in ascending order
double n1 = twoNorm(x);
sort(x.rbegin(), x.rend());   // sort vector in descending order
double n2 = twoNorm(x);
cout << n1 << " " << n2 << " " << n1-n2 << endl;
```

Output Your main program should take the size of the vector on the command line and print the string above. It should compile to a program named `vectorNorm`. What does your program print as you increase the vector size (the value of N)?

Example

```
$ ./vectorNorm 5
.13 .13 0
```

3.5 Writing Vectors to Files and cout (583 only)

Deliverable Create two functions in `amath583.cpp`, similar to your readVector functions, that put formatted vectors onto cout or to an outputfile. You should have a function
void writeVector(**const** vector<**double**>& ..., ostream& ...)
as well as a **void** writeVector(**const** vector<**double**>& ..., std::string ...) .

3.6 Inner Products (583 Only)

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

```
double dot583(const vector<double>& x, const vector<double>& y);
```

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

```
vector<double> x = randomVector(N);
vector<double> y = randomVector(N);
double n1 = dot583(x, y);
cout << n1 << endl;
```

Output Your main program should take the size of the vector on the command line and output the dot product. It should compile to a program named `dot583`.

Example

```
$ ./dot583 5
```

3.7 Written Exercises

Create a text file called ex2.txt in your ps1 folder. In that file, copy the following questions. Start your answers on a new line after each question and leave a blank line between the end of your answer and the next question.

1. At what size N does your Euclidean norm start to exhibit unexpected results? What is the approximate magnitude of the unexpected result? Explain.
2. Is this going to be a problem with our realization of vector spaces as vector<**double**>?

4 Turning in The Exercises

Create a tarball ps2.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 ps1.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 Collect It.

5 Learning Outcomes

At the conclusion of week 2 students will be able to

1. Read and write data files into vector<**double**>, both from a file and from cin.
2. Write basic C++ programs that take command line arguments in argc and argv
3. Describe a basic model for single core CPU operation
4. Explain the functionality of level-1 and level-2 cache
5. Be able to estimate the performance improvement achieved because of level-1 and level-2 cache
6. Create a Makefile to automate compilation of a single file program
7. Create a Makefile to automate compilation of a multiple file program
8. Explain "defensive programming"
9. Explain why you would use assert() in your programs