Git repo: https://github.com/JeffreyJacob1/Chem_281_hw3

Problem 1:

Problem 1 is located in the folder problem 1, it has the make system and a simple chemistry program for example purposes.

type
make

to generate everything accordingly

./bin/main to run exec

Make clean

To clean

2.1: `using n = 1000000 for 100 iterations`

The program took the following time to run
412,143 microseconds spent in program unit 'Vector Operations'

2.2:

Got the following times

51,998 microseconds spent in program unit 'std::vector Operations (single loop)'
139,007 microseconds spent in program unit 'C Array Operations'

It is clear that the single loop version works the fastest and the malloc the second fastest.

Primitive C arrays have the benefit of having contiguous memory, this can make are code more efficient through better memory accsess patterns and superior cache utilization. This allows the CPU to operate on the data more quickly. Using a single loop is more efficient as it reduces some of the overhead of funtion calling and allows the O3 optimization to be more effective by

Git repo: https://github.com/JeffreyJacob1/Chem_281_hw3

allowing for more unrolling, vectorization and register allocation. On the other hand, operator overloading introduces function call overhead for each operation. Each overloaded operator function is called for every element in the vectors, resulting in additional function calls and memory accesses. This is why 1 was the slowest in our testing.

2.3

Got
<lambda> took 198298.22 microseconds

This is faster than our code in 1 but slower than both of our implementations in 2. Numpy does use optimized c code under it's hood which makes it fast for these array problems, however it still lags our optimized C code which uses Malloc, this shows that the lower level control granted by C and hardware optimizations done in the compiler are hard to beat. The python code additionally has to deal with the added overhead of python. Malloc allowed us to control our memory allocations avoiding the use of vectors, without this overhead it is able to be very fast and the clear winner over numpy.

2.4

Got

```
using n = 1000000 for 100 iterations
```

254,275 microseconds spent in program unit 'Armadillo'
262,715 microseconds spent in program unit 'Eigen'
259,972 microseconds spent in program unit 'std::valarray'

```
using n = 1000 for 10000 iterations
```

100,155 microseconds spent in program unit 'Armadillo'
66,300 microseconds spent in program unit 'Eigen'
71,293 microseconds spent in program unit 'std::valarray'

The implementations here do come very close to the hand coded c, although not beating it. When lowering n it looks like armadillo becomes slower than eigen and val array. None of these methods beat the single loop hand coded c however.

2.5:

At O3:
73,199 microseconds spent in program unit 'AVX-512 Operations'

And at O0:

Git repo: https://github.com/JeffreyJacob1/Chem_281_hw3

240,104 microseconds spent in program unit 'AVX-512 Operations'

Optimization is important to AVX intrinsics code, at O0 the compiler generates straightforward machine code without taking advantage of the vectorization AVX offers. 01 and 02 are significantly better with 03 being the best. 01 and 02 are very good only slightly worse than 03. Overall at)3 the intrinsics does offer a boost in performance to the code thanks to SIMD instructions leveraging parallelization.