Bucks for Brains Student Report 2018

Ben Stringer

At its core, my research task was to programmatically implement several matrix algorithms, including the Faddeev-Leverrier algorithm, with arbitrary precision. A convenient tool for this was MATLAB, which many students may download and install freely. MATLAB’s focus is on matrices and their operations, so it was a quick task to write a MATLAB script implementing Faddeev-Leverrier. (In most of my code this has been shortened to Fadlev (**Fad**deev-**Lev**errier), which came from an early version of the algorithm I downloaded from MATLAB’s File Exchange.)

Using MATLAB’s “vpa” (variable-precision arithmetic) and “digits” functions, I was able to run Fadlev on a 100x100 matrix of integers between 1 and 10 in around 3 minutes. For a 100x100 matrix, Fadlev will calculate the 101 (increasingly large) coefficients of the matrix’s characteristic equation, so just looking at the results was not a great way to confirm their accuracy. It is known that the last coefficient Fadlev produces is equal to the determinant of the original matrix, multiplied by -1 to the nth power (n being the number of rows/columns in the matrix). Under certain circumstances, the determinant of a matrix can be very easy to calculate, such as when the matrix is upper or lower triangular. The script for testing accuracy therefore would generate a triangular matrix, calculate the determinant by a simple product of the entries on the main diagonal, and then compare this number to the last coefficient computed by Fadlev for that same triangular matrix. The results confirmed that the last coefficient was indeed correct, and due to its dependence on all the previous terms, there was reasonable evidence that the other coefficients were correct, too.

Now that I had a working program, I was able to concern myself with MATLAB’s performance with arbitrary precision numbers: when running the program without the use of VPA, it took only tenths of a second. I looked around for other arbitrary precision tools and found Advanpix’s Multiprecision Computing Toolbox for MATLAB. Its website showed huge speed improvements compared to Matlab’s VPA, and even cases where VPA lost accuracy. This toolbox was not free, but there was a free trial. The toolbox was meant to be an analog to VPA, so implementing Fadlev using the new toolbox (MP) was quick. And it was just as quick to run the algorithm too; for the same 100x100 as before, MP ran in less than 3 seconds, around 100x faster than Matlab. It is important to note that because the complexity of (naïve) matrix multiplication is , halving the size of the matrices to 50x50 will yield an 8-fold speed increase; similarly, doubling to 200x200 will take roughly 8 times as long. (It is probable that Matlab and MP use better matrix multiplication algorithms, but these are only marginally better at this size of matrix.)

The speed at which MP ran inspired me to look at compiled solutions, namely from C++. I discovered the GNU Multiple Precision Arithmetic Library, GMP for short. I installed GMP onto my Ubuntu virtual machine and started experimenting. GMP has three types of numbers: integers, rationals, and floats. Initially I only experimented with the integers, and eventually got a basic Matrix class with naïve multiplication implemented (there are methods of making matrix multiplication faster). Timing the result of multiplying two random 100x100’s together showed similar results to MP, so I was excited and hopeful that my C++ implementation would be the “way forward.”

Gradually I expanded on the Matrix class, adding functions for trace, entrywise (Hadamard) product, scaling by a constant, etc. I eventually added support for rational matrices so that the inverse of a matrix could be calculated as a nice byproduct of Fadlev. Once the various Matrix operations were implemented, I wrote the Fadlev algorithm to use those matrices. Compiling with the optimization -O3 flag, I ran this new Fadlev on a 100x100 matrix and timed it: under 3 seconds! It still wasn’t quite as fast as MP, but it blew Matlab out of the water (and it wasn’t just a free trial).

Another class of matrix algorithms were specific to hollow, symmetric matrices (hollow as in a main diagonal of zeros, and symmetric as in the i,j entry is equal to the j,i entry). These algorithms are referred to as trace algorithms, because the algorithms are sums of traces. Each algorithm is quite long to implement, but also straightforward if efficiency is not necessary; the amount of memory used by these functions can be reduced if the ordering of the calculations is analyzed—for example, I reduced the first trace algorithm from storing 14 matrices in memory to just 9.

The rest of the programming I did was focused on user experience and extendibility. I wanted to make all the different capabilities of the project easy to run, but I also wanted it to be easy to add future functionality. Additionally, I wanted to make matrix input easy and robust. Finishing these features made the project a convenient and quick tool for matrix research. There is even an extensive Readme that comes with the code which helps with GMP installation, learning all the different functionality, and doing one’s own tests.

There is still plenty more to be done, however. Perhaps the biggest change would be to investigate GMP’s C++ class interface. A bolded warning heads the page in the GMP manual: “Everything described in this chapter is to be considered preliminary and might be subject to incompatible changes if some unforeseen difficulty reveals itself.” If that doesn’t scare one away though, there are several conveniences the class interface provides. For starters, GMP variables no longer need initialization: you can declare and define them all on one line. Several operators have also been overloaded, and functions in general return values rather than voids, as they do with C GMP. So, overall it would be much easier, or at least less tedious, to write code using the class interface.

Some improvements could also be made to performance, but for the immediate purposes of this project it was unnecessary to pursue these. However, if larger matrices are desired, an implementation of Strassen’s algorithm for matrix multiplication might be due.

Lastly, while I’ve tried to make the code easily extendible, especially with the creation of “mpImpl” and the templating of the matrix class, there are obvious cases where it leaves something to be desired. The best example of this is main.cpp, after all the initial argument checking. 8 calls were made to the “fadlevAlgorithm” function, when I was planning on only using 1. The problem was that there was no way for me to declare a general matrix at a broad scope that could later be filled in inside the nested if statements, except with the use of the heap (which I only later discovered).