Matrices and Matrix Operations in OCaml

Akshar Bonu, Jayant Sani

Demo video

Overview

Both of us took Math 21b: Linear Algebra and Differential Equations, this semester, so we were interested in bringing linear algebra to OCaml by implementing a matrix module that could perform a variety of matrix operations. Some operations that we had in mind included matrix multiplication, solving linear systems or Gaussian Elimination, LU decomposition, and possibly even determinants.

We wanted to implement these matrix operations because, by hand, these processes are very time-intensive. Matrix multiplication and solving linear systems using Gaussian Elimination are also key elements of any matrix library.

With regards to matrix multiplication specifically, it is normally of time O(n^3). By implementing the Strassen Algorithm, we plan to bring it down to O(n^2.807), or O(n^log7). While we considered implementing algorithms like the Coppersmith–Winograd algorithm, which has been built upon by a variety of scientists to yield an algorithm of O(n^2.3728639), this will not be attempted in this project because the gains in the algorithm are only for extremely large matrices that are so large that they cannot be processed by modern hardware. Click here for details about the implementation of the Strassen algorithm.

Put simply, the algorithm divides the given two matrices into four sub-matrices, doing computations on the resulting eight matrices to yield seven new matrices, hence the runtime. Further operations yield the eventual matrix product. This algorithm does not run into any problems for matrices with dimensions of 2^n, because the matrix can be evenly divided into four matrices until the base case of a single element. The trouble occurs when we want to multiply matrices with odd dimensions. One could just add columns and rows of zero until reaching the next dimension of 2^n that can be easily divided forever (static padding), but that technique wastes a lot of memory and most of the algorithm will be working on zeros. An extension that we implemented was dynamic padding. Now, an additional row or column is added in every recursive call if the rows and/or columns are odd so that the matrix could be divided into four matrices at least once - before that call returned, the extra rows and columns (if any) of the product had to be removed before being returned. This means any two matrices can be multiplied (regardless of square or rectangular) as long as their dimensions allow multiplication to occur.

LU decomposition has the same complexity as the multiplication algorithm, which in this case is Strassen, so O^(n^log7). Click here for the details about the implementation of this algorithm, LU Decomposition, also called LU factorization, is a method that factors a matrix into the product of a Lower triangular matrix and an Upper triangular matrix. Initially, we had intended to be able to do LU decomposition for matrices that could only be factored into the form A = LU, without pivoting. Sometimes, the rows of the initial matrix A need to be reordered for the matrix to be decomposed. This method yields a pivot matrix that reorders the rows of A such that PA = LU. We had initially intended to only implement LU decomposition for matrices that do not require pivoting, but we included an extension to the implementation such that it works for all square matrices.

Gaussian Elimination is also a time-intensive process that involves three types of row operations.

Type 1: Swap the positions of two rows,

Type 2: Multiply a row by a nonzero scalar

Type 3: Add one row and a scalar multiple of another

Page 2 of this pdf describes the algorithm in greater detail.

Due to time-constraints, we were unable to apply our matrix operations to sports rankings as our furthest extensions, but we were able to optimize our algorithms. As mentioned earlier, our extensions include dynamic padding for the Strassen matrix multiplication algorithm allowing multiplication of any two matrices. In addition, LU decomposition now works for any square matrix as a result of pivoting instead of just matrices that can be converted into the form A = LU. We also implemented a determinant algorithm that reduces a matrix by rows through creating minors.

Planning

We had initially planned to bring a variety of matrix operations, such as multiplication, solving linear systems, and LU decomposition, to OCaml. Here are the links to our draft specification and final specification.

Overall, our planning worked out well because we decided to implement a specific number of algorithms, a task that was easily divided. At first, Akshar coded up the skeleton of the module including the basic helper functions (i.e. identity, zero, add, sub, do_operation) and we both looked over it so that we both knew the interfaces and data structures with which we were going to work. We then split up the implementation of the algorithms: Akshar was assigned with the Strassen algorithm and Jay the LU Decomposition and Gaussian Elimination. LU Decomposition can also be described as Gaussian Elimination with matrices, so the process was pretty similar for both algorithms. We both also did testing for our algorithms with both members helping one another when necessary.

The milestones worked pretty well, as we stuck to our schedule of finishing the skeleton of the code so that we knew what we were working with early. The functionality checkpoint after working on the project for a week can be found here. Within a week of finishing the final specification, we were able to complete the core of the functionality, what we had hoped to submit as a minimum finished product. Therefore, we were able to implement some extensions, such as dynamic padding, pivoting, and determinant calculation, by the time we submitted the project.

Design and implementation

Our experiences with languages, testing, and other related topics largely came from CS50 and CS51.

As a result of the influence of CS51 and its design practices, we decided to modularize our code into separate parts so that it would make the most logical sense. We therefore created a matrix signature that defined the necessary methods that a matrix would need. We then made a matrix struct that would contain the implementations of all our algorithms for multiplication, solving linear systems, determinant, etc. as well as their helper functions.

Initially, we had different files for each algorithm (LU.ml, Strassen.ml, etc), which was very difficult for testing because we needed to merge all the algorithms together into one module in order to test the implementations. We ended up creating two separate testing files, Akshar-Testing.ml and Jayant-Testing.ml, to test the respective functions that we were supposed to implement. At the end, we merged these two files to contain the most up-to-date versions of our code for the final matrix module.

We ended up splitting the code into three separate files because we had three separate elements of the code, the tests, the types of the elements of the matrix, and the matrix functor itself. Therefore, our code consists of three files, matrix.ml, ring.ml, and test.ml. The matrix file opens the ring file, and the test file opens both of the other files.

Implementing the Strassen algorithm was difficult because at first it was challenging to logically split up and join matrices as was necessary in the algorithm. Next, when making the algorithm more powerful to allow the multiplication of any two matrices, Akshar had to research about dynamic padding. The academia on dynamic padding is not very clear and it was challenging initially to understand it how to implement it, but after reading through numerous papers, he was able to understand it and then worked on how to manipulate matrices to add rows and columns and then remove them at the appropriate times in the recursive calls.

Implementing the LU decomposition, Gaussian Elimination for solving linear systems, and determinant algorithms was not terribly difficult. Fortunately, from websites such as Wolfram Alpha, Jay was able to gain information about such algorithms and how they work at a high level. Jay then implemented the algorithms in OCaml in the best way that he saw fit.

Reflection

We were pleased by how we worked together in the beginning of the project to create the skeleton of the code so that we both knew the interface and data structures with which we were working. We then split up the algorithms and coalesced them into one matrix module at the end.

We were surprised by how the testing did not originally work well logistically. We each were working on our own algorithms and wanted to test them. Therefore, we created separate testing files with modified matrix modules so that we could test the implementations of our algorithms. We ran into some problems, but eventually fixed them and merged our code at the end. For example, to test the LU decomposition function, the multiplication function was needed to see if the outputted matrices actually multiplied to be the original matrix.

Our decision to modularize the code into three separate files was great, as that separation made logical sense due to the three disparate elements that we were merging together in the element type, matrix, and tests. This decision resolved the problem that we originally had of putting each algorithm into a separate file. It only made sense to do so at the end, though, when the separate testing procedures had been finished.

If we had more time, we would probably implement additional functions and algorithms that are more specific to sports rankings, our lofty goal. For example, functions that parse input of sports scores or even allowing our program to accept user input. But, our algorithms, specifically the Gaussian Elimination for solving linear systems, has the ability currently to figure out sports rankings if the appropriate matrix and solution are inputted.

Advice for Future Students

Definitely start the project early and create weekly milestones for yourselves. We started coding a good amount in advance, pretty much finishing the skeleton of the module very early, much before the project was due. We then finished our main functionality about a week before and were therefore able to add extensions and tweak our code accordingly before submitting.