CS 51 Final Project Functional Specification[Matrix Module and Simplex Algorithm] —[Annotated!]

Luis Perez | luisperez@college.harvard.edu
Andy Shi | andyshi@college.harvard.edu
Zihao Wang | zihaowang01@college.harvard.edu
Ding Zhou | dzhou@college.harvard.edu

May 5, 2013

[Note for the reader: This is the annotated version of our CS 51 Final Project Functional Specifications. All annotations will appear in this color.].

1 Brief Overview

Our project seeks to implement an efficient matrix library for Ocaml. The end goal is to utilize this matrix library in conjunction with the simplex algorithm to find solution **[Linear Programs]**. In order to accomplish this, there will be multiple small checkpoints. We need to implement and efficient matrix representation (so as to not use the naive implementation of lists of lists)**[(Successfully accomplished. Our implementation uses array for efficienty)]**. We also hope to provide a useful interface that allows the user to input arbitrarily large matrices effectively (possible by mapping over the entries of the matrix)**[The user can input matrices with a simple text file composed of comma seperated values]**. One big challenge for the matrix library will be the implementation of efficient algorithms for multiplication and row reduction. We expect the multiplication to initially consists of the standard algorithm, but will eventually grow into either block multiplication or Strassen **[We were not able to implement more sophisticated multiplication algorithms, but it seems that the current algorithm works fairly well]**. We might even implement some way to chose between the algorithms depending on the inputted matrices, but that would not an essential feature **[This feature can still easily be implemented, but there simply was no time. We already knew this, though, by the time we wrote our Technical Specs]**.

The row reduction algorithm will certainly be one of the most fundamental algorithms in our matrix library implementation. From our current understanding of simplex, it seems like it will heavily rely on row reduction **[The simplex algorithm ended up relying on a different type of row reduction, so we couldn't just plug in a matrix into our row reduction algorithm. We had to do a lot of extra work to get simplex. Though, many of the helper functions for row reduction ended up being extremely useful for Simplex, so this was not as pointless as some might think]**. The only algorithm we can come up with so far is Gauss-Jordan, though

we have found some algorithms which are more suited for computation by computers with imprecise storage of floats **[In fact, we just had to be careful which element to choose as a pivot. Futhermore, we ended up implementing arbitrary precision, via Ocaml's num library, so this became a non-issue]**.

The simplex part of the problem is the main goal, but if we can get down the matrix module, then it appears as if the simplex algorithm should be straightforward **[This was definitely not the case]**. Eventually, we wish to implement a simplex module that allows the input of arbitrary linear equations with linear constraints, and then we parse this information into the corresponding matrix and finally perform the simplex algorithm on that matrix in order to return a solution to the specified problem.**[This is fully functionalt. Input can be made in the form of a text file with a linear program. The algorithm runs and not only returns the right values for simple cases, but also checks for cases that might initially appear to be unsolvable but aren't, and for cases which are unbounded.]**

Other tentative goals are to implement matrices that can accept either arbitrarily large numbers (bignums) or arbitrarily precise floats (`int list * int stream`). It would also be interesting, though not necessary, to implement multiple algorithms for each operation and choose from them the one that is the best for a specific input data **[We were able to implement arbitrary precisions and arbitrarily large sizes with the OCaml Nums library]**.

2 Feature List

1. Find a way to represent a matrix (arrays or list list or other representation)—will likely use an array representation using the Ocaml array module (`http://caml.inria.fr/pub/docs/manual-ocaml/libref/Array.html`) **[Implemented as specified]**
2. Make a matrix from a list of lists (this is how the user will input the data of the matrix). **[Implemented as specified]**
3. Scalar multiplication—as described here (`http://www.purplemath.com/modules/mtrxmult.htm`) **[Implemented as specified]**
4. Matrix addition—as described here (`http://www.purplemath.com/modules/mtrxmult.htm`) **[Implemented as specified]**
5. Standard Matrix multiplication (`http://www.purplemath.com/modules/mtrxmult.htm`) **[Implemented as specified]**
6. Row Reduction (Gauss-Jordan). We will use the following resource to help us calculate row reduction when the entries of the matrix are stored with limited precision (`http://thejuniverse.org/PUBLIC/LinearAlgebra/LOLA/rowRed/var.html`) **[Implemented as specified]**
7. Matrix inverse / Row reduction with LU Decomposition - Description can be found here (`http://www.math.ust.hk/~macheng/math111/LU_Decomposition.pdf`) **[To find the inverse, we augmented the original matrix with an identity matrix, then row-reduced the augmented matrix]**

8. **[Make a matrix from a text file. The user is now able to read in a text file and create a matrix.]**
9. **[For Information of further functionality, take a look at MatrixI.ml.]**
10. Simplex Algorithm (Described in Algorithms book, in addition to the resources provided below) **[Implemented mostly as specified, with some modifications from the Wikipedia article on simplex]**
11. Matrix norm **[Did not implement since it was not necessary for Simplex.]**
12. Matrix transpose **[Implemented as specified.]**
13. Trace—This is a typical trace (sum of the entries on the diagonal) **[Implemented as specified.]**
14. Determinant—found using algorithm described in Section 4.8 of Hubbard's *Vector Calculus, Linear Algebra, and Differential Forms: A Unified Approach* (Math 23a/b textbook) **[Implemented by decomposing the matrix into upper and lower triangular matrices, then multiplying the entries along the diagonals for both and multiplying the two results]**
15. Eigenvectors / Eigenvalues, calculated also by a method in Hubbard's book (this might not be necessary to carry out the simplex algorithm and we might abandon it if we can't figure out how to implement it in a timely manner) **[Did not implement since it was not necssary for Simplex.]**

3 Draft Technical Specification

3.1 Matrices

Our matrix module will actually be a functor that takes in an `ORDERED_AND_OPERATIONAL` module (from ps4 and moogle) so we can have matrices of ints, floats, or floats as streams. **[This actually ended up being very helpful since at the end we we able to use OCaml's Num Library in order to achieve arbitrary precision and size.]**

The following type definition (a la ps4) will give us a way to talk about order for different data types.

```
type order = Equal | Less | Greater
```

The following signature will allow us to provide matrix functionality on a variety of data types. For example, we could make an `ORDERED_AND_OPERATIONAL` type for floats, where add would be defined as (+.), for example. In our matrix functor, we will then use the functions defined in the signature below (for example, add instead of (+.)). Also, we have included some generating functions to help us with testing.

[For the updated view of this, please take a look at EltsI.ml]

```
module type ORDERED_AND_OPERATIONAL
sig
  type t

  val zero : t
  val one: t
  val compare : t -> t -> order

  (* Converts a t to a string *)
  val to_string : t -> string

  val add: t -> t -> t
  val subtract: t -> t -> t
  val multiply: t -> t -> t
  val divide: t -> t -> t

  (* For testing *)
  (* Prints a t *)
  val print: t -> unit

  (* Generates the same t each time when called *)
  val generate: unit -> t

  (* Generates a t greater than the argument passed in *)
  val generate_gt: t -> unit -> t

  (* Generates a t less than the argument passed in *)
  val generate_lt: t -> unit -> t

  (* Generates a t in between the two arguments. Returns none if none exist *)
  val generate_between: t -> t -> unit -> t option
end
```

The following signature will provide all the necessary functions that can be performed on a matrix. We include basic matrix operations like additions, multiplications, scalar multiplication, and then standard operations like finding the inverse of a matrix, finding its trace, determinant, eigenvalues, transpose, norm, and how to row reduce the matrix.

Below each function, in comments, is a brief description of how we will implement the function.

[For the updated view of this, please take a look at MatrixI.ml]

```
module type MATRIX =
sig
  exception NonSquare

  type elt
  type matrix
```

```
(* Type of this is unknown, but will probably be represented using Ocaml's
 * built-in Arrays *)
(* empty matrix *)
val empty

(* Takes a list of lists and converts that to a matrix *)
val from_list : (elt list list) -> matrix
(* Will implement using nested match statements *)

(* Scales every element in the matrix by another elt *)
val scale : matrix -> elt -> matrix
(* Will implement by iterating through the matrix and scaling each element *)

(* Adds two matrices. They must have the same dimensions *)
val add : matrix -> matrix -> matrix
(* Will add the elements elementwise and construct a new matrix *)

(* Multiplies two matrices. If the matrices have dimensions m x n and p x q, n
 * and p must be equal, and the resulting matrix will have dimension m x q *)
val mult: matrix -> matrix -> matrix
(* Will take the dot product of the nth row of the first matrix and the jth
 * column of the second matrix to create the n,j th entry of the resultant *)

(* Returns the row reduced form of a matrix *)
val row_reduce: matrix -> matrix
(* We will implement the algorithm found in the link above *)

(* Returns the inverse of a matrix *)
val inverse: matrix -> matrix
(* Will implement this based on the specification in the Algorithms book *)

(* Returns the norm of the matrix *)
val norm: matrix -> elt

(* Transposes a matrix. If the input has dimensions m x n, the output will
 * have dimensions n x m *)
val transpose: matrix -> matrix
(* Will basically ``flip'' the indices of the input matrix

(* Returns the trace of the matrix *)
val trace: matrix -> elt
(* Will check if the matrix is square, then sum up all the elements along its
 * diagonal *)

(* Returns the determinant of the matrix *)
val det: matrix -> elt
(* Will implement this algorithm based on a description in Hubbard. Involves
 * column reducing the input (or row-reducing the transpose) and then keeping
```

```
   * track of the operations to build a sequence of coefficients to multiply *)

  (* Returns a list of eigenvalues and eigenvectors of a matrix *)
  val eigen: matrix -> (elt *matrix) list option
  (* Calculates successive powers of the input matrix, each multiplied by the
   * same basis vector. Generates a polynomial and solves for zeros, which
   * yields eigenvalues. Repeat for all basis vectors *)

  (* Takes a string and builds a matrix from it *)
  from_string : string -> matrix
  (* We will have some way to express matrices using strings, and then we will
   * parse the string to give the matrix *)

  (* Prints out the contents of a matrix *)
  print :matrix -> unit
  (* Iterate through the matrix and print each element *)
end
```

The exception exists in case the user tries to perform operations, such as trace or determinant, which require a square matrix. Matrices have their own type, and they contain elements of the type `elt`.

We will probably need a helper function to raise matrices to powers and to solve for zeros of a polynomial if we want to implement the `eigen` function.

3.2 Simplex Algorithm

The simplex solves linear programs, which are basically linear inequalities. Given inequalities, using the simplex algorithm, we can find the maximum value of a given expression. What the simplex algorithm does is that it takes in several linear inequalities and writes them so that they are equalities. For instance $5x + 7y \leq 10$ would be $5x + 7y + s = 10$ where $s \geq 0$ is the slack variable. Then it takes all these equations and puts them a a matrix where each entry represents the coefficient of the variable in the expression. The first row represents the equation we are optimizing. The final column is the constant values of each equation. Then we will do a series of computations so that the top row has all nonnegative values. This can be done using the functions from the Matrix module.

The steps are as follows:

Look along the first row and find the smallest negative element. Look at the column of that entry and compute the value/element of each row. If any element is zero, we can ignore it. Otherwise take the element with the smallest value/element and make this the pivot point. Reduce the column so that the pivot point in 1 and any other entry in the column is zero. Repeat the process again until the top row consists of only nonnegative values. The first entry in the last column is the maximum. In addition, for any column consisting of all zeroes and a one, we can find the value of the corresponding variable by looking at the value of the row containing the one. If the column doesn't consists of zeroes and a one, then the value of the corresponding variable is zero.

This algorithm also works with variables that are negative, inequalities where the unknowns is

greater than a constant, and minimum of the equation. This can be easily done by either multiplying the whole inequality by -1 or by negating the variable. If calculating the minimum or negating the variable, we can revert them later.

[For the updated view of this, please take a look at SimplexI.ml]

```
module type simplex
sig
  type elt
  type linear_equation
  type constraint

  val pivot : matrix -> matrix
  val constraint_from_string : string -> constraint
  val linear_from_string : string -> constraint
end
```

These are helper functions that we think will be used:

`no_neg : matrix -> bool`

Check the top row has no negative elements, which can be done by using the next helper function

`min_of_row : row -> elt*(elt ref)`

A helper for the row reduction which will find the minimum element and its location. **[We ended up needing many more helper functions than those shown here.]**

4 What's Next

We will read more into the algorithms and get a basic understanding of how they all work so we can better judge the difficulty of each algorithm. We will also read more into OCaml to see how to put together a project with multiple files (since we have always relied on staff's frameworks for each pset). We will working through the CS50 appliance. We already have setup a git repository on Github (`https://github.com/Fantastic-four`), and most of us are fairly familiar with version control.