

CS 51 Final Project Final Specification

Luis Perez | luisperez@college.harvard.edu
Andy Shi | andyshi@college.harvard.edu
Zihao Wang | zihao wang01@college.harvard.edu
Ding Zhou | dzhou@college.harvard.edu

April 14, 2013

1 Overview

We have decided to reduce the amount of functions available to the outside world. The matrix module will still contain the same functionality as before, but only the functions necessary for the implementation of the Simplex algorithm will be exposed (row-reduction, for example). The main focus of the project has now shifted to having a working simplex algorithm. Other functions, such as finding the eigenvalues of a matrix, while interesting, have been placed in our “Cool Features” category. Furthermore, following the idea that Simplex is our main focus, additional functionality such as parsing input, providing arbitrary float precision and/or providing a matrix implementation with bignums has been degraded in importance. All of our attention will be focused on creating a working simplex algorithm.

2 Detailed Description

2.1 Signatures and Interfaces

2.1.1 Matrices

Our matrix module will actually be a functor that takes in an `ORDERED_AND_OPERATIONAL` module. The `ORDERED_AND_OPERATIONAL` module will be defined as is below. There will be a global order for the elements, also as defined below. In our beta implementations, we plan to simply use Ocaml floats as the elements, but, if time allows, the plan is to make the simplex algorithm rely on abstract as matrices a possible so that we can change the `ORDERED_AND_OPERATIONAL` to something more efficient and/or accurate.

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

```
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 functions in the module, since they will be dedicated to matrices and optimized for matrix elements, will not be exposed to the outside world. The module will be passed in as a constructor to the Matrix module (defined later), which will expose only that while is, once again, necessary for the Simplex module. Additionally, the `ORDERED_AND_OPERATIONAL` module will more than likely have additional values and functions not exposed in its signature. For example, we plan to define the “epsilon for comparison” for floats inside the module, but we keep it hidden simply because it’s not necessary anywhere else.

Users will have no direct access to this module, and therefore there are not standards that need to be enforced (other than those in place by the interface).

The following signature is the one for the Matrix module, and it also the one that has changed the most. Most of the functions we originally had were not necessary for the simplex algorithm, so we have made them “cool features” which we will attempt to implement if we have time.

```
module type MATRIX =
sig
```

```
exception NonSquare
exception ImproperDimension

type elt
type matrix
(* represented using Ocaml's built-in Arrays *)

(* empty matrix *)
val empty

(* returns the nth row of a matrix as a matrix*)
val get_row : int -> matrix
(* implemented in terms of get_elt *)

(* returns the nth column of a matrix as a matrix *)
val get_coln : int -> matrix
(*also implemented in terms of get_elt *)

(* returns the element in the nth row and mth column *)
val get_elt : int -> int -> matrix
(* will just get the (n,m) index of an array *)

(* 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

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

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

The exceptions exists in case the user tries to perform operations which require a square matrix or matrices with certain dimensions. Matrices have their own type, and they contain elements of the type `elt`. The following is a list of helper functions in the module, but not exposed to the outside world.

```
(* 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
 * where the matrices are a single dimensional arrays *)
val dot -> matrix -> matrix
(* Will implement this based on the specification in the Algorithms book *)
```

The following are cool features which we will try to implement if we have time.

Cool Features:

```
(* Returns the inverse of a matrix *)
val inverse: matrix -> matrix

(* 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 *)
```

2.1.2 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 in 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 is 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 consist 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.

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

  val pivot : matrix -> matrix

  (* Will implement these if we have time *)
  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.

2.2 Modules and Actual Code

Please see our attached files or our repo at www.github.com/Fantastic-Four/ocaml-matrix. Files of note include `Elts.ml` and `Matrix.ml`.

2.3 Timeline

- Week 1 (4/7–4/13):
 - Signature definitions
 - Setting up Github (everyone on the same page)
 - Writing makefile
 - Establishing work distribution
- Week 2 (4/14–4/20):
 - Basic matrix functionality
 - Some rudimentary form of matrix multiplication and row reduction
 - Finishing up the element types
- Week 3 (4/21–4/27):
 - Beta testing the simplex algorithm
 - Implementation of simple input parser for testing (we will most likely be inputting arrays)
 - Completion of the matrix module algorithms
- Week 4 (4/28–5/4)
 - Fully working simplex algorithm
 - Implement a better matrix multiplication algorithm
 - Add some of the cool features!

2.4 Progress Report

We have already implemented the design structure for all of the files (specifically, each file has a signature detailing what it needs to do). The Makefile is also ready, while rudimentary functionality has been implemented in the matrix module allowing most of the functions we want matrices to have to be available already. For evidence of our progress please look at `Elts.ml` and `Matrix.ml`.

3 Version Control

We are currently using github for version control. Our repository can be found here, under the organization Fantastic Four (www.github.com/Fantastic-Four/ocaml-matrix). We had some difficulties at first getting the repo set up on everyone's computers but we fixed that over the weekend and we should be good to go.