Statically Typed Linear Algebra in Haskell

Frederik Eaton
frederik@ofb.net

ABSTRACT

Many numerical algorithms are specified in terms of operations on vectors and matrices. Matrix operations can be executed extremely efficiently using specialized linear algebra kernels in libraries such as ATLAS or LAPACK. The resulting programs can be orders of magnitude faster than naive implementations in C, and this is one reason why matrix computation interpreters such at Matlab and Octave are popular in scientific computing. However, the process of expressing an algorithm in terms of matrices can be error-prone. Typical matrix languages are weakly-typed. If we could expose certain properties of operands to a type system, so that their consistency could be statically verified by a type checker, then we would be able to catch many common errors at compile time. We call this idea "strongly typed linear algebra" and describe a prototype implementation in which dimensions are exposed to the type system, which is based on Alberto Ruiz's GSLHaskell [Ruiz(2005)] and uses techniques from Kiselyov and Shan's "Implicit Configurations" [Kiselyov and Shan(2004)].

A great advantage of Matlab is the ability it offers scientists to manipulate and inspect numerical objects interactively. We show how to make our library useful for interactive use, using Template Haskell.

Next, we implement a medium-sized machine learning algorithm using our library, and compare it to a similar implementation in Octave. Drawing from this experience, we suggest Haskell language features which might improve the library's usability. Perhaps surprisingly, we conclude that performance is not a problem area for Haskell. The Haskell version of the program, which takes several minutes to run, is almost twice as fast as the Octave version, and probably comparable to the speed of Matlab.

A basic understanding of Haskell is assumed.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming

General Terms: Design, Languages

Keywords: Linear algebra; implicit parameters; higher-rank polymorphism; type classes

1. INTRODUCTION

Many numerical algorithms are specified in terms of matrices and vectors. In addition, a common numerical programming technique is to express iterative algorithms in terms of operations on vectors and matrices. One class of transformations might turn the loop

```
for $i = 1 \,..\, n$
   $a\,[i] = f\,(b\,[i])$
end
```

into

$$a = F\,(b)$$

where F is a vectorized version of f; in another instance we might convert

```
for $i = 1 \,..\, n$
   $a\,[i] = 0;$
   for $j = 1 \,..\, m$
      $a\,[i] += w\,[i,\,j] \cdot b\,[j]$
   end
end
```

into a matrix-vector multiplication $\mathbf{a} = W \times \mathbf{b}$. Because linear algebra has proven to be a very general framework, highly optimized libraries have been written to perform common linear algebra operations as efficiently as possible, such as ATLAS (implementing the standard BLAS interface) and LAPACK.

The source of the speedup which these libraries accomplish usually comes from better cache utilization - by breaking a matrix down into blocks and processing it block-by-block, where each block is small enough to fit in the entire L1 cache. In ATLAS, the optimal block-size as well as many other algorithm parameters are determined automatically, during compilation of the library, so the resulting object code can be specially optimized to a given architecture. In addition, the libraries may use a CPU's vector instructions. Neither of these optimizations is something that a C compiler can very easily accomplish generically. Therefore a naively-written C program can run orders of magnitude more slowly than the corresponding ATLAS routines.

Linear algebra libraries thus enjoy widespread use in the scientific community. They are usually accessed via high-level programming languages such as Matlab's M-code. However, such languages and even much more sophisticated computer algebra systems are weakly typed. Even in strongly typed languages, types for linear algebra objects are usually limited to distinguishing between vectors and matrices, and specifying element type, for instance $Vector(Real)$ or $Matrix(Complex)$.

We think that if object dimensions were exposed to the type system, then it would be possible to catch a large number of common errors at compile time. For instance, matrices can generally only be multiplied in a certain order - in a matrix multiplication $A \times B$, the number of columns of A matches the number of rows of B. But in Matlab and its clones[1], if matrices are multiplied in the wrong order, the error is only caught at runtime. If the type system kept track of which dimensions matched which, such a mistake would have been detected at compile time, and what is generally called "operand conformability" could be statically guaranteed. We will give the name "strongly typed linear algebra" to any tool which makes such static guarantees possible. (Of course, not all mistakes cause operand conformability errors. For instance, a square matrix has the same dimensions as its inverse, so forgetting to invert a matrix will not be noticed by a strongly typed linear algebra system.)

We explore the implementation of a strongly typed linear algebra system in the functional programming language Haskell. To the best of our knowledge, it is the first such system to exist.

2. DIMENSIONS AS TYPES

The main idea is to encode vector dimensions as types. Dimensions are types which are instances of special class, *Dom*, and indices are members of those types. The class *Dom* has methods for complete enumeration of index values, so that given a type which is an instance of *Dom*, we can list the entire range of values which that type can (legally) have.

```
class (Bounded a, Enum a, Ix a, Eq a, Show a) ⇒ Dom a
  where
    domain :: [a]
    domainSize :: a → Int
    ...
```

Now, for instance, we can define a function with the following type

```
vector :: Dom a ⇒ (a → e) → V e a
```

to construct vectors. Here *V e a* represents a vector with element type *e* and index type *a*. The definition of *vector* will fill in all of the elements of the result by calling the given function on every value of type *a*, which is just the list returned by *domain*.

We guarantee that two dimensions match by using the same type variable for each of them. (Note that this is a stronger constraint than requiring that the vectors have the same size, because two different domain types may have the same number of elements by coincidence. However, in such a case we usually won't want the dimensions to match, so requiring the types to be the same is appropriate)

For example, the dot product function could have the following type:

```
(Num e, Dom a) ⇒ V e a → V e a → e
```

Because both arguments share the same dimensions type, the vectors are guaranteed to be the same length.

As another example, consider the case when we want to extract a subset of a vector (here, "(!) :: *V e a* → *a* → *e*" is the subscripting operation):

```
slice :: (Dom a, Dom b) ⇒ (b → a) → V e a → V e b
slice v f = vector (λ i → v ! f i)
```

For each index i of the result vector, the given function is applied, yielding an index $f\ (i)$ in the source vector. The element at index i of the result is taken to be the element at index $f\ (i)$ in the source. Essentially, vectors are functions from indices to elements, the index type can be seen as occupying a contravariant position[2]; this is why the function argument of *slice* is $b \to a$ rather than $a \to b$.

However, we can also think of a "vector" in the sense of linear algebra, i.e. one whose elements are taken from a field (also for a module, whose elements are taken from a ring), as representing a member of the dual space, which consists of linear maps from ordinary vectors to members of the field (or ring). In the dual space, the index type holds a covariant position. Thus, when the element type is a number, we have another useful transformation, which complements *slice*.

```
margin :: (Num e, Dom a, Dom b) ⇒
  (a → b) → V e a → V e b
```

The effect of *margin* is to map elements of the input according to the supplied function. Where multiple input index values map together, the elements at those indices are added; when there is an output index which is not in the range of the supplied function, its element is set to zero. The name comes from probability theory, from the term "marginal distribution".

2.1 Matrices

We represent matrices as vectors which are indexed by a pair:

```
V e (a, b)
```

This is a special case of a vector - if a and b are instances of *Dom*, then (a, b) is also an instance:

```
instance (Dom a, Dom b) ⇒ Dom (a, b) where
  ...
```

Pairs are enumerated in lexicographic order, e.g.:

```
[(0, 0), (0, 1), (0, 2), ..., (1, 0), (1, 1), ...]
```

Matrix multiplication has the following signature:[3]

```
(⊗) :: V e (a, b) → V e (b, c) → V e (a, c)
```

We define other common operations with reference to Matlab's M-code syntax:

```
trace :: Dom a ⇒ V e (a, a) → e    -- matrix trace
diag :: Dom a ⇒ V a → V (a, a)    -- diagonal matrix
eye :: Dom a ⇒ V e (a, a)    -- identity matrix
ones, zeros :: Dom a ⇒ V e a    -- vector of ones or zeros
```

etc. Note that the last three functions don't take any arguments, whereas in Matlab they each take two arguments. The reason is that in Matlab those arguments serve to specify the dimension of the result, but here the result dimensions are inferred from context. This often leads to more concise code, which is moreover closer to typical mathematical notation; however, occasionally we need to add type signatures to resolve ambiguities:

```
slowSize m = trace (ones `asTypeOf` m)
```

2.2 Reflecting Values

We haven't yet said anything about how domain types are created.

Consider the task of writing a function, *listVec*, which creates a vector from a list of elements. The size of the list is arbitrary; we

[1]Notably Octave, which is developed under the GPL and is mostly compatible with Matlab; also, Scilab is a mostly-free and has syntax similar to Matlab. We only use Octave for experiments - general benchmarks comparing it to Matlab are here: `http://www.sciviews.org/benchmark/index.html` Octave seems to be about two or three times slower than Matlab, but it is constantly improving.

[2]In a function type, the return value of a function is a covariant position; any of its arguments are contravariant positions. More generally, a position is contravariant if an odd number of its containing positions are arguments of functions, and covariant otherwise. So the position of a is covariant in $(a \to r) \to r$, and contravariant in $a \to r$ or $k \to a \to r \equiv k \to (a \to r)$, and covariant in a and $r \to a$.

[3]Since matrix multiplication is not commutative, we have chosen to use an asymmetric operator.

want to create a vector whose dimension reflects that size. Clearly we will need to find a way to store arbitrary integers in types. However, that is not the only difficulty. What should the type of our function be? A first attempt might look like:

$$listVec :: [e] \to V\ e\ a$$

However, here the type variable a is (implicitly) universally quantified over the entire expression; in other words it cannot depend upon the argument to *listVec*. What we need is something like

$$listVec :: [e] \to (\exists a.\ V\ e\ a)$$

where the result is an existential type. Then, each invocation of *listVec* can choose a new type for the type-variable a.

Existential types are not supported directly in Haskell, but we can encode them using universals as follows. The trick is based on a slight modification of the CPS (continuation-passing style) transformation.

The CPS transformation encodes values of type a as values of type $(a \to r) \to r$:

$$x \Longrightarrow \lambda\ f \to f\ x$$

We can reverse the encoding by applying the result to $id \equiv \lambda\ x \to x$ (although this does not give an isomorphism in a non-strict language, since the encoding was not surjective).

We can look at the transformation from a to $(a \to r) \to r$ as being the same as two applications of, for some r,

$$N_r\ a \equiv a \to r$$

A single application of N_r changes the place of a from covariant to contravariant, and can be seen as a kind of negation. The encoding of existentials via universals: [4]

$$\exists a.\ T\ (a) \Longrightarrow (\forall a.\ T\ (a) \to r) \to r \equiv N_r\ (\forall a.\ N_r\ (T\ (a)))$$

can then be seen as analogous to the logical equivalence:

$$\exists x. P(x) \iff \neg \forall x : \neg P(x).$$

Using this representation, we now have a type for *listVec*:

$$listVec :: [e] \to (\forall a.\ V\ e\ a \to w) \to w$$

The problem of encoding integers and other values in types is discussed in detail in [Kiselyov and Shan(2004)], which should be referred to for more detail. The two functions we use from the paper are the following:

class *ReflectNum s* **where**
  reflectNum :: *Num a* ⇒ *s* → *a*
reifyIntegral :: *Integral a* ⇒
  a → (∀*s*. *ReflectNum s* ⇒ *s* → *w*) → *w*

The function *reifyIntegral* encodes an integer as a type belonging to class *ReflectNum*; we can then call *reflectNum* on a dummy value (just ⊥) of that type to recover the encoded integer. Both functions are defined in the module *Misc.Prepose* in our library.[5]

[4]We could also introduce a data type

data *ExistsT* = ∀*a*. *Exists* (*T a*)

which is used in essentially the same way:

convert :: *ExistsT* → (∀*a*. *a* → *r*) → *r*
convert (*Exists x′*) *f* = *f x′*

However, it turns out that this would just make our code more verbose [Kiselyov and Shan(2004), §3.1].

[5]We have made a slight modification to representation of integers used by *Misc.Prepose*. In the original version of Kiselyov and Shan, a binary representation was used, with the most significant digit last. In our representation, we have switched to decimal, with the most significant digit first, to make error messages slightly easier to read:

```
> let v = useFast $ $(dim 13) ones
> let u = $(dim 42) v
    Couldn't match 'X4 (X2 X_)'
        against 'X1 (X3 X_)'
```

Using those facilities, a complete definition of *listVec* is now possible. The existential type variable is now constrained to be an instance of *ReflectNum*.

We use the following simple type to represent an index type taking on integers between 0 and $(n - 1)$: [6]

newtype *L n* = *L Int*
instance *ReflectNum n* ⇒ *Dom* (*L n*)

The definition of *listVec* finally becomes:

listVec :: [*e*] → (∀*n*. (*ReflectNum n*) ⇒ *V e* (*L n*) → *w*) → *w*
listVec l f =
  let *n* = *length l* **in**
  reifyIntegral n (λ (_ :: *rn*) →
    f (*fromList l* :: (⊥ :: *V e* (*L rn*))))

where *fromList* is a function which converts a list to a vector of known size.

The drawback of this approach is that the vector is only available to the function argument to *listVec*; after *listVec* has returned, it is gone. There is no possibility of passing *listVec* a function such as *id* which returns its own argument, because the type of *listVec* will not allow it. The type variable w can be given any type as long as that type does not depend on n, because w appears outside of n's quantification expression.

Thus it usually becomes necessary to structure a program as an argument to a series of functions such as the one above. The return type w is often a monadic action such as *IO* (); all of the interesting work - calculations, printing results - will then be done within the argument to *listVec*.

In an interactive environment, this is unfortunately not practical. We discuss methods for using the library interactively later on.

2.3 Type of Singular Value Decomposition

The type of the function for singular value decomposition (SVD) is worth mentioning. The SVD is defined for any matrix A, and expresses A as a product

$$A = UDV^\top$$

where U and V are orthogonal (unitary), and D is diagonal with positive entries ordered decreasingly. We would like an *svd* function to return the matrices U and V, and a vector which contains the entries in D. The catch is that the number of entries in D, as well as the number of columns of U and V, is the minimum of the number of rows and columns in A. How do we express this in a type signature? We could define D's size to be, say, the number of rows in A, and pad it with zeros when A has more rows than columns. But this is inefficient. Or we could define a type-class *Min a b c* which expresses the property that c is the minimum of a and b. But that would be difficult, and perhaps not sufficiently general. Instead, we have chosen to use the existential type technique again. Our definition is:

class (*Fractional e*, *NumVector v e*) ⇒ *FracVector v e* **where**
  ...

[6]The "*L*" stands for "linear". Since *L* is an instance of *Num*, and the *Show* instance hides the constructor, we generally think of it as just another numeric type:

```
> domain :: [L (Plus (X5 X_))]
[0,1,2,3,4]
```

```
svd :: (Dom a, Dom b) ⇒ v (a, b) →
  (∀c. Dom c ⇒
    (v (a, c), v c, v (b, c)) → r)
  → r
```

Here, the type c is existentially quantified and cannot escape the function argument. This is rarely a problem - in most uses of the SVD (for example, in calculating the pseudo-inverse), we will do something involving the diagonal matrix D, and then multiply by U and V on both sides to get a matrix with the same dimensions as A or $A^{\top}$. In case this isn't enough, variants of *svd* with more specific types can be easily written with *unsafeReshape*, described below.

2.4 Reshaping

Sometimes it is necessary to convert between dimensions which are provably the same, but for which the type checker is unable to derive their equality. For instance, consider the following function:

```
type S = L One
toRow :: Dom a ⇒ V e a → V e (S, a)
```

It converts a vector to a "row vector", i.e. a matrix with one row, and whose column dimension is the same as that of the input vector. We could implement this function with *slice*, but that might be inefficient. Instead, we have provided a function for converting dimensions, *unsafeReshape*. It "casts" a vector from one dimension type to another. If the two types have a different number of elements, then there is a runtime error - hence the "unsafe". However, it can be used to implement "safe" functions such as *toRow*, where it is known by the programmer that the input and output dimensions will always be the same size.

```
unsafeReshape :: (Dom a, Dom b) ⇒ v a → v b
toRow :: Dom a ⇒ V e a → V e (S, a)
toRow = unsafeReshape
```

3. BACKEND

In this section we discuss the implementation of vectors.

As with arrays, one can imagine more than one vector implementation. The semantics could be strict or lazy; the internal representation could be boxed or unboxed.

In order to accommodate multiple implementations, almost all of the vector operations are members of classes. The main class is *Vector*, but *NumVector* (*vsum*, ⊗, *margin*, *ones*, ...), *OrdVector* (*vmin*, *vmax*), *FracVector* (*inv*, *mean*, ...), and *FloatVector* (*logdet*, *rand*) also exist.

In order to allow instances to specify a restricted class of element types, we use a GHC extension called functional dependencies [Jones(2000)]:[7]

```
class Vector v e | v → e where
  ...
```

For example, the *FVector* instance is declared as

```
instance Vector FVector Double where
  ...
```

and the *AVector* instance as

```
instance Vector (AVector e) e where
  ...
```

Unlike the element type, there is no way to put a restriction on the index types which a *Vector* instance will accept, aside from membership in *Dom* which is required by each method:

```
class Vector v e | v → e where
  ...
  vector :: Dom a ⇒ (a → e) → v a
```

We have provided two data types for which instances of these classes are defined. The first, *AVector*, encapsulates an *Array*, and can hold any element type. It is relatively slow for numerical computations (about 200 times slower than GSLHaskell).

3.1 The GSLHaskell Backend

The second vector type we have implemented, *FVector* (for "fast") is based on Alberto Ruiz's GSLHaskell library, which uses *ForeignPtr*s to store arrays compactly in C format, and employs the GSL, ATLAS, and LAPACK for numerical operations. It only supports the *Double* element type.

3.1.1 Detecting Matrices

The most difficult part about writing a vector backend based on GSLHaskell was figuring out how to distinguish matrices and vectors in routines that handle both, such as *vector*. We would like to create a *GSLMatrix* for matrix objects, and a *GSLVector* for vector objects. [8]

In our API, anything with a pair index type is a matrix; and everything else is a vector. But there is no built-in facility in Haskell for querying the type of a variable. At first glance, one should be able to use *dataCast2* of Ralf Lammel's generics library [Lämmel and Jones(2003)]:

```
dataCast2 :: Typeable2 t ⇒ (∀a b. (Data a, Data b) ⇒
  c (t a b)) → Maybe (c a)
```

This facility allows us, once we define some helper functions, to take two different courses of action depending on whether a dimension is a pair or not. However, if it is a pair then we will need the member types of the pair to be instances of *Dom*; but *dataCast2* only ensures that they are members of *Data*. So the SYB library is unfortunately not useful to us.

Instead, what we have done is to add a special member function to the *Dom* class:

```
class (Bounded a, Enum a, Ix a, Eq a, Show a) ⇒ Dom a
  where
    ...
    domCastPair :: c a → y a →
      (∀d e. (Dom d, Dom e) ⇒
        (c (d, e)) → y (d, e)) →
        y a
    domCastPair _ def _ = def
```

For most domains, the default given above is what we want. But for pairs it has been redefined as:

```
domCastPair v _ fn = fn v
```

Thus, in pseudo-code, we can describe the action of *domCastPair* as:

```
domCastPair v def fn =
  if (v is parameterized by a pair type) then
    fn v
  else
    def
```

Here is a small example. Sometimes, it is necessary to define a dummy datatype to get the arguments to match the signature of

[7] "Associated Type Synonyms" provide a possibly more intuitive way of doing the same thing [Chakravarty et al.(2005)Chakravarty, Keller, and Jones].

[8] Although it would be possible to only store *GSLVector*s, and convert each to a *GSLMatrix* whenever an operation calls for a matrix, this would require calculating the sizes of matrix column and row dimensions repeatedly, probably involving calls to *reflectNum* each time, which can be inefficient

domCastPair, as we have done below:

$$\begin{array}{l}
\textbf{newtype}\ \mathit{ArrayWrap}\ e\ a = \mathit{AW}\ (\mathit{Array}\ a\ e)\\
\textbf{instance}\ \mathit{Vector}\ \mathit{FVector}\ \mathit{Double}\ \textbf{where}\\
\quad \ldots\\
\quad \mathit{fromArray}\ (ax :: \mathit{Array}\ a\ \mathit{Double}) :: \mathit{FVector}\ a =\\
\quad\quad \mathit{domCastPair}\ (\mathit{AW}\ ax)\\
\quad\quad\quad (\mathit{GV}\ \$\ \mathit{GSL.fromArrayV}\ ax)\\
\quad\quad\quad (\lambda\ (\mathit{AW}\ ax') \to (\mathit{GM}\ \$\ \mathit{GSL.fromArrayM}\ ax'))\\
\quad \ldots
\end{array}$$

(*fromArrayV* and *fromArrayM* are functions from GSLHaskell for constructing vectors and matrices, respectively)

3.1.2 The FVector datatype

The *FVector* datatype uses the GADT extension:

$$\begin{array}{l}
\textbf{data}\ \mathit{FVector}\ a\ \textbf{where}\\
\quad \mathit{GV} :: \mathit{GSL.GSLVector}\ \mathit{Double} \to \mathit{FVector}\ a\\
\quad \mathit{GM} :: (\mathit{Dom}\ a,\ \mathit{Dom}\ b) \Rightarrow\\
\quad\quad \mathit{GSL.GSLMatrix}\ \mathit{Double} \to \mathit{FVector}\ (a,\ b)
\end{array}$$

This way, it is possible to avoid having to call *domCastPair* for a vector which has already been created. This is used, for example, in the definition of the indexing operator (!):

$$\begin{array}{l}
\textbf{instance}\ \mathit{Vector}\ \mathit{FVector}\ \mathit{Double}\ \textbf{where}\\
\quad \ldots\\
\quad (!)\ (\mathit{GV}\ agv)\ k = \ldots\\
\quad (!)\ (\mathit{GM}\ agm)\ (i,\ j) = \ldots\\
\quad \ldots
\end{array}$$

In the *GM* version of (!), we have made use of the fact that the compiler is able to infer that the second argument is a pair.

4. INTERACTION

One of the great features of languages such as Matlab and Octave is their ability to facilitate interactive experiments via a command line interface. However, in a typical client of our library, vectors only exist within a function called by a routine such as *listVec*. So, it is not possible to create them with one command and use them with another. [9]

We solve the problem with Template Haskell [Sheard and Jones(2002)]. The resulting API is not as concise as Matlab, but it is usable.

Our primary observation is that if we could make vectors instances of *Lift*:

$$\begin{array}{l}
\textbf{class}\ \mathit{Lift}\ t\ \textbf{where}\\
\quad \mathit{lift} :: t \to \mathit{ExpQ}
\end{array}$$

then we could put them in quasi-quotes $[|\ \cdot\ |]$. Since the result, *ExpQ*, would then be independent of the vector dimension, this would allow us to write something like:

$$\textbf{let}\ v = \$(\mathit{listVec}\ [1,\ 2,\ 3]\ (\lambda\ v \to [|\ v\ |]))$$

This is essentially what we have done. It is possible to define a *Lift* instance which is applicable to all vectors; however, since there is no single vector datatype, this would lead to problems with overlapping instances. So we settle for a function:

$$\begin{array}{l}
\mathit{liftVec} :: (\mathit{GetType}\ e,\ \mathit{Lift}\ e,\ \mathit{GetType}\ a,\ \mathit{Dom}\ a,\ \mathit{Vector}\ v\ e,\\
\quad \mathit{GetType}\ (v\ a))\\
\quad\quad \Rightarrow v\ a \to \mathit{ExpQ}
\end{array}$$

where we have defined *GetType* in *Misc.Prepose*:

$$\begin{array}{l}
\textbf{class}\ \mathit{GetType}\ s\ \textbf{where}\\
\quad \mathit{getType} :: s \to \mathit{Type}
\end{array}$$

[9] One can imagine an interpreter which makes this possible, but it would be very tricky and as far as I know, none exists.

With the following helper functions

$$\begin{array}{l}
\mathit{qArray} :: (\mathit{GetType}\ e,\ \mathit{Lift}\ e) \Rightarrow\\
\quad ((\forall w.\ (\mathit{Dom}\ w,\ \mathit{GetType}\ w) \Rightarrow\\
\quad\quad \mathit{AVector}\ e\ w \to \mathit{ExpQ}) \to \mathit{ExpQ})\\
\quad \to \mathit{ExpQ}\\
\mathit{qArray}\ f = f\ (\lambda\ x \to \mathit{liftVec}\ x)\\
\mathit{qFast} :: ((\forall w.\ (\mathit{Dom}\ w,\ \mathit{GetType}\ w) \Rightarrow\\
\quad \mathit{FVector}\ w \to \mathit{ExpQ}) \to \mathit{ExpQ}) \to \mathit{ExpQ}\\
\mathit{qFast}\ f = f\ (\lambda\ x \to \mathit{liftVec}\ x)
\end{array}$$

we can now write for example[10]

$$\begin{array}{l}
> \textbf{let}\ v = \$(\mathit{qArray}\ (\mathit{listVec}\ ([1\mathinner{..}4] :: [\mathit{Double}])))\\
> v\\
< 1.0,\ 2.0,\ 3.0,\ 4.0 >\\
> \mathit{dot}\ v\ v\\
30.0
\end{array}$$

In addition there are macros which create identity functions for vectors or matrices with integer indices of the specified dimensions:

$$\begin{array}{l}
\mathit{dim} :: \mathit{Int} \to \mathit{ExpQ}\\
\mathit{dim2} :: \mathit{Int} \to \mathit{Int} \to \mathit{ExpQ}
\end{array}$$

and identity functions which force the use of a specific implementation:

$$\begin{array}{l}
\mathit{useFast} :: \mathit{FVector}\ a \to \mathit{FVector}\ a\\
\mathit{useFast}\ v = v\\
\mathit{useArray} :: \mathit{AVector}\ e\ a \to \mathit{AVector}\ e\ a\\
\mathit{useArray}\ v = v
\end{array}$$

Together, these allow us to write things like:

$$\begin{array}{l}
> \textbf{let}\ m = \$(\mathit{dim2}\ 3\ 4)\ \$\ \mathit{useFast}\ \mathit{ones}\\
> m\\
< \#\ 1.0,\ 1.0,\ 1.0,\ 1.0;\\
\quad 1.0,\ 1.0,\ 1.0,\ 1.0;\\
\quad 1.0,\ 1.0,\ 1.0,\ 1.0\ \# >\\
> m \otimes (\mathit{eye} + \mathit{ones})\\
< \#\ 5.0,\ 5.0,\ 5.0,\ 5.0;\\
\quad 5.0,\ 5.0,\ 5.0,\ 5.0;\\
\quad 5.0,\ 5.0,\ 5.0,\ 5.0\ \# >
\end{array}$$

Thus, creating a new vector is a bit clumsy, but manipulating it is straightforward.

Lastly, we should note a few ongoing issues with the approach. First, there is an open bug in Template Haskell which prevents us from lifting vectors above a certain size:

```
> $(qFast (listVec [1..10000]))
ghc-6.4.2: panic! (the ‘impossible’ happened,
    GHC version 6.4.2):
        linkBCO: >= 64k insns in BCO
```

This is not a huge burden, since there are ways around it. For instance, if the dimensions of a vector are known, then one can use *dim* and *fromList* rather than *listVec*. The second problem is that programs using Template Haskell currently can't be profiled. This is a more serious problem, and means that we must recommend against using any of our template facilities outside of an interactive interpreter.

[10] Note that $\$(\mathit{qArray}\$\mathit{listVec}\ ([1\mathinner{..}4]::[\mathit{Double}]))$won't typecheck. The function (\$)has type $\forall\ b\ a.\ (a \to b) \to a \to b$, which doesn't explicitly mention the universal quantification in the *a* variable. Since GHC implements predicative polymorphism, in which type variables cannot be instantiated by polymorphic types, \$'s second argument type cannot be unified with the partial application of *listVec* [Jones and Shields(2005)] [Botlan and Remy(2003)].

Lastly, it would be nice to be able to use Template haskell in type signatures, since we are primarily using it to create types. However this doesn't seem to be possible [Jones(2006)].

5. EXPERIMENT

To better understand both the efficiency and the usability of our library, we have implemented a medium-sized algorithm from machine learning, which is described in more detail here: `http://www.gatsby.ucl.ac.uk/~zoubin/course05/lect7var.pdf`.

The exact computation which is performed is not important. Briefly, it is a variational EM (expectation-maximization, ref Dempster et al) algorithm for learning the parameters of the "latent binary factors model" (described in the next section). The algorithm was chosen because it is only partially amenable to expression in terms of matrices - a certain amount of looping over indices is necessary as well - and because of its familiarity to the author. The reader can skip to section 5.2 at this point if desired.

5.1 Algorithm

The model is described as follows. There are K binary latent variables $s_i \in \{0, 1\}$, parameters $\theta = \{\{\boldsymbol{\mu}_i, \pi_i\}_{i=1}^K, \sigma^2\}$, where each $\boldsymbol{\mu}_i$ is a real-valued vector with dimension D, and an observation vector $\mathbf{y}$ of dimension D distributed as:

$$p(\mathbf{s}|\boldsymbol{\pi}) = \prod_{i=1}^{K} p(s_i|\pi_i) = \prod_{i-1}^{K} \pi_i^{s_i}(1-\pi_i)^{(1-s_i)} \qquad (1)$$

$$p(\mathbf{y}|s_1, \ldots, s_K, \boldsymbol{\mu}, \sigma^2) = \mathcal{N}(\sum_{i=1}^{K} s_i\boldsymbol{\mu}_i, \sigma^2 I) \qquad (2)$$

We are given N samples of $\mathbf{y}$, and must estimate the parameters and the posterior over each s_i. The variational approximation we choose ignores dependencies between the variables s_i in the posterior, it estimates $p(\mathbf{s}|\mathbf{y})$ as $\prod_i p(s_i|\mathbf{y})$. In our code, we define $\lambda_i \equiv p(s_i|\mathbf{y})$. The most time consuming part is the E-step, the calculation of $\boldsymbol{\lambda}$, which is done by iterating

```
for $i = 1..K$
    $\lambda_i \leftarrow \mathrm{sigm}(\log \frac{\pi_i}{1-\pi_i} + \frac{1}{\sigma^2}(y - \sum_{j\neq i} \lambda_j \boldsymbol{\mu}_j)^\top \boldsymbol{\mu}_i - \frac{1}{2\sigma^2}\boldsymbol{\mu}_i \top \boldsymbol{\mu}_i)$
end
```

where $\mathrm{sigm}(x) \equiv 1/(1+e^{-x})$ is the logistic (sigmoid) function. This is done in the function *meanFieldStep*.

The M-step updates the parameters using the $\boldsymbol{\lambda}$ vector from the E-step; it is a straightforward maximum-likelihood calculation which is easily expressed in terms of matrix operations.

5.2 Implementation

The Haskell implementation is shown in appendix A, and the Octave implementation in appendix B.

There are many advantages to using Haskell in this kind of application, which we don't go into in detail. For instance, proper closures are apparently not available in Octave or Matlab, and although we don't need to use them here, they are often very useful. Easy concurrency is another benefit of using Haskell.

In the following sections we describe only the *problems* we encountered in implementing the Haskell version of the algorithm, stemming from our use of the Haskell language and the compiler GHC.

5.3 Purity Problems

The first difficulty is that the iterative update above needs to be in exactly that form for convergence. As functional programmers, we'd like to update all of $\boldsymbol{\lambda}$ at once, with each new λ_i depending only on the old value of $\boldsymbol{\lambda}$. However, in this algorithm each λ_i needs to be updated independently, using the updated versions of previous elements $\{\lambda_j\}_{j=0}^{i-1}$, otherwise the dynamics of the fix-point iteration are unstable.[11] To facilitate this, we added a member to the *Vector* class, *vectorUpdate*:

```
class Vector v e | v → e where
  ...
  vectorUpdate :: (Dom a) ⇒ v a → [a] →
    (v a → a → e) → v a
```

This takes a vector, a list of indices to modify, and a function, and returns the "updated" vector obtained by applying the function to each index and to the partially updated vector. For the GSLHaskell-based *FVector*, this has an efficient implementation which does all of the modifications in-place.

The function was sufficient for the example program, but it is not possible to use it with more than one vector at once, so it is possible that a generalization might be needed at some point. (One could imagine a "mutable vector" class, with *freeze* and *thaw* operations in analogy to the array modules in the standard libraries)

5.4 Laziness Problems

Since the program is iterative, we need to be sure that all the computations of one iteration are performed before entering the next iteration. Typically, this is done with the *seq* built-in; however *seq* only forces the evaluation of a single thunk, and therefore isn't sufficient for data with more than one level of structure, and won't work automatically for all *Vector* instances.

Another possibility is to have all our data-types implement a class such as *DeepSeq* (ref); however, this would require many additional class membership annotations in polymorphic programs wishing to make even limited use the facility.

What we have done instead is add a member *vseq* to the *Vector* class, so that all users of the *Vector* API have a way to force the evaluation of a vector. This leaves open the possibility of more organized approaches such as *DeepSeq*, but doesn't commit to them.

With this, our program need only define a function to force the whole state between iterations:

```
data State v d k = S{
  sMu :: v (d, k),
  sSigma2 :: R,
  sPie :: v k
}
seqState s = vseq (sMu s). vseq (sPie s).
  seq (sSigma2 s)
```

Dealing with such considerations is a task which is a bit tiresome and is not required in strict languages, and which is absent from the Octave version of our program. On the bright side, with experimental features such as Rob Ennals' "Optimistic Evaluation" [Ennals and Jones(2003)] it would seem to be unnecessary. However, methods such as his for prospective thunk evaluation seem to have only enjoyed marginal popularity, because they are complex to implement and maintain, and because they can cause significant slow-downs in programs with many small thunks. In vector-based numerical computations, though, most of the memory and time tends to be spent on a few large thunks. We find it unfortunate that there is currently no working optimistic evaluation patch for GHC.

5.5 Casting

[11] Other important algorithms, such as the Gauss-Seidel method, use similar iterative updates.

Another part of the *Vector* interface which was only added after experience with the example program were the following methods:

```
type One = Plus (X1 X_)
type S = L One
class Vector v e | v → e where
  ...
  toRow :: Dom a ⇒ v a → v (S, a)
  toCol :: Dom a ⇒ v a → v (a, S)
  toS :: e → v (S, S)
  fromRow :: Dom a ⇒ v (S, a) → v a
  fromCol :: Dom a ⇒ v (a, S) → v a
  fromS :: v (S, S) → e
  byRow :: (Dom a, Dom b) ⇒ (a → v b) → v (a, b)
  byCol :: (Dom a, Dom b) ⇒ (b → v a) → v (a, b)
```

The first six express simple dimensional equivalences - viewing a vector as a row or column vector, etc. The last two make it easier to construct a matrix from row or column vectors. Collectively, they are used quite a lot in the example program. Perhaps embarrassingly, none of the functions are necessary in Octave - some conversions are done automatically, others are expressed rather cleanly with the M-code syntax.

Another class of conversions is between numeric types: *fromIntegral* is used twice, to convert integer dimension sizes to *Doubles*; again, these are not typically necessary in other numerical languages.

It would be nice if there were a subtyping facility, whereby values could be automatically promoted to members of parent types, to make at least some of these conversions unnecessary. In some cases, such automatic conversions might be undesirable because they could increase the number of bugs by weakening the type system. However, the conversions cited above are between representations which are either strictly or semantically equivalent, so I see no danger in performing them automatically. Yet I imagine such a feature would complicate the type checker considerably.

5.6 Type Checking Difficulties

While the design of our library has made it possible for the type checker to guarantee that programs written with it will not have conformability errors, getting a buggy program to type-check in the first place is much more difficult than we had initially imagined.

As an example, we have left a line commented out in the function *mstep* in appendix A. When this line is swapped for the one above it, so that × appears in the place of ⊗, then GHC gives the following error:

```
LearnBinFactors.hs:132:9:
    Couldn't match the rigid variable ‘d’
        against ‘L s’
      ‘d’ is bound by the type signature for
        ‘doLearn’
      Expected type: State FVector d (L s)
      Inferred type: State FVector d d
```

The line of the error message is in the middle of the last function, *doLearn*, which is quite a bit removed from the actual bug. What happened is that the bug forced the unification of two type variables, *d* and *k*, all over the program. The error message refers to the line which indicates that they should be different, not the place where they were erroneously unified.

One could blame the fact that *mstep* has not been given a type signature. With a type signature for *mstep*, we would be able to indicate that the type variables *d* and *k* are separately universally quantified *in that function*, and that might help us better track down the bug.

Here is a type signature for *mstep*:

```
mstep :: (Dom n, Dom d, Dom k, Vector v R,
  Num (v (n, k)), Num (v (d, k)), Num (v (n, d)),
  FracVector v R) ⇒
  Params v n d → v (n, k) → v (k, k) → State v d k
```

It's quite long! Apparently, polymorphism has a high cost. We can make a version where *v* has been specialized to *FVector*:

```
mstep :: (Dom n, Dom d, Dom k) ⇒
  Params FVector n d → FVector (n, k) →
  FVector (k, k) → State FVector d k
```

However, this is not ideal, because we would like to let our code be typed as generally as possible.

If the language allowed us to omit class membership constraints, it would be very helpful, since such constraints actually have nothing to do with what we're trying to communicate to the type checker, namely that *d* and *k* are different. That would make the first type signature much shorter:

```
mstep :: ... ⇒ Params v n d → v (n, k) → v (k, k) →
  State v d k
```

In any case, when we give a signature to *mstep*, the error becomes:

```
LearnBinFactors.hs:36:0:
    Quantified type variable ‘k’ is unified with
        another quantified type variable ‘d’
    When trying to generalise the type inferred
        for ‘mstep’
```

This is referring to the first line of *mstep*, not *doIters*. It's an improvement, but still not very helpful. There are a lot of operations inside the function *mstep*, and with such a vague error message, the programmer has no recourse but to check each of them manually.

Furthermore, one might wonder why *mstep* needs a single type signature in the first place. We've already made type annotations on the parameters, which use different variables *d* and *k*:

```
mstep (p::Params v n d) (es::v(n,k))
    (ess::v(k,k)) = (s::State v d k)
```

If we had intended for the places occupied by *d* and *k* to always represent the same type, then why would we have put different variable names in them? One might imagine that the compiler should be able to notice this.

But we can go even further. If the compiler knew that the variables were meant to be different at *doIters*, even without the *mstep* type signature, then it should have been able to give us a compile-time error at least as useful as the run-time error which Octave gives us, when we pass objects of different dimensions to *LearnBinFactors*.

Let's try inserting the same bug into the Octave version of the program, in other words changing

```
mu = (ESS \ (ES'*Y))';
```

to

```
mu = (ESS \ (ES'.*Y))';
```

This yields, after 48 seconds, the following error message:

```
error: product: nonconformant arguments
    (op1 is 8x400, op2 is 400x16)
error: evaluating binary operator ‘.*’
    near line 22, column 17
```

Here, Octave gave the exact position of the error, and we didn't even have to add any type signatures. It's true that the error was caught at runtime, not compile-time - and it took 48 seconds for the error to show up. Yet, it took us much longer to track down the error in the Haskell version; and in the case of the Octave version, we spent those 48 seconds reading a book.

Thus, in our opinion, insufficiently helpful compiler errors are currently the greatest impediment to using the library. Conceivably, by following the data flow of the program, GHC could produce an error message which is just as informative as Octave's error message:

```
LearnBinFactors.hs:132:9:
    Couldn't match the rigid variable 'd'
        against 'L s'
      ...
    Probable quantified type mismatch
        arising from use of '*' at
      LearnBinFactors.hs:42:35
```

But that work remains to be done.

It should be noted that we have only used GHC in our tests, and that there are compilers which put much more effort into giving good error messages. However, we depend on many advanced Haskell features - functional dependencies in type classes, GADTs, template haskell - many of which are at present only available in GHC.

This isn't a reason to give up hope. We believe that libraries such as ours, based on strongly typed functional programming languages such as Haskell, are the future of scientific computing. It's just a matter of getting to the point where they are better than the status quo.

Furthermore, we should note that the benefit of using Haskell is expected to be greater for larger programs, where functions are called from multiple contexts, and run-times are longer - in some cases Haskell may already be preferable, in spite of the issues we have mentioned.

6. PERFORMANCE

We found that the Haskell implementation was significantly faster than the Octave implementation. We made some performance improvements to GSLHaskell, most importantly linking to ATLAS and LAPACK and switching the implementation of matrix inversion from GSL to LAPACK (for a factor of 10 speed-up). Octave uses the same or similar libraries, so this is reasonable. We took care to ensure that no operations were hard-coded in one version of the program but not the other.

Our version of ATLAS is the standard Debian package `atlas3-base`, version 3.6.0-20.2, and has not been specially optimized for our system. LAPACK is package `lapack3` version 3.0.20000531a-6, and GSL is package `libgsl0` version 1.8-1. We ran our tests on a 1250 MHz AMD Athlon.

We used GHC version 6.4.2, compiling with `-O3`, and Octave 2.9.5.

The Haskell version of the program takes 545 seconds, while the octave version takes 890 seconds, over 60% longer.

The full library and example programs are available at:

`http://ofb.net/~frederik/stla/`

7. FUTURE WORK

One can imagine a variety of improvements which could be made to our library.

One feature which would be useful is to have a single vector type which can hold arbitrary elements, yet which is as efficient as *FVector* for certain element types such as *Double*. We could require the element type to be *Typeable*, and use *Data.Typeable.cast* to switch to the optimized implementation when possible, using GADTs again as in section 3.1. This could be done as an extension of *FVector*. It would have the advantage of allowing, for instance, an *FVector* matrix to be converted to a vector of row vectors, without having to switch to a vector implementation which can accept other vectors as elements.

We have been thinking about how to expose more information to the type system. To complicate our terminology, one possibility is to make dimensionality in the physics sense part of each numerical type - something like length, time, etc. For instance, it would become illegal to take the logarithm of any value which is not "dimensionless" in the physics sense. We know of one Haskell library (ref Aaron Denney) which accomplishes this. This would appear to solve the problem of distinguishing a matrix from its inverse (since if x is a scalar then $(xA)^{-1} = x^{-1}A^{-1}$) however, it is unclear how physics dimensions could distinguish a (square) matrix from its transpose (which, by the way, is sometimes equal to the inverse). It is an interesting area that we have not looked into.

One would like to make it possible to do fast versions of operations such as *slice* and *margin*. Most of the function arguments to those methods will be very simple - such as rearranging the elements of a tuple or other term:

$$\mathit{trans}\ m \equiv \mathit{slice}\ (\lambda\ (x,\ y) \rightarrow (y,\ x))\ m$$

If we could somehow deconstruct simple functions such as $\lambda\ (x, y) \rightarrow (y, x)$, then we could implement *slice* very efficiently using C helpers. Alternatively, we could require users to specify such functions manually via a special datatype. However, it is unclear how the resulting syntax could be made as clean as the syntax for creating a closure. In any case, a solution to this problem would greatly facilitate the manipulation of more structured quantities such as tensors.

8. CONCLUSION

Our library has shown that strongly typed linear algebra is feasible in Haskell. The implementation is made possible by a number of relatively advanced language features: GADTs, template haskell, functional dependencies, rank-2 polymorphism. While our library is currently less usable than Octave or Matlab, it is already much more efficient than Octave, and we feel that with certain compiler improvements and language features it will become a better programming environment as well.

9. ACKNOWLEDGEMENTS

Thanks to Alberto Ruiz for writing GSLHaskell and answering my questions about it, to Oleg Kiselyov and Chung-chieh Shan for many helpful discussions in the early design of the library, and to Ralf Lammel for patiently explaining his generics library.

10. REFERENCES

D. Botlan and D. Remy. MLF: Raising ML to the power of system-F. *ACM International Conference on Functional Programming. Uppsala, Sweden*, pages 27–38, 2003.

M. Chakravarty, G. Keller, and S. Jones. Associated type synonyms. *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 241–253, 2005.

J. Eaton. GNU Octave: a high-level interactive language for numerical applications. *GNU/Free Software Foundation, Boston*, 1998.

J. Eaton. Octave: Past, present and future. *Proceedings of the 2nd International Workshop on Distributed Statistical Computing*, March 2001.

R. Ennals and S. Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 287–298, 2003.

C. V. Hall, K. Hammond, S. L. P. Jones, and P. L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996. ISSN 0164-0925.

M. P. Jones. Type classes with functional dependencies. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 230–244, London, UK, 2000. Springer-Verlag. ISBN 3-540-67262-1.

S. Jones. Re: splices in type signatures. Message to the Template Haskell mailing list; `http://thread.gmane.org/gmane.comp.lang.haskell.template/321/focus=323`, June 2006.

S. Jones and M. Shields. Practical type inference for arbitrary-rank types. *Submitted to the Journal of Functional Programming*, 2005.

O. Kiselyov and C. Shan. Functional pearl: implicit configurations–or, type classes reflect the values of types. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 33–44, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-850-4.

R. Lämmel and S. Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, 2003.

U. Matlab. The Mathworks Inc. *Natick, MA*, 2003.

C. Okasaki. From fast exponentiation to square matrices: an adventure in types. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 28–35, New York, NY, USA, 1999. ACM Press. ISBN 1-58113-111-9.

D. Rémy. Simple, partial type-inference for System F based on type-containment. *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 130–143, 2005.

A. Ruiz. Matrix computations in haskell based on the gsl. `http://dis.um.es/~alberto/GSLHaskell/matrix.pdf`, June 2005.

T. Sheard and S. Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002.

M. Shields and S. P. Jones. First class modules for Haskell. In *9th International Conference on Foundations of Object-Oriented Languages (FOOL 9), Portland, Oregon*, pages 28–40, Jan. 2002.

APPENDIX

A. EXAMPLE PROGRAM, HASKELL VERSION

```
import Control.Exception
import Debug.Trace
import Random

import Vector

defaults = P{pY = ⊥, pNumFactors = 8,
      pMaxSteps = 30, pIters = 20}

data Params v n d = P{
   pY :: v (n, d),
   pMaxSteps :: Int,
   pIters :: Int,
   pNumFactors :: Int
}
showParams p =
   show (pY p, pMaxSteps p, pIters p, pNumFactors p)

data State v d k = S{
   sMu :: v (d, k),
   sSigma2 :: R,
   sPie :: v k
}
showState s = show (sMu s, sSigma2 s, sPie s)

seqState s = vseq (sMu s). vseq (sPie s).
      seq (sSigma2 s)

but = flip assert

myXlogyx x y = x × log (y / (x +. 1.0e − 20))

mstep (p :: Params v n d) (es :: v (n, k))
      (ess :: v (k, k)) = (s :: State v d k)
    ‘but‘ (sigma2 > 0.0)
where
   s = S{sMu = (mu :: v (d, k)), sSigma2 = sigma2, sPie = pie}
   (y :: v (n, d)) = pY p
   (mu :: v (d, k)) = trans (pinv ess ⋄ (trans es ⋄ y))
 − −           (mu :: v (d, k)) = trans (pinv ess × (trans es ⋄ y))
   sigma2 = (vsumSq y + vsum (mu × (mu ⋄ ess)) −
      2 × vsum (es × (y ⋄ mu))) /
      (fromIntegral $ vlen y)
   pie = fromRow $ unif ⋄ es

meanFieldStep (p :: Params v n d) (s :: State v d k)
      (lambda0 :: (v (n, k))) =
   (lambda :: v (n, k), f :: R, dist :: R)
where
   (mu, sigma2, pie) = (sMu s, sSigma2 s, sPie s)
   y = pY p
   d = fromIntegral $ cols y

   pieExpr = log (pie / (1 − pie))
   lambda = byRow $ λ (p :: n) →
      let y_p = getRow y p in
      vectorUpdate (fromRow $ getRow lambda0 p) domain $
            λ lambda_p′ (i :: k) →
         sigm $ (pieExpr ! i) + (1 / sigma2) ×
         let mu_i = getCol mu i
            mu_ss = vsumSq mu_i / 2
            lambda_p = toRow lambda_p′ in
         fromS ((y_p − lambda_p ⋄ trans mu +
            (lambda_p ! (0, i)) .∗ (trans mu_i)) ⋄ mu_i) − mu_ss
   f = sum $ foreach $ λ p →
      let lp = getRow lambda p
         yp = getRow y p
         f_ = vsum (myXlogyx lp (toRow pie) +
            myXlogyx (1 − lp) (toRow $ 1 − pie))
               − d × log sigma2 / 2
               − (1 / (2 × sigma2)) × vsumSq
                 (yp − lp ⋄ (trans mu))
               − (1 / (2 × sigma2)) × vsum
                 ((lp − lp ∗∗ 2) × (sumCols (mu × mu)))
               − (d / 2) × log (2 × pi)
      in f_
   dist = sqrt $ vsumSq (lambda − lambda0)

initState () = do     -- :: IO (State v d k) = do
   (mu0 :: v (d, k)) ← randIO
   putStrLn $ “size of mu0: ” ++ (show (rows mu0, cols mu0))
   (sigma2_0 :: R) ← randomIO ≫= (return. (+0.1))
   (pie0 :: v k) ← randIO
   return $ S{sMu = mu0, sSigma2 = sigma2_0, sPie = pie0}

learnBinFactors (p :: Params v n d) = do
   s ← initState ()
   doIters p 1 s

doIters p n s | n ⩾ (pIters p) = return s
doIters (p :: Params v n d) (n :: Int) (s :: State v d k) =
   seqState s $ trace (“EM iteration: ” ++ show n) $
   do
   let (mu, sigma2, pie) = (sMu s, sSigma2 s, sPie s)
   (lambda0 :: v (n, k)) ← randIO
   let (lambda :: v (n, k), f) = meanField p s lambda0
   let es = lambda
   let ess = (trans lambda) ⋄ lambda +
            diag (fromRow $ sumCols lambda −
              sumCols (lambda × lambda))
      let s′ = mstep p es ess
      doIters p (n + 1) s′
  meanField (p :: Params v n d) (s :: State v d k)
        (lambda0 :: v (n, k)) =
      let {(lambda, f, n) =
            loopUntil (lambda0, minBound, 0)
              (λ (lambda, f, n) →
        let (lambda′, f′, dist) = meanFieldStep p s lambda
        in
        (if f′ < f then
            trace ("F decreased in MeanField step " ++
              show n ++ " from " ++ show f ++ " to " ++ show f′)
            else id) $
        trace ("f=" ++ show f) $
        trace ("dist=" ++ show dist) $
        ((lambda′, f′, n + 1), n + 1 < (pMaxSteps p))
      in trace ("meanField took " ++ show n ++ " steps") $
      (lambda, f)
  loopUntil :: a → (a → (a, Bool)) → a
  loopUntil x0 f = let (x, cont) = f x0 in
      if cont then loopUntil x f else x
  main = do
      readMatrixFile "data.txt" doLearn
  doLearn :: ∀ d n. (Dom n, Dom d) ⇒ FVector (n, d) → IO ()
  doLearn y = do
      let p = defaults{pY = y}
      putStrLn $ "(n,d)=" ++ show (rows y, cols y)
      reifyIntegral (pNumFactors p) (λ (_ :: k_) → do
        (s :: State v d (L k_)) ←
          learnBinFactors p
        putStrLn $ "mu=" ++ show (sMu s)
        putStrLn $ "pi=" ++ show (sPie s)
        putStrLn $ "sigma2=" ++ show (sSigma2 s)
        )
```

B. EXAMPLE PROGRAM, OCTAVE VERSION

```
function [mu, sigma2, pie] = MStep(Y,ES,ESS)
# Y is NxD
# ES is NxK
# ESS is KxK
# mu is DxK
# sigma2 is 1x1
# pie is 1xK

[N,D] = size(Y);

if (size(ES,1) != N)
  error('ES must have the same number of rows as Y');
endif

K = size(ES,2);

if (!isequal(size(ESS),[K,K]))
  error('ESS must be square and have \
          the same number of columns as ES');
endif

mu = (ESS \ (ES'*Y))';

sigma2 = (sum(sumsq(Y))+sum(sum(mu.*(mu*ESS)))-
  2*sum(sum(ES.*(Y*mu))))/(N*D);

if sigma2 < 0
  error('sigma2 negative in MStep');
endif

pie = mean(ES,1);

endfunction


function [l,F,dist] =
  MeanFieldStep(y,mu,sigma2,pie,lambda0)
# y is N x D
# mu is D x K
# pie is 1 x K
# lambda is N x K
sigm = @(x) 1./(1+exp(-x));
xlogax = @(x,a) x.*log(a./(x+realmin));
n = rows(y);
d = columns(y);
k = columns(mu);
l = lambda0;
F = 0;
for p=(1:n)
  yp = y(p,:);
  for i=(1:k)
    l(p,i) = sigm(log(pie(i)./(1-pie(i))) + \
      (1/sigma2) * \
      (yp-l(p,:)*mu'+l(p,i)*mu(:,i)') * mu(:,i) - \
      sumsq(mu(:,i))/(2*sigma2));
  endfor
  lp = l(p,:);
  f_ = sum(xlogax(lp,pie) + xlogax(1-lp,1-pie)) - \
      d*log(sigma2)/2 - \
      (1/(2*sigma2))*sumsq(y(p,:) - lp*mu') - \
      (1/(2*sigma2))* \
        (sum((lp-lp.^2).*sumsq(mu,1))) - \
      (d/2)*log(2*pi);
  if !isnan(f_)
    F += f_;
  endif
endfor
dist = sqrt(sum(sumsq(l-lambda0)))
endfunction


function [lambda,F] =
    MeanField(Y,mu,sigma2,pie,lambda0,maxsteps)

lambda=lambda0;
F = -realmax;
for step=(1:maxsteps)
  F0 = F
  [lambda,F,dist] =
    MeanFieldStep(Y,mu,sigma2,pie,lambda);
  if F < F0
    fprintf(stderr, "F decreased in \
                MeanField step %d from %f to \
                %f\n", step, F0, F);
  endif
endfor

fprintf(stderr,"MeanField took %d steps\n",maxsteps);

endfunction


function [mu, sigma2, pie] =
    LearnBinFactors(Y,K,iterations,maxsteps)
# Y: data
# K: number of features

N = rows(Y);
D = columns(Y);

# mu is D x K
# pie is 1 x K
# lambda is N x K
```

```
pie = rand(1,K);
mu = rand(D,K);
sigma2 = rand(1,1)+0.1;
lambda0 = rand(N,K);

for iter=(1:iterations)
  lambda0 = rand(N,K);
  fprintf(stderr,"Doing E step\n");
  [lambda, F] = MeanField(Y, mu, sigma2,
                  pie, lambda0, maxsteps);
  # F is lower bound on likelihood
  F
  ES = lambda;
  ESS = lambda'*lambda + \
    diag(sum(lambda,1)-sumsq(lambda,1));

  fprintf(stderr,"Doing M step\n");
  [mu, sigma2, pie] = MStep(Y,ES,ESS);
endfor

endfunction
```