

CamlFloat

A Tutorial

SHIVKUMAR CHANDRASEKARAN

1	Introduction	2
2	Top-level	3
3	Constructing Matrices	4
3.1	Accessing elements	4
3.2	Accessing size information	4
4	Sub-matrices	5
5	Matrix Arithmetic	6
5.1	+, −, ×	6
5.2	Conjugate transpose	6
5.3	Multiplying transposes	6
5.4	Multiply add	7
5.5	Matrix Inversion	7
6	Matrix factorizations	8
6.1	QL and LQ factorizations	8
6.2	Unitary transforms	8
6.3	Forward and backward substitutions	8
6.4	Singular Value Decompositions	9
6.5	Row and Column scaling	9
7	Constructing via sub-matrices	10
7.1	Example: Cholesky factorization	11
8	End notes	12
8.1	Efficiency	12
8.2	Missing pieces	12

1 Introduction

CamlFloat provides a stylized interface to some common Lapack and Blas routines for OCaml. Much of this interface was developed to support my research programming needs for designing and implementing fast matrix algorithms. A similar package is also developed and maintained in parallel for Clean. That package is called CleanFloat.

Further documentation can be found in the directory `Doc` in the distribution and in the file `nla.mli`.

2 Top-level

The easiest way to get familiar with the package is to use the top-level `camlFloat` program. After you start `camlFloat`, you must choose whether you want to work in real single precision, real double precision, complex single precision, or complex double precision arithmetic. Say we want to work in real double precision. We would then say

```
open N1a.DB1op
```

If we wanted to work in complex single precision we would open the module `CB1op` instead.

3 Constructing Matrices

The most basic data type is a **matrix**. There are quite a few ways to create matrices in CamlFloat. The call `rawMatrix m n` returns an $m \times n$ matrix whose elements are uninitialized. So it is very efficient. Another common call is `zeros m n`, which creates an $m \times n$ matrix of zeros.

3.1 Accessing elements

You can access the $(2, 3)$ element of the matrix `a` as `a [2,3]`. Note that all indices are one based just like Matlab or Fortran.

You can also change the value of the $(2, 3)$ element of a matrix `a`. The code `(1.3 => [2,3]) a` sets the $(2, 3)$ entry of `a` to 1.3. This is a rather cumbersome notation. You can define your own infix operator if you wish. But this notation has two purposes. First it emphasizes that this operation should be rarely used; it is inefficient. Second it supports the **storage-passing** style of programming which this package is designed to support.

3.2 Accessing size information

To find out the number of rows that matrix `a` has, you can say `noOfRows a`. Similarly you can use `noOfCols` to get information about the number of columns that a matrix has.

4 Sub-matrices

In mathematical notation we often use block partitioning to specify sub-matrices

$$A = \begin{matrix} & \begin{matrix} n_0 & n_1 \end{matrix} \\ \begin{matrix} m_0 \\ m_1 \end{matrix} & \begin{pmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{pmatrix} \end{matrix}. \quad (4.1)$$

The same can be achieved in CamlFloat as follows

```
let (a00, a01, a10, a11) = partition2x2 n0 n1
                               m0
                               m1   a
```

Unlike Matlab, the sub-matrices will **share** their storage with **a**. It is extremely important to remember this. Unlike Clean, the type system will not help you in this regard.

Sometimes you just need say $A_{1,0}$. You can achieve that as follows:

```
let a10 = a $ (m0+1 <-> m0+m1, 1 <-> n0)
```

Sometimes you want to just do a column partitioning.

$$A = \begin{matrix} & \begin{matrix} n_0 & n_1 \end{matrix} \\ \begin{pmatrix} A_0 & A_1 \end{pmatrix} \end{matrix} \quad (4.2)$$

You can achieve this in CamlFloat as follows:

```
let (a0, a1) = partition1x2 n0 n1 a
```

On the other hand if you just need A_1 , then you can do

```
let a1 = a $ (A11, n0+1 <-> n0+n1)
```

Sometimes you need to break the sharing and make copies. In that case `copy a` will make a unique copy of the matrix **a**.

You can also construct matrices by specifying their sub-matrices. However, we will wait until we have described the matrix arithmetic operators before getting into that.

5 Matrix Arithmetic

5.1 $+$, $-$, \times

To add two matrices `a` and `b` you can do

```
let c = a +$ b
```

The trailing `$` sign signals an arithmetic operator for matrices. Sometimes you already have a matrix (or sub-matrix) `c`, where the sum of `a` and `b` needs to be written. This can be achieved as follows:

```
(a |+| b) c
```

Similarly to do subtraction we have `-$` and `|-|`, and to do multiplication we have `*$` and `|*|`.

5.2 Conjugate transpose

```
let aT = transp a
```

will transpose¹ `a` by copying. If you have pre-allocated storage `aT`, to hold the transpose, you can say instead

```
transp' a aT
```

5.3 Multiplying transposes

Frequently we need to multiply matrices, one of which must be transposed first. It is inefficient to first transpose the matrix. For example to let $C = A^H B$, we would code it in CamlFloat as

```
let c = a *~$ b
```

On the other hand if we wanted to form $C = AB^H$, we would do

```
let c = a *$~ b
```

Note how the position of the `~` matters a great deal.

¹ Conjugate transpose if `a` is a complex matrix.

5.4 Multiply add

If we wanted to modify c as follows $C \leftarrow C - AB$, we would do it as follows:

```
(a |*--| b) c
```

Similarly we have the following calls `|*++|`, `|~*++|`, `|*~++|`, `|~*--|`, and `|*~--|`. Their meanings must be obvious by now.

5.5 Matrix Inversion

To solve a system of equations $Ax = b$, where A can be either square, fat, or skinny, we can do

```
let x = a /$ b
```

If a is square, then x will be the inverse of a times b . If a is fat, then x will be the minimum norm solution, and if a is skinny, x will be the least-squares solution.

We can be more efficient by saying

```
a |/| b
```

in which case both a and b will be destroyed and x will be returned in b . If a is skinny, x will be returned as a sub-matrix of b . If a is fat, then the actual right-hand side must be passed in as the upper sub-matrix of b , and b must be the size of the required x . This is of course just the native Lapack requirements. A bit ugly.

If you want to solve $A^H x = b$ instead then we have the two functions `/~$` and `|~/|`.

6 Matrix factorizations

Two basic types are defined to hold the basic factors returned by Lapack. The first `transform` is for orthogonal and unitary factors, and the second `factor` is for upper and lower triangular factors.

6.1 QL and LQ factorizations

To compute the QL factor of the matrix `a` we have

```
let (q, l) = ql' a
```

The call will destroy `a`. Similarly to compute the LQ factorization of the matrix `a` we have

```
let (l, q) = lq' a
```

In these calls `a` can be any shape. Lapack is difficult to deal with in this situation; for example, QL factorization of a fat matrix, but CamlFloat will keep track of the necessary details.

6.2 Unitary transforms

To apply the `q` orthogonal factor computed by either the QL or LQ factorization to the matrix `b` we have

```
q @* b
```

To instead apply it from the right we have

```
b *@ q
```

In short the relative positions of the `*` and `@` determines on which the side the transform is located. Static typing is a good thing!

If we wish to apply the conjugate transpose of `q` instead we have `@~*` and `*~@`.

6.3 Forward and backward substitutions

To solve the system of equation $Lx = b$, where `l` is an L factor from a QL or LQ factorization we can do

```
subs' l b
```

As a result `b` will be overwritten with `x`. However, `l` must be a square and triangular matrix. Otherwise you will get a run-time error. If we wish to solve $L^H x = b$ instead we would do

```
subsT' l b
```

To convert the `l` factor into a regular dense lower-triangular matrix we can use the functions `copyL` and `iDL`.

6.4 Singular Value Decompositions

You can get the full economy version SVD of the matrix `a` via

```
let (u, s, vt) = svd' a
```

Note the right singular vectors are returned (conjugate) transposed, the way Lapack does. Sometimes you don't need all of the SVD. For that we have `svdL'`, which does not return `vt`, and `svdR'`, which does not return `u`, and also `singValues'`, which only returns the singular values in `s`.

6.5 Row and Column scaling

Since the singular values are returned in a regular `Bigarray.Array1` array, we need routines to multiply matrices by these arrays, which can be thought of as diagonal matrices. The function `rowScale_real'` does row scaling. That is

```
rowScale_real' s vt
```

scales the i -th row of `vt` by `s.{i}`. Similarly `colScale_real'` is available to do column scaling. There are also `rowScale'` and `colScale'` for non-real diagonal matrices.

7 Constructing via sub-matrices

In many instances we need to construct matrices by specifying their sub-matrices. In CamlFloat we can do this pretty much without inducing any extra copying. Suppose we wanted to construct the matrix D from the pre-existing matrices A , B and C as follows

$$D = \begin{matrix} & \begin{matrix} n_0 & n_1 \end{matrix} \\ \begin{matrix} m_0 \\ m_1 \end{matrix} & \left(\begin{array}{cc} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} & 0 \\ C & AB^H \end{array} \right) \end{matrix}. \quad (7.1)$$

We can do this in CamlFloat as follows

```
let d = (zeros (m0+m1) (n0+n1)) in
matrix2x2    n0          n1
  m0 (one => (0,0))    ooo
  m1 (iD c)           (a |*~| b) d
```

First of all the function `matrix2x2` has 9 arguments. But, they are arranged in a logical manner. The first two prescribe the column partitioning. The third and sixth describe the row partitions. The ninth is the storage into which the sub-matrices will be written. The other arguments describe the entries of the corresponding partitions.

The latter must all be functions that take matrices as arguments and modify them as necessary. Such functions have been given the type

```
type transformer = matrix -> unit
```

for convenience.

For example, in the (2, 2) entry we see the familiar function `|*~|` set up to write the product `a*$~b` into the (2, 2) position.

What about the other entries? They are new. The entry in the (2, 1) position uses the function `iD` which takes any matrix as its first argument and writes into the matrix in the second argument.

The entry in the (1, 2) position, `ooo`, is a do nothing function, which is very useful in situations such as this.

The entry in the (1, 1) position explains why I defined `=>` the way I did.

As you can see `matrix2x2` leads to a very elegant way to construct matrices. It is the preferred approach in CamlFloat.

² And its kin `matrix1x2`, `matrix2x1`, etc..

7.1 Example: Cholesky factorization

One can describe Cholesky factorization of a symmetric positive-definite matrix A , very elegantly as follows:

$$A = \begin{matrix} & & 1 & & m-1 \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{matrix} \begin{pmatrix} A_{1,1} & A_{2,1}^T \\ A_{2,1} & A_{2,2} \end{pmatrix}, \quad G_{1,1} = \sqrt{A_{1,1}}, \quad G_{2,1} = \frac{A_{2,1}}{G_{1,1}}. \quad (7.2)$$

Then

$$\text{chol}(A) = G = \begin{matrix} & & 1 & & m-1 \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{matrix} \begin{pmatrix} G_{1,1} & 0 \\ G_{2,1} & \text{chol}(A_{2,2} - G_{2,1}G_{2,1}^T) \end{pmatrix}. \quad (7.3)$$

One can directly code this up using CamlFloat as follows:

```
let chol' a =
  let m = noOfRows a in
  if m >= 1 then begin
    let (a11, _ , a21, a22) = partition2x2 1 (m-1)
                                     1
                                     (m-1) a in
    let g11 = sqrt (a11 $@ (1,1)) in
    a21 <-| scale' (1.0 /. g11);
    let g21 = a21 in
    matrix2x2      1      (m-1)
      1 (g11 => (1,1)) (const 0.0)
    (m-1) ooo      (g21 |*~--| g21) a;
    chol' a22
  end
```

The function `scale'` scales a matrix in place by a given scalar. The code `a <-| b` is just syntax for `b a`. Note that the implementation does no unnecessary copying, and it uses no extra memory, not even on stack as it is tail recursive.

As a contrast observe how nasty a corresponding Matlab implementation will be. It is also interesting to look at the same example provided in the tutorial for Clean-Float.

8 End notes

8.1 Efficiency

As a rule, one should avoid element-wise operations on matrices using such functions as `@`, `=>`, etc.. Rather one should use the matrix algebra operations since these go directly to the corresponding Lapack and Blas calls.

On the other hand, sometimes array manipulation is needed. In such cases OCaml arrays must be manipulated directly. Then these can be converted to CamlFloat matrices using the call `matrix`. This function will take a two-dimensional OCaml array in row-major order and convert it into a CamlFloat matrix.

8.2 Missing pieces

This tutorial does not describe all the functionality of CamlFloat. For that please look at the file `nla.mli` which has comments for all the available routines. The `ocamldoc` version of those comments should be accessible from `Doc/index.html` in the distribution. After reading this tutorial, those comments should make more sense. In case they don't please send me email.

However, there are many missing routines. For example, there are no eigenvalue routines. Please send in your requests for any routines in Lapack or Blas that you need, and I will try to add them as soon as possible. All structured matrix routines, like those for banded and symmetric, will require a new module. So they might take longer than expected to finish.

Please send in bug reports!