CleanFloat

A Tutorial

Shivkumar Chandrasekaran

1 Introduction

CleanFloat provides a stylized interface to some common Lapack and Blas routines for Clean. 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 OCaml. That package is called CamlFloat.

2 Constructing Matrices

The most basic data type is a matrix.

```
:: Matrix e
```

Currently only real and complex (both in double precision) are supported. The necessary complex arithmetic routines are provided in the accompanying complex.icl and complex.dcl modules. Those modules should be self-explanatory.

2.1 Uniqueness

The whole library is critically dependent on utilizing Clean's ability to do destructive updates on unique arrays. The reader should be familiar with uniqueness typing in Clean for those purposes.
One can create a unique matrix with zero entries[1].

```
a = zeros 5 4
```

produces a 5×4 zero matrix that is unique.

2.2 Accessing elements

You can access the $(2, 3)$ element of the matrix `a` as follows:

```
a23 = a @ (2,3)
```

Note that all indices are zero based. This is quite unlike Matlab or Fortran in that regard. However, accessing the element of a unique matrix in this fashion will destroy uniqueness. Hence, like most other functions in Clean where uniqueness plays a role, there is a second way to access the elements of a matrix without destroying uniqueness:

```
#! (a23, a) = a @! (2,3)
```

You can also change the value of the $(2, 3)$ element of a unique matrix `a`. The following code snippet

```
#! a = set a 2 3 -2.63748
```

[1] Real or complex will be determined by Clean if the type can be inferred. Otherwise, the user must provide that information.

will change the $(2, 3)$ entry to -2.63748.

2.3 Accessing size information

To find out the number of rows that matrix `a` has, you can say

```
m = noOfRows a
```

To do it without destroying uniqueness, you will say instead

```
#! (m, a) = noOfRows‘ a
```

Similarly you can use `noOfCols` and `noOfCols‘` to get information about the number of columns that a matrix has.

3 Sub-matrices

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

$$A = \begin{matrix} & \begin{matrix} n_0 & n_1 \end{matrix} \\ \begin{matrix} m_0 \\ m_1 \end{matrix} & \begin{pmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{pmatrix} \end{matrix}. \tag{3.1}$$

The same can be achieved in CleanFloat as follows

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

Note, none of the sub-matrices or `a` will be unique after this. However, unlike Matlab, the sub-matrices will **share** their storage with `a`. It is extremely important to remember this in other languages, but Clean will not let you forget it!
Sometimes you just need say $A_{1,0}$. You can achieve that as follows:

```
a10 = a $ (m0 ... m0+m1-1, 0 ... n0-1)
```

Sometimes you want to just do a column partitioning.

$$A = \begin{matrix} & \begin{matrix} n_0 & n_1 \end{matrix} \\ & \begin{pmatrix} A_0 & A_1 \end{pmatrix} \end{matrix} \tag{3.2}$$

You can achieve this in CleanFloat as follows:

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

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

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

There is one way to **copy** out a sub-matrix, and hence preserve the uniqueness, if any, of the original matrix.

```
#! (a0, a) = a $! (<.>, n0 ... n0+n1-1)
```

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.

4 Matrix Arithmetic

4.1 $+$, $-$, $\times$

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

```
c = a * b
```

as expected. However, `c` will not be unique. This is a built-in limitation of the standard library of Clean.
However, sometimes you already have a unique matrix (or sub-matrix) `c`, where the sum of `a` and `b` needs to be written. This can be achieved as follows:

```
#! c = (a :+: b) c
```

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

4.2 Conjugate transpose

```
aT = transp a
```

will transpose[2] `a` by copying. If you have pre-allocated unique storage `aT`, to hold the transpose, you can say instead

```
#! aT = transp‘ a aT
```

4.3 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 CleanFloat as

```
c = a ~* b
```

Now `c` will be a unique matrix!. On the other hand if we wanted to form $C = AB^H$, we would do

```
c = a *~ b
```

[2] Conjugate transpose if `a` is a complex matrix.

Again `c` would be unique. Note how the position of the `~` matters a great deal.

4.4 Multiply add

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

```
#! c = (a :*--: b) c
```

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

4.5 Matrix Inversion

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

```
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 `a` is skinny, `x` will be the least-squares solution.
If `a` and `b` are unique we can be more efficient by saying

```
#! x = 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 `:~\:`.

5 Matrix factorizations

Two basic types are defined to hold the basic factors returned by Lapack. The first is for orthogonal and unitary factors

```
:: Transform e
```

and the second is for upper and lower triangular factors

```
:: Factor e
```

5.1 QL and LQ factorizations

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

```
#! (q, l) = ql‘ a
```

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

```
#! (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 CleanFloat will keep track of the necessary details.

5.2 Unitary transforms

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

```
#! b = q @* b
```

To instead apply it from the right we have

```
#! b = 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 `*@~`.

5.3 Forward and backward substitutions

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

```
#! x = triSolve l b
```

As a result `b`, which must be unique, will be overwritten with `x`. If we wish to solve $L^H x = b$ instead we would do

```
#! x = triSolveT l b
```

There is some ambiguity when the `l` factor came from the QL factorization of a fat matrix, for example. In such a case CleanFloat will warn you.
To convert the `l` factor into a regular dense lower-triangular matrix we can use the function `copyL`.

5.4 Singular Value Decompositions

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

```
(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`.

5.5 Row and Column scaling

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

```
#! a = s :.*: vt
```

returns `a` in `vt` after scaling the i-th row of `vt` by `a.[i]`. Similarly `:*.` is available to do column scaling.

6 Constructing via sub-matrices

In many instances we need to construct matrices by specifying their sub-matrices. In CleanFloat 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

$$D = \begin{array}{c} \\ m_0 \\ m_1 \end{array} \begin{array}{c} \begin{array}{cc} n_0 & \quad \end{array} \\ \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \quad 0 \\ C \qquad AB^H \end{array} \begin{array}{c} \\ \\ \end{array} \begin{array}{c} \\ \\ \end{array} \begin{array}{c} n_1 \\ \\ \end{array} \tag{6.1}$$

We can do this in CleanFloat as follows

```
d = matrix2x2    n0                n1
           m0 ((0,0) <=. one)        ooo
           m1 (iD c)           (a :*~: b)  (zeros (m0+m1) (n0+n1))
```

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 unique 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 unique matrices as arguments and return them as their results. Such functions have been given the type

```
:: Transformer e :== *(Matrix e) -> *(Matrix e)
```

for convenience.
For example, in the $(1,1)$ entry we see the familiar function `:*:` set up to wrote the product `a*b` into the $(1,1)$ position.
What about the other entries? They are new. The entry in the $(1,0)$ position uses the function `iD` which takes any matrix as its first argument and writes into the matrix in the second argument, which must be necessarily unique.
The entry in the $(0,1)$ position, `ooo`, is a do nothing function!
The entry in the $(0,0)$ position has the following semantics:

```
#! a = ((i,j) <=. x) a
```

is exactly the same as `set a i j x`.

As you can see `matrix2x2`[3] leads to a very elegant way to construct matrices. It is the preferred approach in CleanFloat.

6.1 Example: Cholesky factorization

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

$$A = \begin{matrix} \\ 1 \\ m-1 \end{matrix} \begin{matrix} \begin{matrix} 1 & m-1 \end{matrix} \\ \begin{pmatrix} A_{0,0} & A_{1,0}^T \\ A_{1,0} & A_{1,1} \end{pmatrix} \end{matrix}, \qquad G_{0,0} = \sqrt{A_{0,0}}, \qquad G_{0,1} = \frac{A_{1,0}}{G_{0,0}}. \tag{6.2}$$

Then

$$\mathrm{chol}(A) = G = \begin{matrix} \\ 1 \\ m-1 \end{matrix} \begin{matrix} \begin{matrix} 1 & m-1 \end{matrix} \\ \begin{pmatrix} G_{0,0} & 0 \\ G_{1,0} & \mathrm{chol}(A_{1,1} - G_{1,0}G_{1,0}^T) \end{pmatrix} \end{matrix}. \tag{6.3}$$

One can directly code this up using CleanFloat as follows:

```
chol‘ a
    #! (m, a)   = noOfRows‘ a
    |  m < 1    = a
    #! (a00, a) = a @! (0,0)
    #! g00      = sqrt a00
    |  m == 1   = ((0,0) <=. g00) a
    #! (a0, a)  = a $! (1 ... m-1, 0 ... 0)
    #! g0       = scale‘ (1.0 / g00) a0
    = matrix2x2        1                 (m-1)
              1  ((0,0) <=. g00)             (scale‘ 0.0)
         (m-1) (iD g0)          (chol‘ o (g0 :*~--: g0))   a
```

The function `scale‘` scales a unique matrix by a given scalar. After

```
#! a = scale‘ x a
```

the (i, j)-th entry of `a` is `x` times the old (i, j)-th entry.

[3] And its kin `matrix1x2`, `matrix2x1`, etc..

7 End notes

7.1 Efficiency

As a rule, one should avoid element-wise operations on matrices using such functions as `@`, `set`, 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 Clean arrays must be manipulated directly. Then these can be converted to CleanFloat matrices using the call `toMatrix`. This function will take a one-dimensional Clean array and convert it into a CleanFloat matrix, assuming that the one-dimensional array is in Fortran-style **column-major** order! The utility function `to2D` is provided to convert a two-dimensional Clean array in row-major order into a one-dimensional Clean array in column-major order.

7.2 Missing pieces

This tutorial does not describe all the functionality of CleanFloat. For that please look at the file blop.dcl which has comments for all the available routines. 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 you 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!