DSC 2003 Working Papers
(Draft Versions)
http://www.ci.tuwien.ac.at/Conferences/DSC-2003/

Practical uses of external references

VJ Carey
Harvard University
stvjc@channing.harvard.edu

1 Introduction

Facilities for interactive computing with pointers have recently been introduced to R (see http://www.stat.uiowa.edu/~luke/R/regexp.html and http://www.stat.uiowa.edu/~luke/R/simpleref.html). Two implications of the development are

- more computations can be moved outside the `.C/.Call` boundary and into R, and
- interactive computing becomes feasible for larger data objects.

There are significant development and safety costs incurred when programming with pointers. This paper reviews examples of pointer use in R in the domains of matrix computations and fitting GLMs.

2 Pointers to matrices

The *Cmat* package (available at the repository under http://www.biostat.harvard.edu/~carey) defines classes and methods for manipulating pointers to C language `structs` that represent matrices. The basic C `struct` is

```
typedef struct matrix
 {
  int Cmat_nrows, Cmat_ncols;
  double *data;
  int permanence;
```

```
 } MATRIX;
```

C routines for manipulating this structure have been available in the `gee` package for many years. In that package, numerical data and modeling options are gathered from the modeling interface function, using a `formula`. Data and model specifications are passed to C via the `.C` interface. All the underlying algebra and iteration is performed in C. The *Cmat* package is designed to permit the expression of the algebra and iteration in R, without sacrificing performance relative to a pure C implementation.

2.1 *Cmat* basics

The *Cmat* package uses formal methods and classes.

```
> library(Cmat)
```

```
Loading required package: methods
```

First, we use the `Cmat` function to construct a `Cmat` instance (here pointing to a diagonal matrix).

```
> CX <- Cmat(diag(4))
> show(CX)
```

```
Cmat object CX
 dimensions:
  4 4
 persistence: TRUE
```

Note the setting of the persistence flag. This flag is modified using the `setPersis` method.

```
> setPersis(CX, FALSE)
> show(CX)
```

```
Cmat object CX
 dimensions:
  4 4
 persistence: FALSE
```

Coercion back to an R matrix is supported. The default behavior of the coercion function (indeed all *Cmat* functions and methods) is to reclaim the memory used in C unless the persistence flag is set. Thus the second coercion of `CX` fails.

```
> print(as.matrix(CX))
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

```
> print(as.matrix(CX))

<0 x 0 matrix>
```

Understanding and using persistence effectively is a basic challenge in using *Cmat*. The problem arises from dealing with anonymous intermediate computations. For example, computing a matrix product ABC it is desirable to discard the resources used for the intermediate factor AB. This is most easily accomplished if the multiplication operator destroys its inputs after use. But some inputs need to be reused, and this can be achieved by setting the persistence property on such matrices.

To avoid coding explicit calls to `setPersis`, a special assignment operator `%<-%` sets persistence of the assignee.

```
> CX %<-% Cmat(diag(4))
> show(CX)

Cmat object CX
 dimensions:
  4 4
 persistence: TRUE
```

Arithmetic and linear algebra can be performed with `Cmat` instances following S conventions. Thus

```
> Z <- Cmat(matrix(1:4, nc = 1))
> W %<-% (2 * CX - CX/Z)
> show(W)

Cmat object W
 dimensions:
  4 4
 persistence: TRUE

> print(as.matrix(W))

     [,1] [,2]     [,3] [,4]
[1,]    1  0.0 0.000000 0.00
[2,]    0  1.5 0.000000 0.00
[3,]    0  0.0 1.666667 0.00
[4,]    0  0.0 0.000000 1.75
```

When a binary operator combines a `Cmat` instance with a non-`Cmat`, the non-`Cmat` will be replicated and coerced to conformity according to the usual rules of S. Warnings are emitted if non-integral replication occurs.

The QR decomposition is supported, with computations performed by LAPACK (now included in R if a configuration switch is set).

```
> data(stackloss)
> QR.SX <- CmatQR(cbind(1, stack.x))
> show(QR.SX)
```

```
An object of class "CmatQR"
Slot "ref":
<pointer: 01529440>
```

Coercion to R data is straightforward:

```
> print(as.qr(QR.SX))

$qr
            [,1]          [,2]         [,3]          [,4]
 [1,] -4.5825757 -276.91850271 -96.67052537 -3.954108e+02
 [2,]  0.2182179  -41.00174212 -11.05179521 -1.198555e+01
 [3,]  0.2182179    0.26988176   8.81290795 -3.779760e-03
 [4,]  0.2182179   -0.04717794  -0.27073499 -2.075168e+01
 [5,]  0.2182179   -0.04717794  -0.04379514  3.395087e-02
 [6,]  0.2182179   -0.04717794  -0.15726506  3.751722e-02
 [7,]  0.2182179   -0.04717794  -0.27073499  3.302168e-01
 [8,]  0.2182179   -0.04717794  -0.27073499  3.302168e-01
 [9,]  0.2182179   -0.14473477  -0.28371496  8.612300e-02
[10,]  0.2182179   -0.14473477   0.28363466 -2.690308e-01
[11,]  0.2182179   -0.14473477   0.28363466  1.646690e-01
[12,]  0.2182179   -0.14473477   0.39710458  1.129138e-01
[13,]  0.2182179   -0.14473477   0.28363466 -1.726531e-01
[14,]  0.2182179   -0.14473477   0.17016474  3.609908e-01
[15,]  0.2182179   -0.33984843   0.03073486  2.618805e-01
[16,]  0.2182179   -0.33984843   0.03073486  1.173139e-01
[17,]  0.2182179   -0.33984843  -0.08273507 -5.537639e-01
[18,]  0.2182179   -0.33984843  -0.08273507 -2.164418e-01
[19,]  0.2182179   -0.33984843  -0.19620499 -1.646866e-01
[20,]  0.2182179   -0.19351318  -0.00653014 -1.412175e-01
[21,]  0.2182179    0.14793572   0.43604452  1.223621e-01

$rank
[1] 4

$qraux
[1] 1.218218 1.391828 1.026757 1.041084

$pivot
[1] 1 2 3 4
```

Compare to the built-in:

```
> print(qr(cbind(1, stack.x)))

$qr
                  Air.Flow   Water.Temp    Acid.Conc.
```

```
1  -4.5825757 -276.91850271 -96.67052537 -3.954108e+02
2   0.2182179  -41.00174212 -11.05179521 -1.198555e+01
3   0.2182179    0.26988176   8.81290795 -3.779760e-03
4   0.2182179   -0.04717794  -0.27073499 -2.075168e+01
5   0.2182179   -0.04717794  -0.04379514  3.395087e-02
6   0.2182179   -0.04717794  -0.15726506  3.751722e-02
7   0.2182179   -0.04717794  -0.27073499  3.302168e-01
8   0.2182179   -0.04717794  -0.27073499  3.302168e-01
9   0.2182179   -0.14473477  -0.28371496  8.612300e-02
10  0.2182179   -0.14473477   0.28363466 -2.690308e-01
11  0.2182179   -0.14473477   0.28363466  1.646690e-01
12  0.2182179   -0.14473477   0.39710458  1.129138e-01
13  0.2182179   -0.14473477   0.28363466 -1.726531e-01
14  0.2182179   -0.14473477   0.17016474  3.609908e-01
15  0.2182179   -0.33984843   0.03073486  2.618805e-01
16  0.2182179   -0.33984843   0.03073486  1.173139e-01
17  0.2182179   -0.33984843  -0.08273507 -5.537639e-01
18  0.2182179   -0.33984843  -0.08273507 -2.164418e-01
19  0.2182179   -0.33984843  -0.19620499 -1.646866e-01
20  0.2182179   -0.19351318  -0.00653014 -1.412175e-01
21  0.2182179    0.14793572   0.43604452  1.223621e-01

$rank
[1] 4

$qraux
[1] 1.218218 1.391828 1.026757 1.041084

$pivot
[1] 1 2 3 4
```

Many functionalities available for R matrices are missing at this time. Chief among them is a concept of subscripting.

2.2 `apply` analogs

Elementwise function application over matrices is currently supported in two ways.

Arbitrary R function application. Any 1 to 1 R function can be applied using the method `ewApplySLOW`. The name emphasizes the fact that the implementation is very slow owing to the need to install a symbol for every matrix element to allow its use as a function argument.

```
> print(as.matrix(ewApplySLOW(W, exp))[1:2, ])

         [,1]     [,2] [,3] [,4]
[1,] 2.718282 1.000000    1    1
[2,] 1.000000 4.481689    1    1
```

To achieve better performance, a family of function pointers is supplied with *Cmat*. Some examples are

```
> ol <- objects(pos = "package:Cmat", all = TRUE)
> print(ll <- ol[grep("^.L", ol)])

[1] ".LOG"   ".LOGIT"

> show(get(ll[1]))

<pointer: 112C40B0>
```

The object `.LOG` is a pointer to a C function that evaluates the natural log on the target of a pointer to a double. The `funcptr` class adds some information. This particular function pointer gets used in a GLM family object:

```
> show(Cpoisson()$linkfun)

function pointer, 'log'
```

and the inverse is also on hand:

```
> show(Cpoisson()$linkinv)

function pointer, 'exp'
```

More efficient elementwise application uses these function pointers:

```
> print(as.matrix(ewApply(W, Cpoisson()$linkinv))[1:2, ])

         [,1]     [,2] [,3] [,4]
[1,] 2.718282 1.000000    1    1
[2,] 1.000000 4.481689    1    1
```

2.3 Performance assessment: GLM

A basic question concerns the speed and expressiveness of R programs enhanced with *Cmat*. To explore this, a partial implementation of GLM is supplied as `glmDemo` with *Cmat*. For a series of independent responses indexed by i, let $\mu_i(\beta) = EY_i$, and $V_i(\mu) = \text{Var } Y_i$. The algorithm used in R to fit a GLM solves the equation

$$\sum_i \frac{\partial \mu_i(\beta)}{\partial \beta}^t V_i^{-1}(\mu)[Y_i - \mu_i(\beta)] = 0$$

by iteratively reweighted least squares (IRWLS); see McCullagh and Nelder (1989) for formal details. We wish to preserve as much of the relevant informal notation as possible, for code comprehensibility.

The `glmDemo` function takes arguments `X` and `y` which are to be `Cmat` instances embodying design and outcome, and a `family` object, which in this case is a function that emulates the standard R `family` structure using `funcptr` instances. The most fully developed at time of writing is `Cpoisson`:

```
> show(Cpoisson)

function (link = "log")
{
    if (link != "log")
        stop("pointer family currently requires canonical link")
    linkf <- new("funcptr", ref = .LOG, desc = "log")
    ilinkf <- new("funcptr", ref = .EXP, desc = "exp")
    mu.eta <- new("funcptr", ref = .EXP, desc = "exp")
    varf <- new("funcptr", ref = .ID, desc = "id")
    dev.resids <- new("funcptr", ref = .poiDR, desc = "poiDevRes")
    list(linkfun = linkf, linkinv = ilinkf, variance = varf,
        mu.eta = mu.eta, dev.resids = dev.resids)
}
```

`glmDemo` contains the following code to obtain a function `mu` that can be evaluated on a `Cmat` embodying the parameter vector β:

```
muMk <- function(x) function(beta)
   ewApply( x %*% beta, family()$linkinv )
mu <- muMk(X)
```

The adjusted dependent variable of the IRWLS scheme is obtained via

```
eta <- X%*%beta
mu.eta <- ewApply( eta, family()$mu.eta )
z <- X%*%beta + (Y - mu(beta))/mu.eta
```

Very similar code for `z` is present in `glm.fit`. The least squares problems are solved using QR decompositions.

Performance for poisson fitting is assessed via simulation.

```
> print(memory.size())

[1] 12272488

> N <- 75240
> P <- 8
> Y <- rpois(N, P)
> X <- matrix(rnorm(N * P), nc = P)
> am <- function(x) matrix(x, nc = 1)
> CY %<-% Cmat(am(Y))
> CX %<-% Cmat(cbind(1, X))
> binit <- rep(0.01, 9)
> print("timing for glm.fit")

[1] "timing for glm.fit"

> print(gu <- unix.time(g1 <- glm(Y ~ X, fam = poisson, start = binit)))
```

```
[1] 39.11  0.49 75.26    NA    NA

> print(memory.size())

[1] 73782560

> cat("timing for Cmat demo version")

timing for Cmat demo version

> print(cu <- unix.time(cg1 <- glmDemo(CX, CY, fam = Cpoisson,
+     binit = binit)))

[1] 12.96  0.22 16.65    NA    NA

> print(memory.size())

[1] 53962616

> print(g1$coef - cg1$coef)

  (Intercept)            X1            X2            X3            X4
 2.076161e-10 -6.698303e-11 -8.182598e-11 -8.026366e-11 -7.466743e-11
           X5            X6            X7            X8
-7.729480e-11 -7.116448e-11 -6.339779e-11 -7.605555e-11
```

Speedup factor by using references is estimated at 4.52. There is an indication of diminished memory consumption in the pointer-based approach. Speedup estimates based on the current implementation are biased upwards as there are a number of additional calculations (e.g., of deviance) needed to fully emulate `glm.fit`. Of course the specific estimate provided here (computed for convenience in *Sweave*) is very approximate.