APL in R

Jan de Leeuw
Masanao Yajima

Version 009, March 07, 2016

Abstract

R versions of the array manipulation functions of APL are presented. We do not translate the system functions or other parts of the runtime. Also, the current version has does not have the nested arrays of APL-2.

Contents

Note: This is a working paper which will be expanded/updated frequently. All suggestions for improvement are welcome. The directory gifi.stat.ucla.edu/apl has a pdf version, the complete Rmd file with all code chunks, the bib file, and the R, C, and C++ source code.

1 Introduction

APL was introduced by Iverson (1962). It is an array language, with many functions to manipulate multidimensional arrays. R also has multidimensional arrays, but not as many functions to work with them.

In R there are no scalars, there are vectors of length one. For a vector `x` in R we have `dim(x)` equal to `NULL` and `length(x) > 0`. For an array, including a matrix, we have `length(dim(x)) > 0`. APL is an array language, which means everything is an array. For each array both the shape ρA and the rank ρρA are defined. Scalars are arrays with shape equal to one, vectors are arrays with rank equal to one.

If you want to evaluate APL expressions using a traditional APL virtual keyboard, we recommend the nice webpage at ngn.github.io/apl/web/index.html. EliStudio at fastarray.appspot.com/default.html is essentially an APL interpreter running in a Qt GUI, using ascii symbols and symbol-pairs to replace traditional APL symbols (Chen and Ching (2013)). Eli does not have nested arrays. It does have ecc, which compiles eli to C.

In 1994 one of us coded most APL array operations in `XLISP-STAT`. The code is still available at gifi.stat.ucla.edu/apl.

There are some important differences between the R and Lisp versions, because Lisp and APL both have C's row-major ordering, while R (like `Matlab`) has `Fortran`'s column-major ordering in the array layout. Our R version of APL uses column-major ordering. By slightly changing the two basic building blocks of our code, the `aplDecode()` and `aplEncode()` functions, it would be easy to choose between row-major and column-major layouts. But this would make it more complicated to use the code with the rest of R.

Because of layout, the two arrays 3 3 3ρι27 and `array(1:27,rep(3,3))` are different. But what is really helpful in linking the two environments is that ,3 3 3ρι27 and `as.vector(array(1:27,rep(3,3))`, which both ravel the array to a vector, give the same result, the vector ι27 or `1:27`. This is, of course, because ravelling an array is the inverse of reshaping a vector.

Most of the functions in R are written with arrays of numbers in mind. Most of them will work for array with elements of type logical, and quite a few of them will also work for arrays of type character. We have to keep in mind, however, that APL and R treat character arrays quite differently. In R we have `length("aa")` equal to 1, because `"aa"` is a vector with as its single element the string `"aa"`. R has no primitive character type, characters are just strings which happen to have only one character in them. In APL strings themselves are vectors of characters, and ρaa is 2. In R we can say `a<-array("aa",c(2,2,2))`, but in APL this gives a domain error. In APL we can say 2 2 2ρ"aa", which gives the same result as 2 2 2ρ"a" or 2 2 2ρ'a'.

In this version of the code we have not implemented the nested arrays of `APL-2`. Nesting gives every array `A` not just a shape ρA and a rank ρρA, but also a depth. The depth of an array of numbers or characters is one, the depth of a nested array is the maximum depth of its elements.

There are many dialects of APL, and quite a few languages derived from APL, such as `A+` and `J`. As a standard for `APL-I` we use Helzer (1989).

2 Core

The basic engine behind the APL array manipulation functions is the pair `aplDecode()` and `aplEncode()`. They make it easy to go back and forth between multidimensional arrays and the one-dimensional vectors that store their elements.

Suppose a is the array

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]   19   21   23
## [2,]   20   22   24
```

with shape equal to

```
## [1] 2 3 4
```

and rank

```
## [1] 3
```

Of course the length of `a` is the product of the elements of its shape. If `r` is an integer between 1 and `length(a)` then `aplEncode(r,aplShape(a))` returns the index vector `k` such that element with indices `k` of `a` is `a[r]`.

```r
aplSelect(a, aplEncode (1, aplShape(a)), drop = TRUE)
```

```
## [1] 1
```

```r
aplSelect(a, aplEncode (10, aplShape(a)), drop = TRUE)
```

```
## [1] 10
```

```r
aplSelect(a, aplEncode (24, aplShape(a)), drop = TRUE)
```

```
## [1] 24
```

If `k` is an admissible index vector then `aplDecode(k,aplShape(a))` returns an integer such that element with indices `k` of `a` is `a[aplDecode(k,dims)]`

```r
a[aplDecode (c(1,1,1), aplShape(a))]
```

```
## [1] 1
```

```r
a[aplDecode (c(2,2,2), aplShape(a))]
```

```
## [1] 10
```

```r
a[aplDecode (c(2,3,4), aplShape(a))]
```

```
## [1] 24
```

Note that `aplDecode()` and `aplEncode()` are inverses of each other because

```r
r <- 2
r == aplDecode(aplEncode(r, aplShape(a)), aplShape(a))
```

```
## [1] TRUE
```

```r
k <- c(1,2,3)
all(k == aplEncode(aplDecode(k, aplShape(a)), aplShape(a)))
```

```
## [1] TRUE
```

and this is true for all admissible `r` and `k`.

3 Functions

3.1 Compress

Compress (IBM (1988) , p. 91—92) is defined as the Replicate operator L/R in the special case that L is binary. So look under Replicate for the definition.

3.2 Decode

The decode dyadic operator L⊥R is also known as base value (Helzer (1989), p. 17-21). If L is scalar and R is a vector, then L⊥R is the polynomial $r_1x^{m-1}+r_2x^{m-2}+\cdots+r_m$ evaluated at L. This means that if the r_i are nonnegative integers less than L, then L⊥R gives the base-10 equivalent of the base-L number R.

Normally, however, and in our R implementation, the arguments L and R are vectors of the same length. This is also because for scalar L the expression L⊥R is expanded to ((ρR)ρL)⊥R If L and R are vectors of the same length then decode returns the index of element `A[R]` in an array `A` with `dim(A)=L`.

Obviously the R implementation, which uses colum-major ordering, will give results different from the APL implementation. In APL the expression 3 3 3⊥1 2 3 evaluates to 18, while `aplDecode(1:3, rep(3,3))` gives 22. Also note the order of the arguments is interchanged. Also note that if x and y are of length m, and y has all elements equal to z, then `aplDecode(x, rep(y, length(x)))` is the value of the polynomial

$$p(u)=1+(x_1-1)u^0+(x_2-1)u^1+\cdots+(x_m-1)u^{m-1}$$

evaluated at $u=z$.

```
for (k in 1:3) for (j in 1:3) for (i in 1:3)
print (aplDecode (c(i,j,k), c(3,3,3)))
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
## [1] 21
## [1] 22
## [1] 23
## [1] 24
## [1] 25
## [1] 26
## [1] 27
```

3.3 Drop

See Helzer (1989), p. 49—51. If L is a positive integer and R is a vector then L↓R drops the first L elements. If L is a negative integer the last L elements are dropped. If R is an array, then L must have ρρR elements. Depending on whether the elements or L are positive or negative, the L first or last slices of the dimension are dropped.

Our R implementation again interchanges the two arguments. Note that in R the default on subsetting arrays is `drop = TRUE`. In our implementations the default is always `drop = FALSE`. Note that more general subsetting can be done with `aplSelect()`.

So for a vector

```
aplDrop(1:10,3)
```

```
## [1]  4  5  6  7  8  9 10
```

```
aplDrop(1:10,-3)
```

```
## [1] 1 2 3 4 5 6 7
```

and for an array

```
aplDrop(a,c(1,0,1), drop = TRUE)
```

```
##      [,1] [,2] [,3]
## [1,]    8   14   20
## [2,]   10   16   22
## [3,]   12   18   24
```

```
aplDrop(a,c(-1,-1,0), drop = TRUE)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    7   13   19
## [2,]    3    9   15   21
```

3.4 Encode

Encode, also known as representation is the inverse of decode (Helzer (1989), p. 17—21). LTR takes a radix vector L and a number R and returns the array indices corresponding to cell L in an array with shape ρA.

```
for (i in 1:27)
print (aplEncode (i, c(3,3,3)))
```

```
## [1] 1 1 1
## [1] 2 1 1
## [1] 3 1 1
## [1] 1 2 1
## [1] 2 2 1
## [1] 3 2 1
## [1] 1 3 1
## [1] 2 3 1
## [1] 3 3 1
## [1] 1 1 2
## [1] 2 1 2
## [1] 3 1 2
## [1] 1 2 2
## [1] 2 2 2
## [1] 3 2 2
## [1] 1 3 2
## [1] 2 3 2
## [1] 3 3 2
## [1] 1 1 3
## [1] 2 1 3
## [1] 3 1 3
## [1] 1 2 3
## [1] 2 2 3
## [1] 3 2 3
## [1] 1 3 3
## [1] 2 3 3
## [1] 3 3 3
```

3.5 Expand

Expand (Helzer (1989), p. 64—66) L\R replaces slices of a vector or an array by zeroes. In APL we use a boolean vector for L, and an array for R. If R is a vector then the number of elements of L equal to one (i.e. `TRUE`) should be equal to the length of R. Then L\R produce a vector of length ρL with the elements of R in the places where L is one, and zeroes elsewhere. In our function `aplExpand()` we again reverse the order of the arguments. The second argument in the R verson can be both logical or binary.

```r
aplExpand(1:3, c(1,0,0,0,1,1))
```

```
## [1] 1 0 0 0 2 3
```

For an array expand takes an optional axis argument. For a matrix, for example, axis=1 expands rows (i.e. inserts row of zeroes at specfied places), while axis=2 expands columns. This generalizes to multidimensional arrays, where there are just more axis to consider, but any one of them can be expanded.

```r
aplExpand(matrix(1,2,3), c(1,0,0,1), axis = 1)
```

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

```r
aplExpand(matrix(1,2,3), c(1,1,0,1,0), axis = 2)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    1    0    1    0
## [2,]    1    1    0    1    0
```

3.6 Get

There is no APL function corresponding to get. We wrote `aplGet()` as a simple utility to get an element out of an array,

```
a <- array(1:24, c(2,3,4))
aplGet (a, c(2,2,2))
```

```
## [1] 10
```

```
aplGet (a, aplEncode (14, c(2,3,4)))
```

```
## [1] 14
```

3.7 Inner Product

APL has a generalized inner product (Helzer (1989), p. 100-103), which we write as Lf.gR, where f and g are any scalar dyadic functions. In obvious notation the f, g inner product of two vector x and y of length n is $f(g(x_1, y_1), g(x_2, y_2), \cdots, g(x_n, y_n))$. If L and R are arrays, then the last item of ρL must be equal to the first item of ρR and the generalized inner product operates along that common dimension. The default for g is multiplicaton, and for f is addition, leading to the ordinary inner product and to matrix multiplication.

Of course there are many examples we can give.

```
x <- matrix (1:12,4,3)
y <- matrix (1:12,3,4)
aplInnerProduct (x, y)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   38   83  128  173
## [2,]   44   98  152  206
## [3,]   50  113  176  239
## [4,]   56  128  200  272
```

```
h <- function (x, y) ifelse (x == y, 1, 0)
aplInnerProduct (x, y, h, "+")
```

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

```
a <- array (1:24, c(2,3,4))
b <- rep (1, 4)
aplInnerProduct (a, b)
```

```
##      [,1] [,2] [,3]
## [1,]   40   48   56
## [2,]   44   52   60
```

3.8 Join

Join catenates two arrays of the same rank, creating another larger array of that rank (Helzer (1989), p. 104—110). If L and R are vectors L,R simply concatenates, same as c() in R. If L and R are arrays they can be catenated along axis k if (ρL)[-k] = (ρR)[-k]. Thus matrices with the same number of columns can be stacked on top of each other, and matrices with the same number of rows can be stacked next to each other. Same for higher-dimensional arrays.

In APL there are some special rules for handling scalar arguments (they get reshaped accordingly first), and even for fractional axis parameters, which can be used for lamination of arrays with different rank. We have not implemented these more complicated optiosn in our function aplJoin().

```
x<-1:3
y<-3:1
aplJoin(x,y)
```

```
## [1] 1 2 3 3 2 1
```

```
x <- matrix (1:12, 3, 4)
y <- matrix (1:8, 2, 4)
aplJoin (x, y, axis = 1)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
## [4,]    1    3    5    7
## [5,]    2    4    6    8
```

```
a <- array (1:24, c(2, 3, 4))
b <- array (1:30, c(2, 3, 5))
dim(aplJoin(a, b, axis = 3))
```

```
## [1] 2 3 9
```

3.9 Member Of

The member of function in APL L∈R returns a binary array with the shape of L. Array R can be of any shape. If an element of L occurs in R then the corresponding element in L∈R is one, otherwise it is zero. Our function `aplMemberOf()` returns an array or vector of the same dimension or length as the first argument. The test for equality is a simple ==, so integers and doubles can be compared.

```
a <- array (1:24, c(2,3,4))
aplMemberOf(a, c(1,2,15,25))
```

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

3.10 Outer Product

The APL outer product L ∘.f R is the same as the outer product in R (Helzer (1989), p. 147—149). Both L and R can be arbitrary arrays, and the result is an array with shape (ρL),ρR, formed by applying the dyadic function f to each combination of elements of L and R. Thus our `aplOuterProduct()` just calls the R function `outer()` on its arguments.

```r
x <- matrix (1:4, 2,2)
y <- matrix (1:4, 2,2)
aplOuterProduct (x, y, "+")
```

```
## , , 1, 1
##
##      [,1] [,2]
## [1,]    2    4
## [2,]    3    5
##
## , , 2, 1
##
##      [,1] [,2]
## [1,]    3    5
## [2,]    4    6
##
## , , 1, 2
##
##      [,1] [,2]
## [1,]    4    6
## [2,]    5    7
##
## , , 2, 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
```

3.11 Ravel

The APL function ravel reshapes its argument into a vector. So ,R is identical to `as.vector()` in R, and `aplRavel()` is just a call to `as.vector()`.

```r
aplRavel(array(1:24, c(2,3,4)))
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24
```

3.12 Rank

There is no APL function rank, but we just implemented `aplRank()` as a convenient shorthand for ρρR. In R we just call `length(dim())`.

```r
aplRank(array(1:24, c(2,3, 4)))
```

```
## [1] 3
```

3.13 Reduce

It is time for a quote from Helzer (1989), p. 165. “Summing a list of numbers is a common programming task. In APL this, and much more, is accomplished using the reduction operator. In APL terminology an operator modifies the action of a function. Standard APL has four operators, inner product (f.g), outer product (∘.f), reduction (f/), and scan (f\). Inner product creates a new dyadic function out of two scalar dyadic functions f and g. Outer product creates a new dyadic functionout of a scalar dyadic function f. Reduction and scan both create a new monadic function out of a scalar dyadic function f.” In APL there are two operators ⌿, which reduces along the first axis and /, which reduces along the last axis. Reducing along the k-th axis is f/[k].

Our R version aplReduce() of reduce takes three arguments: the array, the axes to reduce over, and the dyadic function used for reduction (by default addition). Note that the second argument can be a vector with more than one axis, in which case we reduce over all those axes.

```r
a <- array(1:24, c(2,3,4))
aplReduce (a, c(1,2), max)
```

```
## [1]  6 12 18 24
```

```r
aplReduce (a, 1, function (x, y) ifelse (x > y, x, y))
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    8   14   20
## [2,]    4   10   16   22
## [3,]    6   12   18   24
```

3.14 Replicate

If L and R in replicate L/R are vectors of the same length, then the result is a vector of length +/L in which element $r[i]$ is repeated $l[i]$ times. If L is binary then some elements will be deleted, and replicate is called compress.

```r
aplReplicate(1:3,c(3,1,3))
```

```
## [1] 1 1 1 2 3 3 3
```

```r
aplReplicate(1:10,rep(c(0,1),5))
```

```
## [1]  2  4  6  8 10
```

This can be extended to higher dimensional arrays in the usual way by specifying the axis along which to replicate. By default, as in APL, we select the last axis. In APL L⌿R is used to replicate along the first axis. Our function `aplReplicate()` covers both cases.

```r
a <- array(1:24,c(2,3,4))
aplReplicate(aplReplicate (a, c(2,2), 1), c(0,2,0), 2)
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    3    3
## [2,]    3    3
## [3,]    4    4
## [4,]    4    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    9    9
## [2,]    9    9
## [3,]   10   10
## [4,]   10   10
##
## , , 3
##
##      [,1] [,2]
## [1,]   15   15
## [2,]   15   15
## [3,]   16   16
## [4,]   16   16
##
## , , 4
##
```

```
##      [,1] [,2]
## [1,]   21   21
## [2,]   21   21
## [3,]   22   22
## [4,]   22   22
```

```
aplReshape (c(1,2), c(2,2,2))
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    2
##
## , , 2
##
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    2
```

```
aplReshape (array(1:24, c(2,3,4)), c(2,2))
```

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

3.15 Rotate

Rotate (Helzer (1989), p. 191—193) cyclically shifts the elements of a vector or array dimension. In APL we write either L⌽R or L⊖R, depending on whether rotation is along the first or the last axis.

For vectors `aplRotate(x, k)` with positive `k` takes the first `k` elements of `x` and moves them to the end, with negative `k` it takes the last `k` elements and moves them to the front.

```
aplRotate(1:6, 2)
```

```
## [1] 3 4 5 6 1 2
```

```r
aplRotate(1:6, -2)
```

```
## [1] 5 6 1 2 3 4
```

For a matrix we can shift rows or columns. For a matrix R the expression L⌽R, or `aplRotate (R, L, axis = 2)`, shifts the elements of rows, with within each row possibly a different shift. Thus the number of elements of L must be the number of rows of R. If L is a scalar, then it is basically first blown up into a vector, and the same shift L is applied to each row (which means that a column is shifted). If `axis = 1` then elements within columns are shifted. By default the argument axis equals `aplRank(a)`.

```r
a <- cbind(1:5, matrix (0,5,4))
aplRotate (a, 2)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    0    1    0
## [2,]    0    0    0    2    0
## [3,]    0    0    0    3    0
## [4,]    0    0    0    4    0
## [5,]    0    0    0    5    0
```

```r
aplRotate (a, -c(0,1,2,3,4))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    0    0    0
## [2,]    0    2    0    0    0
## [3,]    0    0    3    0    0
## [4,]    0    0    0    4    0
## [5,]    0    0    0    0    5
```

```r
aplRotate (a, 2, 1)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    3    0    0    0    0
## [2,]    4    0    0    0    0
## [3,]    5    0    0    0    0
## [4,]    1    0    0    0    0
## [5,]    2    0    0    0    0
```

For arrays matters become more complicated. To rotate R along k we need dim(L) to be dim(R)[-k]. Then each element in L defines a slice of R that is a vector of length dim(R)[k]. That vector then gets rotated using the positive or negative value of the corresponding element of L.

```r
a <- array(1:24, c(2,3,4))
aplRotate (a, 1, 1)
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    1    3    5
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    8   10   12
## [2,]    7    9   11
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   14   16   18
## [2,]   13   15   17
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]   20   22   24
## [2,]   19   21   23
```

```r
b <- matrix (c(0,0,0,1,1,1), 2, 3, byrow = TRUE)
aplRotate (a, b, 3)
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    8   10   12
##
## , , 2
##
```

```
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]   14   16   18
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   20   22   24
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]   19   21   23
## [2,]    2    4    6
```

3.16 Scan

The APL operator scan does not change dimension (Helzer (1989), p. 195—197). The expression f\R for a vector R can be defined in terms of reduction. It produces the vector with elements f/R[1] , f/R[1 2] , f/R[1 2 3], …. Thus it generalizes R funcions such as `cumsum()`, `cumprod()`, and so on. For a matrix f\R scans the rows and f\[1]R or f⍀R scans the columns. For arrays we again need an axis to expand along.

```
a<-matrix(1:9,3,3)
aplScan(a, 1, "+")
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    3    9   15
## [3,]    6   15   24
```

```
aplScan(a, f = min)
```

```
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    2    2    2
## [3,]    3    3    3
```

```
a<-array(1:24,c(2,3,4))
aplScan(a, 3, "*")
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7   27   55
## [2,]   16   40   72
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   91  405  935
## [2,]  224  640 1296
##
## , , 4
##
##      [,1]  [,2]  [,3]
## [1,] 1729  8505 21505
## [2,] 4480 14080 31104
```

3.17 Select

Select is not an APL function. Our function `aplSelect()` has an array `a` as its first argument, and a list of `aplRank(a)` vectors of integers as it second element. It then selects the corresponding rows, columns, slices, and so on.

```
a <- array(1:24, c(2,3,4))
aplSelect(a, list(1,c(1,2),c(3,4)))
```

```
##      [,1] [,2]
## [1,]   13   19
## [2,]   15   21
```

```
aplSelect(a, list(1,c(1,2),c(3,4)), drop = FALSE)
```

```
## , , 1
##
##      [,1] [,2]
## [1,]   13   15
```

```
##
## , , 2
##
##      [,1] [,2]
## [1,]   19   21
```

3.18 Set

There is no APL function corresponding with set. We wrote aplSet() as a simple utility to set an element of an array to a value.

```
a <- array(1:12, c(2,3,2))
aplSet (a, 11, c(2,2,2))
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   11   12
```

3.19 Shape

Shape is the monadic version of ρ, while reshape is the dyadic version. Shape gives the dimensions of an array, an reshape modifies them. Of course aplShape() is just a wrapper for the basic R function dim().

```
a <- array(1:12, c(2,3,2))
aplShape(a)
```

```
## [1] 2 3 2
```

```
aplShape(aplShape(a))
```

```
## [1] 3
```

3.20 Take

Take L↑R extracts items from vectors and arrays, in the same way as drop L↓R deletes items. If R is a vector then L items from the beginning of R are taken if L is positive, and L items from the end if L is negative. If R is an array, then L must have ρρR elements. Depending on whether the elements or L are positive or negative, the L first or last parts of the dimension are taken

Our R implementation again interchanges the two arguments. Note that in base R the default on subsetting arrays is `drop = TRUE`. In our implementations the default is always `drop = FALSE`. Both `aplTake()` and `aplDrop()` are implemented by constructing a suitable list of index vectors for `aplSelect()`. Note that more general subsetting can be done with `aplSelect()`. Note that

So for a vector

```
aplTake(1:10,3)
```

```
## [1] 1 2 3
```

```
aplTake(1:10,-3)
```

```
## [1]  8  9 10
```

and for an array

```
a <- array(1:24, c(2,3,4))
aplTake(a, c(2,3,2), drop = TRUE)
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

```r
aplTake(a, c(2,-2,1), drop = TRUE)
```

```
##      [,1] [,2]
## [1,]    3    5
## [2,]    4    6
```

3.21 Transpose

APL has both a monadic ⍉R and a dyadic L⍉R transpose. This APL transpose has a somewhat tortuous relationship with `aperm()` in R.

The monadic `aplTranspose(a)` and `aperm(a)` are always the same, they reverse the order of the dimensions.

If `x` is a permutation of `1:aplRank(a)`, then `aperm(a,x)` is actually equal to `aplTranspose(a,order(x))` For permutations we could consequently define `aplTranspose(a,x)` simply as `aperm(a,order(x))` (which would undoubtedly be more efficient as well).

```r
a <- array(1:24, c(2, 3, 4))
aplTranspose (a)
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    7    9   11
## [3,]   13   15   17
## [4,]   19   21   23
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    8   10   12
## [3,]   14   16   18
## [4,]   20   22   24
```

```r
aplTranspose (a, c(2,1,3))
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
##
## , , 2
##
##      [,1] [,2]
## [1,]    7    8
## [2,]    9   10
## [3,]   11   12
##
## , , 3
##
##      [,1] [,2]
## [1,]   13   14
## [2,]   15   16
## [3,]   17   18
##
## , , 4
##
##      [,1] [,2]
## [1,]   19   20
## [2,]   21   22
## [3,]   23   24
```

If `x` is not a permutation, then `aperm(a,x)` is undefined, but `aplTranspose(a,x)` can still be defined in some cases. If `x` has `aplRank(a)` elements equal to one of `1:m`, with each of `1:m` occurring a least once, then `aplTranspose(a,x)` has rank `m`. If an integer between 1 and `m` occurs more than once, then the corresponding diagonal of `a` is selected.

```
a
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

```
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18
##
## , , 4
##
##      [,1] [,2] [,3]
## [1,]   19   21   23
## [2,]   20   22   24
```

```
aplTranspose (a, c(2,2,1))
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    7   10
## [3,]   13   16
## [4,]   19   22
```

4 Implementations

4.1 Six Versions

We have implemented the APL functions in six different ways

1. As pure R.
2. As R, with only decode and encode in C using `.C()`.
3. As C, using the `.C()` interface for decode, encode, transpose, select, reduce, scan, and inner product.
4. As C, using the `.Call()` interface for decode, encode, transpose, select, reduce, scan, and inner product.
5. As C, using the `.Call()` interface, for transpose, select, reduce, scan, and inner product, with decode and encode inlined using `.C()`.
6. As C using `Rcpp` for decode, encode, transpose, select, reduce, scan, and inner product.

It must be emphasized that the `Rcpp` interface was written five years ago, with a very early version of `Rcpp`, that uses only a tiny subset of the possibilities offered by newer versions. We are sure a great deal of improvement is possible there.

Also for reduce, scan, and inner product some of our arguments are functions. In the .C() implementation we use the old Call_R() interface, dating back to the Blue Book (Becker, Chambers, and Wilks (1988)), to handle function pointers.

Note that inlining decode and encode makes sense, since they are called so many times in all functions, but the inline keyword is handled only as a hint to the compiler. It does not guarantee actual inlining of the function.

The computations in the body of the paper use the Call() interface. The different implementations calling C routines use different shared libraries and different glue routines in R.

4.2 Timing

We compare running time of the six implementations. The parameters we use are

```
repNum <- 10
timeArr <- array(NA, c(repNum,3,3,6))
a<-array(1:10000,c(10,10,100))
b<-array(1:10000,c(100,10,10))
c<-array(1:100000,rep(10,5))
d<-array(1:100000,rep(10,5))
x<-list(1:5,1:5,1:5,1:5,1:5)
```

and the results are

```
## , , InnerProduct
##
##                R    R+.C     .C  .Call Inline   Rcpp
## user    9.7987 10.8804 0.8723 0.3620 0.2348 0.7785
## system  0.0372  0.0635 0.0491 0.0046 0.0030 0.0100
## elapsed 9.9047 11.1021 0.9257 0.3687 0.2398 0.8048
##
## , , Reduce
##
##                R   R+.C     .C  .Call Inline   Rcpp
## user    1.6892 1.0227 0.0464 0.0476 0.0322 0.0732
## system  0.0086 0.0071 0.0034 0.0016 0.0010 0.0025
## elapsed 1.7131 1.0435 0.0498 0.0495 0.0335 0.0767
##
## , , Select
##
##                R   R+.C    .C .Call Inline   Rcpp
## user    0.0624 0.0497 8e-04 8e-04  5e-04 0.0024
## system  0.0002 0.0005 0e+00 0e+00  0e+00 0.0001
## elapsed 0.0627 0.0563 7e-04 9e-04  6e-04 0.0024
```

5 Appendix: Code

5.1 R Code

```r
# select

aplSelect <- function(a, x, drop = FALSE) {
  sa <- aplShape(a)
  ra <- aplRank(a)
  sz <- sapply(x, length)
  z <- array(0, sz)
  nz <- prod(sz)
  for (i in 1:nz) {
    ivec <- aplEncode(i, sz)
    jvec <- vector()
    for (j in 1:ra)
      jvec <- c(jvec, x[[j]][ivec[j]])
    z[i] <- a[aplDecode(jvec, sa)]
  }
  if (drop)
    return(drop(z))
  else
    return(z)
}

#  drop

aplDrop <- function(a, x, drop = FALSE) {
  sa <- aplShape(a)
  ra <- aplRank(a)
  y <- as.list(rep(0, ra))
  for (i in 1:ra) {
    ss <- sa[i]
    xx <- x[i]
    sx <- ss + xx
    if (xx >= 0)
      y[[i]] <- (xx + 1):ss
    if (xx < 0)
      y[[i]] <- 1:sx
  }
  return(aplSelect(a, y, drop))
}
```

```
#  take

aplTake <- function(a, x, drop = FALSE) {
  sa <- aplShape(a)
  ra <- aplRank(a)
  y <- as.list(rep(0, ra))
  for (i in 1:ra) {
    ss <- sa[i]
    xx <- x[i]
    sx <- ss + xx
    if (xx > 0)
      y[[i]] <- 1:xx
    if (xx < 0)
      y[[i]] <- (sx + 1):ss
  }
  return(aplSelect(a, y, drop))
}

# reduce vector

aplRDV <- function(x, f = "+") {
  if (length(x) == 0)
    return(x)
  s <- x[1]
  if (length(x) == 1)
    return(s)
  for (i in 2:length(x))
    s <- match.fun(f)(s, x[i])
  return(s)
}

# scan vector

aplSCV <- function(x, f = "+") {
  if (length(x) <= 1)
    return(x)
  return(sapply(1:length(x), function(i)
    aplRDV(x[1:i], f)))
}

# inner product vector

aplIPV <- function(x, y, f = "*", g = "+") {
  if (length(x) != length(y))
```

```
    stop("Incorrect vector length")
  if (length(x) == 0)
    return(x)
  z <- match.fun(f)(x, y)
  return(aplRDV(z, g))
}

#  expand vector

aplEXV <- function(x, y) {
  z <- rep(0, length(y))
  m <- which(y == TRUE)
  if (length(m) != length(x))
    stop("Incorrect vector length")
  z[m] <- x
  return(z)
}

#  expand

aplExpand <- function(x, y, axis = 1) {
  if (is.vector(x))
    return(aplEXV(x, y))
  d <- dim(x)
  m <- which(y == TRUE)
  n <- length (y)
  e <- d
  e[axis] <- n
  if (length(m) != d[axis])
    stop("Incorrect dimension length")
  z <- array(0, e)
  for (i in 1:prod(d)) {
    k <- aplEncode (i, d)
    k[axis] <- m[k[axis]]
    z[aplDecode (k, e)] <- x[i]
   }
  return (z)
}

#  compress/replicate vector

aplCRV <- function(x, y) {
  n <- aplShape(x)
  m <- aplShape(y)
  if (m == 1)
    y <- rep(y, n)
  if (length(y) != n)
    stop("Length Error")
  z <- vector()
  for (i in 1:n)
    z <- c(z, rep(x[i], y[i]))
  return(z)
}

#  compress/replicate

aplReplicate <- function(x, y, k = aplRank (y)) {
  if (is.vector(x))
    return(aplCRV(x, y))
  sx <- aplShape(x)
  sy <- aplShape(y)
  sk <- sx[k]
  if (max(sy) == 1)
    y <- rep(y, sk)
  if (length(y) != sk)
    stop("Length Error")
  sz <- sx
  sz[k] <- sum(y)
  nz <- prod(sz)
  gg <- aplCRV(1:sk, y)
  z <- array(0, sz)
  for (i in 1:nz) {
    jvec <- aplEncode(i, sz)
    jvec[k] <- gg[jvec[k]]
    z[i] <- x[aplDecode(jvec, sx)]
  }
  return(z)
}

#  rotate vector

aplRTV <- function(a, k) {
  n <- aplShape(a)
  if (k > 0)
    return(c(a[-(1:k)], a[1:k]))
  if (k < 0)
    return(c(a[(n + k + 1):n], a[1:(n + k)]))
  return(a)
```

```
}

#  rotate

aplRotate <- function(a, b, axis = aplRank (a)) {
  if (is.vector(a))
    return(aplRTV(a, b))
  sa <- aplShape(a)
  sx <- aplShape(b)
  if (max(sx) == 1)
    b <- array(b, sa[-axis])
  if (!identical(sa[-axis], aplShape(b)))
    stop("Index Error")
  z <- array(0, sa)
  sz <- sa
  nz <- prod(sz)
  sk <- sz[axis]
  for (i in 1:nz) {
    ivec <- aplEncode(i, sz)
    xx <- b[aplDecode(ivec[-axis], sx)]
    ak <- rep(0, sk)
    for (j in 1:sk) {
      jvec <- ivec
      jvec[axis] <- j
      ak[j] <- a[aplDecode(jvec, sz)]
    }
    bk <- aplRTV(ak, xx)
    for (j in 1:sk) {
      jvec <- ivec
      jvec[axis] <- j
      z[aplDecode(jvec, sz)] <- bk[j]
    }
  }
  return(z)
}

# transpose -- will be overwritten by the C version

aplTranspose <- function(a, x = rev(1:aplRank(a))) {
  sa <- aplShape(a)
  ra <- aplRank(a)
  if (length(x) != ra)
    stop("Length Error")
  rz <- max(x)
```

```r
  sz <- rep(0, rz)
  for (i in 1:rz)
    sz[i] <- min(sa[which(x == i)])
  nz <- prod(sz)
  z <- array(0, sz)
  for (i in 1:nz)
    z[i] <- a[aplDecode(aplEncode(i, sz)[x], sa)]
  return(z)
}

#  representation -- will be overwritten by the C version

aplEncode <- function(rrr, base) {
  b <- c(1, butLast(cumprod(base)))
  r <- rep(0, length(b))
  s <- rrr - 1
  for (j in length(base):1) {
    r[j] <- s %/% b[j]
    s <- s - r[j] * b[j]
  }
  return(1 + r)
}

#  base value -- will be overwritten by the C version

aplDecode <- function(ind, base) {
  b <- c(1, butLast(cumprod(base)))
  return(1 + sum(b * (ind - 1)))
}

# get

aplGet <- function(a, cell) {
  dims <- dim(a)
  n <- length(dims)
  b <- 0
  if (any(cell > dims) || any(cell < 1))
    stop("No such cell")
  return(a[aplDecode(cell, dims)])
}

# set

aplSet <- function(a, b, cell) {
  dims <- dim(a)
  n <- length(dims)
  if (any(cell > dims) || any(cell < 1))
    stop("No such cell")
  a[aplDecode(cell, dims)] <- b
  return(a)
}

#  join

aplJoin <- function(a, b, axis = 1) {
  if (is.vector(a) && is.vector(b))
    return(c(a, b))
  sa <- aplShape(a)
  sb <- aplShape(b)
  ra <- aplRank(a)
  rb <- aplRank(b)
  if (ra != rb)
    stop("Rank error in aplJoin")
  if (!identical(sa[-axis], sb[-axis]))
    stop("Shape error")
  sz <- sa
  sz[axis] <- sz[axis] + sb[axis]
  nz <- prod(sz)
  u <- unit(axis, ra)
  z <- array(0, sz)
  for (i in 1:nz) {
    ivec <- aplEncode(i, sz)
    if (ivec[axis] <= sa[axis])
      z[i] <- a[aplDecode(ivec, sa)]
    else
      z[i] <- b[aplDecode(ivec - sa[axis] * u, sb)]
  }
  return(z)
}

# ravel

aplRavel <- function(a) {
  as.vector(a)
}

# outer product
```

```r
aplOuterProduct <- function(x, y, f = "*") {
  return(outer(x, y, f))
}

# shape

aplShape <- function(a) {
  if (is.vector(a))
    return(length(a))
  return(dim(a))
}

#  rank

aplRank <- function(a) {
  aplShape(aplShape(a))
}

# reshape

aplReshape <- function(a, d) {
  return(array(a, d))
}

# reduce -- will be overwritten by C version

aplReduce <- function(a, k), f = "+") {
  if (is.vector(a))
    return(aplRDV(a, f))
  ff <- if (is.function(f))
    f
  else
    match.fun(f)
  sa <- aplShape(a)
  ra <- aplRank(a)
  na <- prod(sa)
  sz <- sa[(1:ra)[-k]]
  z <- array(0, sz)
  nz <- prod(sz)
  ind <- rep(0, nz)
  for (i in 1:na) {
    ivec <- aplEncode(i, sa)
    jind <- aplDecode(ivec[-k], sz)
    if (ind[jind] == 0) {
      z[jind] <- a[i]
      ind[jind] <- 1
    }
    else
      z[jind] <- ff(z[jind], a[i])
  }
  return(z)
}

# scan -- will be overwritten by the C version

aplScan <- function(a, k = aplRank(a), f = "+") {
  if (is.vector(a))
    return(aplSCV(a, f))
  ff <- if (is.function(f))
    f
  else
    match.fun(f)
  sa <- aplShape(a)
  ra <- aplRank(a)
  sk <- sa[k]
  u <- unit(k, ra)
  na <- prod(sa)
  z <- a
  for (i in 1:na) {
    ivec <- aplEncode(i, sa)
    sk <- ivec[k]
    if (sk == 1)
      z[i] <- a[i]
    else
      z[i] <- ff(z[aplDecode(ivec - u, sa)], a[i])
  }
  return(z)
}

# inner product -- will be overwritten by the C version

aplInnerProduct <- function(a, b, f = "*", g = "+") {
  sa <- aplShape(a)
  sb <- aplShape(b)
  ra <- aplRank(a)
  rb <- aplRank(b)
  ia <- 1:(ra - 1)
  ib <- (ra - 1) + (1:(rb - 1))
  ff <- match.fun(f)
  gg <- match.fun(g)
  ns <- last(sa)
  nt <- first(sb)
  if (ns != nt)
    stop("Incompatible array dimensions")
  sz <- c(butLast(sa), butFirst(sb))
  nz <- prod(sz)
  z <- array(0, sz)
  for (i in 1:nz) {
    ivec <- aplEncode(i, sz)
    for (j in 1:ns) {
      aa <- a[aplDecode(c(ivec[ia], j), sa)]
      bb <- b[aplDecode(c(j, ivec[ib]), sb)]
      tt <- ff(aa, bb)
      if (j == 1)
        z[i] <- tt
      else
        z[i] <- gg(z[i], tt)
    }
  }
  return(z)
}

# member of

aplMemberOf <- function(a, b) {
  sa <- aplShape(a)
  sb <- aplShape(b)
  na <- prod(sa)
  nb <- prod(sb)
  z <- array (0, sa)
  for (i in 1:na) {
    aa <- a[i]
    for (j in 1:nb)
      if (aa == b[j])
        z[i] <- 1
  }
  return(z)
}

# utilities below

first <- function(x) {
  return(x[1])
}

butFirst <- function(x) {
  return(x[-1])
}

last <- function(x) {
  return(x[length(x)])
}

butLast <- function(x) {
  return(x[-length(x)])
}

unit <- function(i, n) {
  ifelse(i == (1:n), 1, 0)
}
```

5.2 R Glue for .C()

```
aplDecode <-
  function(cell, dims) {
    n <- length(dims)
    if (any(cell > dims) || any(cell < 1))
      stop("No such cell")
    .C("aplDecodeC ",
       as.integer(cell),
       as.integer(dims),
       as.integer(n),
       as.integer(1))[[4]]
  }


aplEncode <-
  function(ind, dims) {
    n <- length(dims)
    cell <- integer(n)
    if ((ind < 1) || (ind > prod(dims)))
      stop("No such cell")
    .C("aplEncodeC ",
       as.integer(cell),
      as.integer(dims),
      as.integer(n),
      as.integer(ind))[[1]]
  }


aplInnerProduct <-
  function(a, b, f = "*", g = "+") {
    sa <- aplShape(a)
    sb <- aplShape(b)
    ra <- aplRank(a)
    rb <- aplRank(b)
    ia <- 1:(ra - 1)
    ib <- (ra - 1) + (1:(rb - 1))
    ff <- match.fun(f)
    gg <- match.fun(g)
    ns <- last(sa)
    nt <- first(sb)
    if (ns != nt)
      stop("Incompatible array dimensions")
    sz <- c(butLast(sa), butFirst(sb))
    nz <- prod(sz)
    z <- array(0, sz)
    rz <- aplRank(z)
    res <-
      .C(
        "aplInnerProductC",
        list(ff, gg),
        as.double(a),
        as.double(b),
        as.integer(sa),
        as.integer(ra),
        as.integer(sb),
        as.integer(rb),
        as.integer(sz),
        as.integer(rz),
        as.integer(nz),
        as.integer(ns),
        as.double(z)
      )
    return(array(res[[12]], sz))
  }

aplReduce <-
```

```
  function(a, k, f = "+") {
    if (is.vector(a))
      return(aplRDV(a, f))
    ff <- if (is.function(f))
      f
    else
      match.fun(f)
    sa <- aplShape(a)
    ra <- aplRank(a)
    na <- prod(sa)
    sz <- sa[(1:ra)[-k]]
    z <- array(0, sz)
    rz <- aplRank(z)
    nz <- prod(sz)
    z <-
      .C(
        "aplReduceC",
        list(ff),
        as.double(a),
        as.integer(k),
        as.integer(length(k)),
        as.integer(na),
        as.integer(sa),
        as.integer(ra),
        as.integer(nz),
        as.integer(sz),
        as.integer(rz),
        as.double(z)
      )
    return(array(z[[11]], sz))
  }

aplScan <-
  function(a, k, f = "+") {
    if (is.vector(a))
      return(aplSCV(a, f))
    ff <- if (is.function(f))
      f
    else
      match.fun(f)
    sa <- aplShape(a)
    ra <- aplRank(a)
    sk <- sa[k]
    u <- unit(k, ra)
    na <- prod(sa)
    z <- a
    res <- .C(
      "aplScanC",
      list(ff),
      as.double(a),
      as.integer(k),
      as.integer(na),
      as.integer(sa),
      as.integer(ra),
      as.double(z)
    )
    return(array(res[[7]], sa))
  }

aplSelect <-
  function(a, x, drop = FALSE) {
    sa <- aplShape(a)
    ra <- aplRank(a)
    sz <- sapply(x, length)
    z <- array(0, sz)
    rz <- aplRank(z)
    nz <- prod(sz)
    z <-
      array(
        .C(
          "aplSelectC",
          as.double(a),
          as.integer(sa),
          as.integer(ra),
          lapply(x, as.integer),
          as.double(z),
          as.integer(sz),
          as.integer(rz),
          as.integer(nz)
        )[[5]],
        sz
      )
    if (drop)
      return(drop(z))
    else
      return(z)
  }
```

```r
aplTranspose <-
  function(a, x = rev(1:aplRank(a))) {
    sa <- aplShape(a)
    ra <- aplRank(a)
    na <- prod(sa)
    if (length(x) != ra)
      stop("Length Error")
    rz <- max(x)
    sz <- rep(0, rz)
    for (i in 1:rz)
      sz[i] <- min(sa[which(x == i)])
    nz <- prod(sz)
    z <- array(0, sz)
    array(
      .C(
        "aplTransposeC",
        as.double(a),
        as.integer(x),
        as.integer(sa),
        as.integer(ra),
        as.integer(na),
        as.integer(sz),
        as.integer(rz),
        as.integer(nz),
        as.double(z)
      )[[9]],
      sz
    )
  }
```

5.3 R Glue for .Call()

```r
aplDecode <- function(cell, dims) {
  if (length(cell) != length(dims)) {
    stop("Dimension error")
  }
  if (any(cell > dims) || any (cell < 1)) {
    stop("No such cell")
  }
  .Call("APLDECODE", as.integer(cell), as.integer(dims))
}
```

```r
aplEncode <- function(ind, dims) {
  if (length(ind) > 1) {
    stop ("Dimension error")
  }
  if ((ind < 1) || (ind > prod(dims))) {
    stop ("No such cell")
  }
  .Call("APLENCODE", as.integer(ind), as.integer(dims))
}


aplInnerProduct <- function(a, b, f = "*", g = "+") {
  sa <- aplShape(a)
  ra <- aplRank(a)
  ia <- 1:(ra - 1)
  sb <-
    aplShape(b)
  rb <- aplRank(b)
  ib <- (ra - 1) + (1:(rb - 1))
  if (!is.function(f)) {
    f <- match.fun(f)
  }
  if (!is.function(g)) {
    g <- match.fun(g)
  }
  env <- new.env()
  environment(f) <- env
  environment(g) <- env
  ns <- last(sa)
  nt <- first(sb)
  if (ns != nt) {
    stop("Incompatible array dimensions")
  }
  sz <- c(butLast(sa), butFirst(sb))
  z  <- .Call(
    "APLINNERPRODUCT",
    f,
    g,
    as.double(a),
    as.double(b),
    as.integer(sa),
    as.integer(sb),
    as.integer(sz),
    as.integer(ns),
    env
  )
  return(array(z, sz))
}


aplReduce <- function(a, k, f = "+") {
  if (is.vector(a)) {
    return(aplRDV(a, f))
  }
  nk  <- length(k)
  sa  <- aplShape(a)
  sz  <- sa[(1:length(sa))[-k]]
  if (!is.function(f)) {
    f <- match.fun(f)
  }
  env <- new.env()
  environment(f) <- env
  z <- .Call("APLREDUCE",
             f,
             as.double(a),
             as.integer(k),
             as.integer(sa),
             as.integer(sz),
             env)
  return(array(z, sz))
}

aplScan <- function(a, k = aplRank (a), f = "+") {
  if (is.vector(a)) {
    return(aplSCV(a, f))
  }
  if (!is.function(f)) {
    f <- match.fun(f)
  }
  env <- new.env()
  environment(f) <- env
  sa <- aplShape(a)
  ra <- aplRank(a)
  z  <- .Call("APLSCAN",
              f,
              as.double(a),
              as.integer(k),
              as.integer(sa),
              as.integer(ra),
              env)
  return(array(z, sa))
}

aplSelect <- function (a, x, drop = TRUE) {
  dima = aplShape(a)
  if (length(dima) != length(x)) {
    stop("Dimension error")
  }
  z <- .Call("APLSELECT",
             as.double(a),
             as.integer(dima),
             lapply(x, as.integer))

  z <- array(z, sapply(x, length))
  if (drop) {
    return(drop(z))
  }
  return(z)
}


aplTranspose <- function(a, x = rev(1:aplRank(a))) {
  sa <- aplShape(a)
  if (length(x) != length(sa)) {
    stop("Length Error")
  }
  rz <- max(x)
  sz <- rep(0, rz)
  for (i in 1:rz) {
    sz[i] <- min(sa[which(x == i)])
  }
  z <- .Call(
    "APLTRANSPOSE",
    as.double(a),
    as.integer(x),
    as.integer(sa),
    as.integer(sz),
    as.integer(rz)
  )
  return(array(z, sz))
}
```

5.4 C Code using .C()

```
#include <R.h>
#include <Rinternals.h>
#include <Rinterface.h>

static char* f2_char;
static char* g2_char;

void aplDecodeC(int* cell, int* dims, int* n, int* ind) {
  int i, aux = 1; *ind = 1;
  for (i = 0; i < *n; i++) {
    *ind += aux * (cell[i] - 1);
    aux *= dims[i];
  }
}

void aplEncodeC(int* cell, int* dims, int* n, int* ind) {
  int i, aux = *ind, nn = *n, pdim = 1;
  for (i = 0; i < nn - 1; i++)
    pdim *= dims[i];
  for (i = nn - 1; i > 0; i--) {
    cell[i] = (aux - 1)/pdim;
    aux -= pdim*cell[i];
    pdim /= dims[i-1];
    cell[i] += 1;
  }
  cell[0] = aux;
}

void aplSelectC(double *a, int *sa, int *ra, SEXP *list, double *z, int *sz, int *rz, in
{
  int i, j, k;
  int ivec[*rz], jvec[*ra];
  for (i = 0; i < *nz; i++) {
    k = i + 1;
    (void) aplEncodeC (ivec,sz,rz,&k);
    for (j = 0; j < *ra; j++) {
      SEXP vec = list[j];
      jvec[j] = INTEGER(vec)[ivec[j]-1];
    }
    k = 1;
    (void) aplDecodeC (jvec,sa,ra,&k);
    z[i] = a[k - 1];
  }
}

void aplTransposeC(double *a, int *x, int *sa, int *ra, int *na, int *sz, int *rz, int *
{
  int i, j, r, ivec[*rz], jvec[*ra];
  for (i = 0; i < *nz; i++){
    r = i + 1;
    (void) aplEncodeC(ivec,sz,rz,&r);
    for (j = 0; j < *ra; j++)
      jvec[j] = ivec[x[j]-1];
    (void) aplDecodeC(jvec,sa,ra,&r);
    z[i] = a[r - 1];
  }
}

void aplReduceC(char** funclist, double *a, int *k, int *nk, int *na, int *sa, int *ra,
  double f2_glue(double, double);
  double f2_comp(double (*)(),double,double);
  int i, j, u, v, r, m, kk = (*ra) - (*nk), ivec[*ra], kvec[kk], ind[*nz];
  for (i = 0; i < *nz; i++) ind[i] = 0;
  f2_char = funclist[0];
  for (i = 0; i < *na; i++){
    r = i + 1;
    (void) aplEncodeC(ivec,sa,ra,&r);
    u = 0;
    for (j = 0; j < *ra; j++) {
      r = 0;
      for (v = 0; v < *nk; v++) {
        if (j == (k[v] - 1)) r = 1;
      }
      if (r == 0)   {
        kvec[u] = ivec[j];
        u += 1;
      }
    }
    (void) aplDecodeC(kvec,sz,rz,&m);
    if (ind[m - 1] == 0) {
      z[m - 1] = a[i];
      ind[m - 1] = 1;
    }
    else
      z[m - 1] = f2_comp((double(*)())f2_glue,z[m - 1],a[i]);
  }
```

```
}

void aplScanC(char** funclist,double *a, int *k, int *na, int *sa, int *ra, double *z)
{
  double f2_glue(double, double);
  double f2_comp(double (*)(),double,double);
  int i, r, sk, l, ivec[*ra];
  l = *k - 1;
  f2_char = funclist[0];
  for (i = 0; i < *na; i++){
    r = i + 1;
    (void) aplEncodeC(ivec,sa,ra,&r);
    sk = ivec[l];
    if (sk == 1) z[i] = a[i];
    else {
      ivec[l] -= 1;
      (void) aplDecodeC(ivec,sa,ra,&r);
      z[i] = f2_comp((double(*)())f2_glue,z[r - 1],a[i]);
    }
  }
}

void aplInnerProductC(char** funclist, double *a, double *b, int *sa, int *ra, int *sb,
  double f2_glue(double, double);
  double g2_glue(double, double);
  double f2_comp(double (*)(),double,double);
  double g2_comp(double (*)(),double,double);
  double t; int i, j, r, k, l, u, ivec[*rz], jvec[*ra], kvec[*rb];
  f2_char = funclist[0]; g2_char= funclist[1]; k = l = 0;
  for (i = 0; i < *nz; i++) {
    r = i + 1;
    (void) aplEncodeC(ivec,sz,rz,&r);
    for (j = 0; j < *ns; j++) {
      for (u = 0; u < *ra - 1; u++)
        jvec[u] = ivec[u];
      jvec[*ra - 1] = j + 1;
      (void) aplDecodeC(jvec,sa,ra,&k);
      for (u = 1; u < *rb; u++)
        kvec[u] = ivec[*ra + u - 2];
      kvec[0] = j + 1;
      (void) aplDecodeC(kvec,sb,rb,&l);
      t = f2_comp((double(*)())f2_glue,a[k-1],b[l-1]);
      if (j == 0) z[i] = t;
      else z[i] = g2_comp((double(*)())g2_glue,t,z[i]);
    }
  }
}

double f2_glue(double x, double y){
  char *modes[2], *values[1];
  void *args[2];
  double xx[1], yy[1], *result;
  long lengths[2], nargs = (long)2,  nvals = (long)1;
  lengths[0] = lengths[1] = (long)1;
  nargs = (long)2;
  args[0] = (void *)xx; xx[0] = x;
  args[1] = (void *)yy; yy[0] = y;
  modes[0] = modes[1] = "double";
  call_R(f2_char,nargs,args,modes,lengths,0,nvals,values);
  result = (double*)values[0];
  return(result[0]);
}

double g2_glue(double x, double y){
  char *modes[2], *values[1];
  void *args[2];
  double xx[1], yy[1], *result;
  long lengths[2], nargs = (long)2,  nvals = (long)1;
  lengths[0] = lengths[1] = (long)1;
  nargs = (long)2;
  args[0] = (void *)xx; xx[0] = x;
  args[1] = (void *)yy; yy[0] = y;
  modes[0] = modes[1] = "double";
  call_R(g2_char,nargs,args,modes,lengths,0,nvals,values);
  result = (double*)values[0];
  return(result[0]);
}

double f2_comp(double (*f)(), double x, double y) {
  return f(x,y);
}

double g2_comp(double (*g)(), double x, double y) {
  return g(x,y);
}
```

5.5 C Code using .Call()

```c
#include <R.h>
#include <Rinternals.h>

SEXP APLDECODE( SEXP, SEXP );
SEXP APLENCODE( SEXP, SEXP );
SEXP APLSELECT( SEXP, SEXP, SEXP );
SEXP APLTRANSPOSE( SEXP, SEXP, SEXP, SEXP, SEXP );
SEXP APLSCAN( SEXP, SEXP, SEXP, SEXP, SEXP );
SEXP APLREDUCE( SEXP, SEXP, SEXP, SEXP, SEXP, SEXP );
SEXP APLINNERPRODUCT( SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP );

SEXP
APLDECODE( SEXP cell, SEXP dims )
{
    int             aux = 1, n = length(dims);
    SEXP            ind;
    PROTECT( ind = allocVector( INTSXP, 1 ) );
    INTEGER( ind )[0] = 1;
    for( int i = 0; i < n; i++ ) {
        INTEGER( ind )[0] += aux * ( INTEGER( cell )[i] - 1 );
        aux *= INTEGER(dims)[i];
    }
    UNPROTECT( 1 );
    return (ind);
}

SEXP
APLENCODE( SEXP ind, SEXP dims )
{
    int             n = length(dims), aux = INTEGER(ind)[0], pdim = 1;
    SEXP            cell;
    PROTECT( cell = allocVector( INTSXP, n ) );
    for ( int i = 0; i < n - 1; i++ )
        pdim *= INTEGER( dims )[i];
    for ( int i = n - 1; i > 0; i-- ){
        INTEGER( cell )[i] = ( aux - 1 ) / pdim;
        aux -= pdim * INTEGER( cell )[i];
        pdim /= INTEGER( dims )[i - 1];
        INTEGER( cell )[i] += 1;
    }
    INTEGER( cell )[0] = aux;
    UNPROTECT( 1 );
    return cell;
}

SEXP
APLSELECT( SEXP a, SEXP dima, SEXP list ) {
   int             r = length (dima), lz = 1, dimzi, nProtect = 0;
   SEXP            dimz, itel, cell, czll, nind, z;
   PROTECT(dimz = allocVector (INTSXP, r));   nProtect++;
   PROTECT(cell = allocVector (INTSXP, r));   nProtect++;
   PROTECT(czll = allocVector (INTSXP, r));   nProtect++;
   PROTECT(itel = allocVector (INTSXP, 1));   nProtect++;
   PROTECT(nind = allocVector (INTSXP, 1));   nProtect++;
   for( int i = 0; i < r; i++ ){
       dimzi = length( VECTOR_ELT( list, i ) );
       INTEGER( dimz )[i] = dimzi;
       lz *= dimzi;
   }
   PROTECT( z = allocVector( REALSXP, lz ) );
   nProtect++;
   for( int i = 0; i < lz; i++ ){
       INTEGER(itel)[0] = i + 1;
       cell = APLENCODE (itel, dimz);
       for (int j = 0; j < r; j++) {
           INTEGER (czll)[j] = INTEGER (VECTOR_ELT (list, j))[INTEGER(cell)[j] - 1];
       }
       nind = APLDECODE( czll, dima );
       REAL(z)[i] = REAL(a)[INTEGER(nind)[0] - 1];
   }
   UNPROTECT(nProtect);
   return(z);
}

SEXP
APLTRANSPOSE(SEXP a, SEXP x, SEXP sa, SEXP sz, SEXP rz)
{
    int i, j, na=1, nz=1, ra = length (sa), lsz = length( sz ), nProtected=0;
    SEXP ivec, jvec, z, itel, nind;
    for( i=0;i<ra ;i++){ na *= INTEGER( sa )[i]; }
    for( i=0;i<lsz;i++){ nz *= INTEGER( sz )[i]; }
    PROTECT(itel = allocVector( INTSXP,              1  ) ); ++nProtected;
    PROTECT(nind = allocVector( INTSXP,              1  ) ); ++nProtected;
    PROTECT(ivec = allocVector( INTSXP,  INTEGER(rz)[0] ) ); ++nProtected;
    PROTECT(jvec = allocVector( INTSXP,             ra  ) ); ++nProtected;
    PROTECT(z    = allocVector( REALSXP,            nz  ) ); ++nProtected;
```

```c
    for( i = 0; i < nz; i++ ){
        INTEGER( itel )[0] = i + 1;
        ivec = APLENCODE( itel, sz );
        for( j = 0; j < ra; j++ ){
            INTEGER( jvec )[j] = INTEGER( ivec )[INTEGER( x )[j] - 1];
        }
        nind = APLDECODE( jvec, sa );
        REAL( z )[i] = REAL( a )[INTEGER(nind)[0] - 1];
    }
    UNPROTECT( nProtected );
    return z;
}

SEXP
APLREDUCE(SEXP f, SEXP a, SEXP k, SEXP sa, SEXP sz, SEXP env)
{
    int i, j, u, v, r, kk, na=1, nz=1, nProtected = 0;
    int nk = length( k ), ra = length( sa ), rz = length( sz );
    SEXP ivec, kvec, ind, z,itel, nind, R_fcall= R_NilValue;
    SEXP Z = R_NilValue, A = R_NilValue;
    for( i=0;i<ra;i++){ na *= INTEGER( sa )[i]; }
    for( i=0;i<rz;i++){ nz *= INTEGER( sz )[i]; }
    kk = ra-nk;
    PROTECT( R_fcall= lang3(f, R_NilValue, R_NilValue) ); ++nProtected;
    PROTECT( Z      = allocVector( REALSXP,       1  ) ); ++nProtected;
    PROTECT( A      = allocVector( REALSXP,       1  ) ); ++nProtected;
    PROTECT( itel   = allocVector( INTSXP,        1  ) ); ++nProtected;
    PROTECT( ivec   = allocVector( INTSXP,        ra ) ); ++nProtected;
    PROTECT( kvec   = allocVector( INTSXP,        kk ) ); ++nProtected;
    PROTECT( ind    = allocVector( INTSXP,        nz ) ); ++nProtected;
    PROTECT( z      = allocVector( REALSXP,       nz ) ); ++nProtected;
    for( i = 0; i < nz; i++ ){
        INTEGER( ind )[i] = 0;
    }
    for( i = 0; i < na; i++ ){
        INTEGER( itel )[0] = i + 1;
        ivec = APLENCODE( itel, sa );
        u = 0;
        for( j = 0; j < ra; j++ ){
            r = 0;
            for( v = 0; v < nk; v++ ){
                if( j == ( INTEGER( k )[v] - 1 ) ) r = 1;
            }
            if( r == 0 ){
                INTEGER( kvec )[u] = INTEGER( ivec )[j];
                u += 1;
            }
        }
        nind = APLDECODE( kvec, sz );
        if ( INTEGER( ind )[INTEGER( nind )[0] - 1] == 0 ) {
            REAL( z )[INTEGER( nind )[0] - 1] = REAL( a )[i];
            INTEGER( ind )[INTEGER( nind )[0] - 1] = 1;
        } else {
            REAL( Z )[0] = REAL( z )[INTEGER( nind )[0] - 1];
            REAL( A )[0] = REAL( a )[i];
            SETCADR( R_fcall, Z );
            SETCADDR( R_fcall, A );
            REAL( z )[INTEGER( nind )[0] - 1] = REAL( eval( R_fcall, env ) )[0];
        }
    }
    UNPROTECT( nProtected );
    return z;
}

SEXP
  APLSCAN( SEXP f, SEXP a, SEXP k, SEXP sa, SEXP ra, SEXP env )
  {
    int i, sk, l, na=1, ka = INTEGER( ra )[0],nProtected=0;
    for( i=0;i<ka ;i++ ){ na *= INTEGER( sa )[i]; }
    SEXP ivec, z, itel, nind, Z, A,R_fcall=R_NilValue;
    PROTECT( R_fcall = lang3( f, R_NilValue, R_NilValue ) ); ++nProtected;
    PROTECT( itel    = allocVector( INTSXP,          1  ) ); ++nProtected;
    PROTECT( nind    = allocVector( INTSXP,          1  ) ); ++nProtected;
    PROTECT( Z       = allocVector( REALSXP,         1  ) ); ++nProtected;
    PROTECT( A       = allocVector( REALSXP,         1  ) ); ++nProtected;
    PROTECT( ivec    = allocVector( INTSXP,          ka ) ); ++nProtected;
    PROTECT( z       = allocVector( REALSXP,         na ) ); ++nProtected;
    l = INTEGER( k )[0] - 1;
    for( i = 0; i < na; i++ ){
      INTEGER( itel )[0] = i + 1;
      ivec = APLENCODE( itel, sa );
      sk = INTEGER( ivec )[l];
      if( sk == 1 ){
        REAL( z )[i] = REAL( a )[i];
      }else{
        INTEGER( ivec )[l] -= 1;
        nind = APLDECODE( ivec, sa );
        REAL( Z )[0]=REAL( z )[INTEGER( nind )[0]-1];
        REAL( A )[0]=REAL( a )[i];
        SETCADR( R_fcall, Z );
        SETCADDR( R_fcall, A );
        REAL( z )[i] = REAL( eval( R_fcall, env ) )[0];
      }
    }
    UNPROTECT( nProtected );
    return z;
  }

SEXP
APLINNERPRODUCT(SEXP f, SEXP g, SEXP a, SEXP b, SEXP sa, SEXP sb, SEXP sz, SEXP ns, SEXP
{
    int i, j, u, nz = 1, nProtected = 0;
    int ra = length( sa ),rb = length( sb ), rz = length( sz );
    SEXP ivec, jvec, kvec, z, A, B, Z, itel, k, l, t;
    SEXP R_fcall = R_NilValue, R_gcall = R_NilValue;
    for( i = 0; i < rz; i++ ){ nz *= INTEGER( sz )[i]; }
    PROTECT(R_fcall = lang3( f, R_NilValue, R_NilValue ) ); ++nProtected;
    PROTECT(R_gcall = lang3( g, R_NilValue, R_NilValue ) ); ++nProtected;
    PROTECT( itel   = allocVector( INTSXP,          1  ) ); ++nProtected;
    PROTECT( k      = allocVector( INTSXP,          1  ) ); ++nProtected;
    PROTECT( l      = allocVector( INTSXP,          1  ) ); ++nProtected;
    PROTECT( ivec   = allocVector( INTSXP,          rz ) ); ++nProtected;
    PROTECT( jvec   = allocVector( INTSXP,          ra ) ); ++nProtected;
    PROTECT( kvec   = allocVector( INTSXP,          rb ) ); ++nProtected;
    PROTECT( z      = allocVector( REALSXP,         nz ) ); ++nProtected;
    PROTECT( t      = allocVector( REALSXP,         1  ) ); ++nProtected;
    PROTECT( Z      = allocVector( REALSXP,         1  ) ); ++nProtected;
    PROTECT( B      = allocVector( REALSXP,         1  ) ); ++nProtected;
    PROTECT( A      = allocVector( REALSXP,         1  ) ); ++nProtected;
    for( i = 0; i < nz; i++ ){
        INTEGER( itel )[0] = i + 1;
        ivec = APLENCODE( itel, sz );
        for( j = 0; j < INTEGER( ns )[0]; j++ ){
            for( u = 0; u < ra - 1; u++ ){
                INTEGER( jvec )[u] = INTEGER( ivec )[u];
            }
            INTEGER( jvec )[ra - 1] = j + 1;
            k = APLDECODE( jvec, sa );
            for( u = 1; u < rb; u++ ){
                INTEGER( kvec )[u] = INTEGER( ivec )[ra + u - 2];
            }
            INTEGER( kvec )[0] = j + 1;
            l = APLDECODE( kvec,sb );
            REAL( A )[0] = REAL( a )[INTEGER( k )[0]-1];
            REAL( B )[0] = REAL( b )[INTEGER( l )[0]-1];
            SETCADR( R_fcall, A );
            SETCADDR( R_fcall, B );
            REAL( t )[0] = REAL( eval( R_fcall, env ) )[0];
            if( j == 0 ){
                REAL( z )[i] = REAL( t )[0];
            } else {
                REAL( Z )[0] = REAL( z )[i];
                SETCADR( R_gcall, t );
                SETCADDR( R_gcall, Z );
                REAL( z )[i] = REAL( eval( R_fcall, env ) )[0];
            }
        }
    }
    UNPROTECT( nProtected );
    return z;
}
```

5.6 Rcpp code

5.6.1 hpp

```
#ifndef _apl_RCPP_H
#define _apl_RCPP_H

/* The Rcpp header takes care of everything you should need. */
#include <Rcpp.h>

/*
 * note : RcppExport is an alias to `extern "C"` defined by Rcpp.
 *
 * It gives C calling convention to the rcpp_hello_world function so that
 * it can be called from .Call in R. Otherwise, the C++ compiler mangles the
 * name of the function and .Call can't find it.
 *
 * It is only useful to use RcppExport when the function is intended to be called
 * by .Call. See the thread http://thread.gmane.org/gmane.comp.lang.r.rcpp/649/focus=6
 * on Rcpp-devel for a misuse of RcppExport
 */
//RcppExport SEXP rcpp_hello_world() ;
```

```
RcppExport SEXP APLDECODErcpp( SEXP, SEXP );
RcppExport SEXP APLENCODErcpp( SEXP, SEXP );
RcppExport SEXP APLSELECTrcpp( SEXP, SEXP, SEXP );
RcppExport SEXP APLTRANSPOSErcpp( SEXP, SEXP, SEXP, SEXP, SEXP );
RcppExport SEXP APLSCANrcpp( SEXP, SEXP, SEXP, SEXP, SEXP );
RcppExport SEXP APLREDUCErcpp( SEXP, SEXP, SEXP, SEXP, SEXP, SEXP );
RcppExport SEXP APLINNERPRODUCTrcpp( SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEXP, SEX

#endif
```

5.6.2 cpp

```
//
#include "aplRcpp.hpp"
using namespace Rcpp ;

SEXP
APLDECODErcpp( SEXP cell_r, SEXP dims_r )
{
    IntegerVector cell( cell_r );
    IntegerVector dims( dims_r );
    int n = dims.size(), aux = 1;
    IntegerVector ind( 1 ); ind[0] = 1;
    for( int i = 0; i < n; i++ ) {
        ind[0] += aux * ( cell[i] - 1 );
        aux    *= dims[i];
    }
    return wrap( ind );
}

SEXP
APLENCODErcpp( SEXP ind_r, SEXP dims_r )
{
    IntegerVector dims( dims_r );
    IntegerVector  ind(  ind_r );
    int  n = dims.size(), aux = ind[0], pdim = 1;
    IntegerVector cell( n );
    for( int i = 0; i < n - 1; i++ ){
        pdim *= dims[i];
    }
    for( int i = n - 1; i > 0; i-- ){
            cell[i] = ( aux - 1 ) / pdim;
            aux -= pdim * cell[i];
        pdim /= dims[i - 1];
        cell[i] += 1;
    }
      cell[0] = aux;
    return wrap( cell );
}

SEXP
APLSELECTrcpp( SEXP a_r, SEXP dima_r, SEXP list_r )
{
   IntegerVector dima( dima_r );
   NumericVector    a(    a_r );
   int  r = dima.size(), lz = 1, dimzi;
   List list( list_r );
   IntegerVector  dimz( r );
   IntegerVector  cell( r );
   IntegerVector  czll( r );
   IntegerVector  itel( 1 );
   IntegerVector  nind( 1 );
   for( int i = 0; i < r; i++ ){
       dimzi = as<IntegerVector>( list[i] ).size();
       dimz[i] = dimzi;
       lz *= dimzi;
   }
   NumericVector z(lz);
   for( int i = 0; i < lz; i++ ){
       itel[0] = i + 1;
       cell = APLENCODErcpp( wrap( itel ), wrap( dimz ) );
       for ( int j = 0; j < r; j++ ) {
           IntegerVector listj= as<IntegerVector>( list[j] );
           czll[j] = listj[cell[j] - 1];
       }
       nind = APLDECODErcpp( wrap( czll ), wrap( dima ) );
       z[i] = a[nind[0] - 1];
   }
   return wrap( z );
}

SEXP
APLTRANSPOSErcpp( SEXP a_r, SEXP x_r, SEXP sa_r, SEXP sz_r, SEXP rz_r )
{
    IntegerVector   sa( sa_r ), sz( sz_r );
    int  na = 1, nz = 1, ra = sa.size(), lsz = sz.size();
    IntegerVector   rz( rz_r ), x( x_r ), a( a_r );
    for( int i = 0; i <  ra - 1; i++ ){ na *= sa[i]; }
    for( int i = 0; i < lsz - 1; i++ ){ nz *= sz[i]; }
    IntegerVector itel( 1 ), nind( 1 ), ivec( rz[0] ), jvec( ra );
    NumericVector    z( nz );
    for( int i = 0; i < nz; i++ ){
        itel[0] = i + 1;
        ivec = APLENCODErcpp( wrap( itel ), wrap( sz ) );
        for( int j = 0; j < ra; j++ ){
            jvec[j] = ivec[x[j] - 1];
        }
        nind = APLDECODErcpp( wrap( jvec ), wrap( sa ) );
        z[i] = a[nind[0] - 1];
    }
    return wrap( z );
}

SEXP
APLREDUCErcpp( SEXP f_r, SEXP a_r, SEXP k_r, SEXP sa_r, SEXP sz_r, SEXP env_r )
{
    int u, r, kk, na = 1, nz = 1, nProtected=0;
    IntegerVector  k( k_r ), sa( sa_r ), sz( sz_r );
    NumericVector  a( a_r );
    int nk = k.size(), ra = sa.size(), rz = sz.size();
    SEXP R_fcall= R_NilValue;
    for( int i = 0; i < ra; i++ ){ na *= sa[i]; }
    for( int i = 0; i < rz; i++ ){ nz *= sz[i]; }
    kk = ra - nk;
    PROTECT( R_fcall= Rf_lang3(f_r, R_NilValue, R_NilValue) ); ++nProtected;
    IntegerVector itel( 1 ), nind( 1 ), ind( nz ), ivec( ra ), kvec( kk );
    NumericVector    z( nz );
    for( int i = 0; i < na; i++ ){
        itel[0] = i + 1;
        ivec = APLENCODErcpp( wrap( itel ), wrap( sa ) );
        u = 0;
        for( int j = 0; j < ra; j++ ){
            r = 0;
            for( int v = 0; v < nk; v++ ){
                if( j == ( k[v] - 1 ) ) r = 1;
            }
            if( r == 0 ){
                kvec[u] = ivec[j];
                u += 1;
            }
        }
        nind = APLDECODErcpp( wrap( kvec ), wrap( sz ) );
        if ( ind[nind[0] - 1] == 0 ) {
              z[nind[0] - 1] = a[i];
            ind[nind[0] - 1] = 1;
        } else {
            SETCADR(  R_fcall, wrap( z[nind[0] - 1] ) );
            SETCADDR( R_fcall, wrap( a[i] ) );
            z[nind[0] - 1] = REAL( Rf_eval( R_fcall, env_r ) )[0];
        }
    }
    UNPROTECT( nProtected );
    return wrap( z );
}

SEXP
APLSCANrcpp( SEXP f_r, SEXP a_r, SEXP k_r, SEXP sa_r, SEXP env_r )
{
    IntegerVector  k( k_r ) , sa( sa_r );
    NumericVector  a( a_r );
    int sk, l, na=1, ra = sa.size(), nProtected=0;
    for( int i = 0; i < ra; i++ ){ na *= sa[i]; }
    SEXP R_fcall=R_NilValue;
    PROTECT( R_fcall = Rf_lang3( f_r, R_NilValue, R_NilValue ) ); ++nProtected;
    IntegerVector  itel( 1 ), nind( 1 ), ivec( ra );
    NumericVector  z( na );
    l = k[0] - 1;
    for( int i = 0; i < na; i++ ){
        itel[0] = i + 1;
        ivec = APLENCODErcpp( itel, sa );
        sk = ivec[l];
        if( sk == 1 ){
             z[i] = a[i];
        } else {
            ivec[l] -= 1;
            nind = APLDECODErcpp( wrap( ivec ), wrap( sa ) );
            SETCADR(  R_fcall, wrap( z[nind[0]-1] ) );
            SETCADDR( R_fcall, wrap( a[i] ) );
            z[i] = REAL( Rf_eval( R_fcall, env_r ) )[0];
        }
    }
    UNPROTECT( nProtected );
    return wrap( z );
}
```

```
SEXP
APLINNERPRODUCTrcpp(SEXP f_r, SEXP g_r, SEXP a_r, SEXP b_r, SEXP sa_r, SEXP sb_r, SEXP s
{
    int nz = 1, nProtected = 0;
    IntegerVector  sa( sa_r ) , sb( sb_r ) , sz( sz_r ), ns( ns_r );
    NumericVector  a( a_r ), b( b_r );
    int ra = sa.size(),rb = sb.size(), rz = sz.size();
    SEXP R_fcall = R_NilValue, R_gcall = R_NilValue;
    for( int i = 0; i < rz; i++ ){ nz *= sz[i]; }
    PROTECT( R_fcall = Rf_lang3( f_r, R_NilValue, R_NilValue ) ); ++nProtected;
    PROTECT( R_gcall = Rf_lang3( g_r, R_NilValue, R_NilValue ) ); ++nProtected;
    IntegerVector itel( 1  ), k( 1  ), l( 1  ), ivec( rz ), jvec( ra ), kvec( rb );
    NumericVector  z( nz ), t( 1  ), Z( 1  ), B( 1  ), A( 1  );
    for( int i = 0; i < nz; i++ ){
        itel[0] = i + 1;
        ivec = APLENCODErcpp( itel, sz );
        for( int j = 0; j < ns[0]; j++ ){
            for( int u = 0; u < ra - 1; u++ ){
                jvec[u] = ivec[u];
            }
            jvec[ra - 1] = j + 1;
            k = APLDECODErcpp( jvec, sa );
            for( int u = 1; u < rb; u++ ){
                kvec[u] = ivec[ra + u - 2];
            }
            kvec[0] = j + 1;
            l = APLDECODErcpp( kvec,sb );
            SETCADR(  R_fcall, wrap( a[k[0]-1]) );
            SETCADDR( R_fcall, wrap( b[l[0]-1] ));
            t[0] = REAL( Rf_eval( R_fcall, env_r ) )[0];
            if( j == 0 ){
                z[i] = t[0];
            } else {
                SETCADR(  R_gcall, wrap( t[0] ) );
                SETCADDR( R_gcall, wrap( z[i] ) );
                z[i] = REAL( Rf_eval( R_gcall, env_r ) )[0];
            }
        }
    }
    UNPROTECT( nProtected );
    return wrap( z );
}
```

6 NEWS

001 03/04/16

- First release

002 03/04/16

- Make pdf, using Code2000 Unicode font
- fixed bug in aplDrop()

003 03/05/16

- fixed bug in aplExpand()
- examples and text drop, expand, decode, encode
- aplExpand() can now have logical or binary second argument

004 03/05/16

- bug corrected in APLINNERPRODUCT (.Call() interface).
- writeup and examples aplExpand()
- writeup and examples aplInnerProduct()
- bug corrected in aplJoin()
- writeup and examples aplJoin()
- writeup and example aplMemberOf()
- changed aplMemberOf() so it applies to vectors and numbers of different types
- added R glue code for various interfaces

005 03/06/16

- writeup and examples aplOuterProduct()
- writeup and examples aplRavel()
- writeup and examples aplRank()
- writeup and examples aplReduce()
- writeup and examples aplReplicate()
- writeup and examples aplReshape()
- added aplRcpp.hpp to code section

006 03/06/16

- writeup and examples aplGet() and aplSet()

- writeup and examples aplShape()

007 03/07/16

- writeup and examples aplRotate()
- fixed small bug, changed some variable names in aplRotate()
- removed some unused variables from aplDotC.c
- fixed bug in APLSCAN in aplDotCall.c

008 03/07/16

- writeup and examples aplScan()
- correct small bug in aplReplicate()
- writeup and examples aplSelect()
- correct bug in aplTake()
- writeup and examples aplTake()
- writeup and examples aplTranspose()

009 03/08/16

- number_sections in pdf version
- various typos corrected

References

Becker, R.A., J.M. Chambers, and A.R. Wilks. 1988. The New S Language: a Programming Environment for Data Analysis and Graphics. Wadsworth.

Chen, H., and W.-M. Ching. 2013. "ELI: a simple system for Array Programming." Vector 26 (1). http://archive.vector.org.uk/art10501180.

Helzer, G. 1989. An Encyclopedia of APL. Second. St. Albans, G.B.: I-APL LTD.

IBM. 1988. APL2 Programming: Language Reference. Fourth. San Jose, California: IBM.

Iverson, K. 1962. A Programming Language. Wiley.