APL IN R

JAN DE LEEUW

ABSTRACT. R versions of the array manipulation functions of APL are given. We do not translate the system functions or other parts of the runtime. Also, the current version has no nested arrays.

CONTENTS

Date: August 31, 2008 — 16h 52min — Typeset in TIMES ROMAN.

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 far fewer 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 ⍴A and the rank ⍴⍴A are defined. Scalars are arrays with shape equal to one, vectors are arrays with rank equal to one.

In 1994 I coded most APL array operations in XLISP-STAT. The code is still available at http://idisk.mac.com/jdeleeuw-Public/utilities/apl/array.lsp. 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. The 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 A←3 3 3⍴⍳27 and array(1:27,rep(3,3)) are different. But what is really helpful in linking the two environments is that ,A←3 3 3⍴⍳27 and as.vector(array(1:27,rep(3,3)), which both ravel the array to a vector, give the same result, the vector ⍳27 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 ⍴"aa" 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 2⍴"aa", which gives the same result as 2 2 2⍴"a" or 2 2 2⍴'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 ⍴A and a rank ⍴⍴A, but also a depth. The depth of an array of numbers of 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. For APL-I we use Helzer [1989] and for APL2 we use IBM [1988] and MicroAPL [2007].

2. FUNCTIONS

2.1. **Compress.**

Compress [IBM, 1988, p. 91–92] is Replicate L/R in the special case that L is binary. So look under Replicate.

2.2. **Decode.**

2.2.1. *Definition. Decode* [IBM, 1988, p. 94] The dyadic operator L⊥R is known as *Base Value* [Helzer, 1989, p. 17-21] in APL-I.

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.

If the arguments L and R are vectors of the same length. In the array context, decode returns the index of element x[b] in an array x with dim(x)=a. 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.

2.2.2. *R code.*

```
aplDecode<-function(ind,base) {
    if (length(base) == 1)
        base<-array(base,aplShape(ind))
    b<-c(1,butLast(cumprod(base)))
    return(1+sum(b*(ind-1)))
}
```

2.2.3.

2.3. **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))
}
```

2.4. **Encode.** Encode `A⊤B` is the inverse of Decode. In `APL-I` it is known as Representation [Helzer, 1989, 17–21]. It takes a radix vector `A` and a number `B` and returns the array indices corresponding to cell `B` in an array with a `⍴` of `A`.

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

2.5. **Expand.**

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

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

2.6. **Inner Product.**

```
aplIPV<-function(x,y,f="*",g="+"){
if (length(x) != length(y))
    stop("Incorrect vector length in aplIPV")
if (length(x) == 0) return(x)
z<-match.fun(f)(x,y)
return(aplRDV(z,g))
}
```

```
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("non-compatible array dimensions in aplInner")
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)]
        z[i]<-gg(z[i],ff(aa,bb))
        }
    }
```

```
return(z)
}
```

2.7. **Join.**

```
aplJN<-function(a,b,k) {
    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 aplJN")
    if (!identical(sa[-k],sb[-k]))
        stop("Shape error in aplJN")
    sz<-sa; sz[k]<-sz[k]+sb[k]; nz<-prod(sz); u<-unit(k,
       ra)
    z<-array(0,sz)
    for (i in 1:nz) {
        ivec<-aplEncode(i,sz)
        if (ivec[k] <= sa[k]) z[i]<-a[aplDecode(ivec,sa)]
            else z[i]<-b[aplDecode(ivec-sa[k]*u,sb)]
        }
return(z)
}
```

2.8. **Member Of.**

```
aplMemberOf<-function(a,b) {
    if (!identical(typeof(a),typeof(b)))
        warning("Arguments of different types in aplMO")
    arrTest(a); arrTest(b)
    sa<-aplShape(a); sb<-aplShape(b)
    na<-prod(sa); nb<-prod(sb)
    z<-array(0,sa)
    for (i in 1:na) {
        z[i]<-0; aa<-a[i]
        for (j in 1:nb)
            if (identical(aa,b[j])) z[i]<-1
```

```
        }
return(z)
}
```

2.9. **Outer Product.**

```
aplOP<-function(x,y,f="*") return(outer(x,y,f))
```

2.10. **Ravel.**

```
aplRV<-function(a) as.vector(a)
```

2.11. **Rank.**

```
aplRank<-function(a) aplShape(aplShape(a))
```

2.12. **Reduce.**

```
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)
}
```

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

```
    }
return(z)
}
```

2.13. **Replicate.**

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

```
aplReplicate<-function(x,y,k) {
    if (is.vector(x)) return(aplRPV(x,y))
    sx<-aplShape(x); sy<-aplShape(y); sk<-sx[k]
    if (sy == 1) y<-rep(y,sk)
    if (length(y) != sk)
        stop("Length Error in aplRPV")
    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)
}
```

2.14. **Reshape.**

```
aplReshape<-function(a,d) return(array(a,d))
```

2.15. **Rotate.** Rotate [Helzer, 1989, p. 191–193] shifts the elements of a vector or array dimension. In APL we write AΦB.

```
aplRTV<-function(a,k) {
    n<-aplShape(a)
    if (k == 0) return(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)]))
}
```

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

2.16. **Scan.**

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

```
aplSC<-function(a,k,f="+") {
if (is.vector(a)) return(aplSCV(a,f))
sa<-aplShape(a); ra<-aplRank(a); sk<-sa[k]; u<-unit(k,ra)
ff<-match.fun(f)
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)
}
```

2.17. **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)
}
```

2.18. **Shape.** Shape is the monadic version of ⍴, while reshape is the dyadic version. Shape gives the dimensions of an array, an Reshape modifies them. `aplRank()` is not really a standard APL function, but we use it as shorthand for ⍴⍴A.

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

2.19. **Take.**

```
aplTK<-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))
}
```

2.20. **Transpose.** APL has both a monadic ⍉A and a dyadic B⍉A transpose. This APL transpose has a somewhat tortuous relationship with R's `aperm()`.

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).

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`. For obvious reasons dyadic transpose is not used a great deal.

```
aplTranspose<-function(a,x=rev(1:aplRank(a))) {
sa<-aplShape(a); ra<-aplRank(a)
if (length(x) != ra)
    stop("Length Error in aplTranspose")
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)
for (i in 1:nz)
    z[i]<-a[aplDecode(aplEncode(i,sz)[x],sa)]
return(z)
}
```

3. Utilities

```
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)
```

References

G. Helzer. *An Encyclopedia of APL*. I-APL LTD, St. Albans, G.B., second edition, 1989.

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

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

MicroAPL. *APLX Language Manual*. MicroAPL Ltd., Version 4 edition, November 2007.

DEPARTMENT OF STATISTICS, UNIVERSITY OF CALIFORNIA, LOS ANGELES, CA 90095-1554

E-mail address, Jan de Leeuw: deleeuw@stat.ucla.edu

URL, Jan de Leeuw: http://gifi.stat.ucla.edu