Multidimensional Array Indexing and Storage

Jan de Leeuw

Version 27, August 31, 2017

Abstract

We give R, C, and R->C code to access lineary stored multidimensional arrays and compactly stored multidimensional super-symmetric arrays.

Contents

Note: This is a working paper which will be expanded/updated frequently. All suggestions for improvement are welcome. The directory deleeuwpdx.net/pubfolders/indexing has a pdf version, the bib file, the complete Rmd file with the code chunks, and the R and C source code.

1 Introduction

Suppose A is an array of *rank* r and *dimension* $(n_1, \cdots, n_r)$. Here rank is used in the APL sense, i.e. the length of the dimension vector.

A has elements $a_{i_1\cdots i_m}$. It is easy enough to visualize, or at least conceptualize, these $n := n_1 \times \cdots n_m$ elements in an m-dimensional rectangular block of cells, but unfortunately such blocks cannot be stored directly in contiguous locations in computer memory. The array A is usually stored in n contiguous locations in memory, i.e. in a form that corresponds with a vector, an array of rank one.

Say b is a pointer to the first element stored. We need a mapping f from $\mathcal{I}_{n_1} \otimes \cdots \mathcal{I}_{n_m}$ into $\mathcal{I}_n$ such that $b + f(i_1, \cdots, i_m)$ is a pointer to $a_{i_1\cdots i_m}$. Or, using C, $b[f(i_1, \cdots, i_m)] = *(b + f(i_1, \cdots, i_m))$ must be equal to $a_{i_1\cdots i_m}$. Let's call such an f an *index mapping*. We are interested in both the index mapping f and its inverse $f^{-1} : \mathcal{I}_n \rightarrow \mathcal{I}_{n_1} \otimes \cdots \mathcal{I}_{n_m}$.

If A is symmetric, or anti-symmetric, or triangular, or hollow, then it is not necessary to store all its elements. This means that f is no longer bijective, because fewer than n elements are stored in memory. In fact, the main motivation for this paper is to derive the index mapping, and its inverse, for the super-symmetric arrays of partials derivatives discussed in De Leeuw (2017).

In this paper I review and program index mappings for arrays that can be used to assess and retrieve array elements from contiguous memory locations. I am 99.99% sure there is nothing new in the paper, I just wanted to collect these results in a single convenient place, and provide code. As for the programming languages, we first develop in R, and then translate to C.

2 Matrices

2.1 General Rectangular Matrices

Storing the elements of a general rectangular matrix in contiguous locations in memory can be done in many ways, but two specific ones are the most obvious. We can store row one first, then continue with row two, and so on. This is *row major order*. Or we can store column one, append column two, and so on. That is *column major order*. If A is

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

then column major order stores the matrix as

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

and row major order stores it as

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

The index mapping for column major order is

```
##  1  1  ***   1
```

```
##  2  1  ***   2
##  3  1  ***   3
##  4  1  ***   4
##  1  2  ***   5
##  2  2  ***   6
##  3  2  ***   7
##  4  2  ***   8
##  1  3  ***   9
##  2  3  ***  10
##  3  3  ***  11
##  4  3  ***  12
```

and for row major order it is

```
##  1  1  ***   1
##  1  2  ***   2
##  1  3  ***   3
##  2  1  ***   4
##  2  2  ***   5
##  2  3  ***   6
##  3  1  ***   7
##  3  2  ***   8
##  3  3  ***   9
##  4  1  ***  10
##  4  2  ***  11
##  4  3  ***  12
```

Because the row index changes fastest, we could call column major ordering *first-fast*. Row major ordering has the second, i.e. the last index changing fastest, and could therefore be called *last-fast*. That terminology has the advantage it generalizes to multidimensional arrays, while the use of rows and columns is more or less limited to matrices.

The main application we have in mind is to pass matrices and arrays from R to C, with I/O and memory allocation done in R, and computation in C. This implies that the C versions of our indexing routines are the most useful ones.

R stores matrices first-fast (showing its origins, as a REPL interface to FORTRAN routines). Thus then `a[3] =`8 and `a[11] =`5, because `as.vector(a)` is

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

First-fast is how the arrays arrive in C, where the index calculations are done to retrieve the elements. As a consequence, both for matrices and arrays, first-fast is the most relevant storage mode for our purposes. We do not have to worry about how C stores matrices multidimensional arrays, because for our purposes it is better to pretend that C does not have any matrices or multidimensional arrays at all.

2.2 Symmetric and Triangular Matrices

For symmetric and anti-symmetric matrices we only have to store half of the elements. This means making two choices, first if we want to store the upper or the lower triangle, second if we want to use first-fast or last-fast. Storing the lower triangle means storing all a_{ij} with $i \geq j$, storing the upper triangle stores all a_{ij} with $i \leq j$. Because of generalization to multidimensional arrays we call the first option *decreasing* and the second option *increasing*.

We will choose the upper or increasing triangle combined with first-fast storage throughout. For a symmetric matrix we only have to store $N = \frac{1}{2}n(n+1)$ elements. The mapping function will be defined for all $i \leq j$, and it maps those index pairs bijectively on $1, \cdots, N$. This mapping function also has an inverse. We then extend the mapping function to all index pairs by first ordering the pair, making the extended mapping function surjective. We will not give the mapping function yet, because it will be just a special case of the one for super-symmetric arrays.

Just as an aside, our conventions imply that the elements of a lower triangular matrix, in which the non-zero a_{ij} have $i \geq j$, will be stored in increasing or upper triangle form, in other words as its transpose. The extended mapping function for that case, which we do not really consider in detail, will have to take that into account. Also, strictly lower and upper triangular matrices, anti-symmetric matrices, and symmetric hollow matrices such as distance matrices, only need to store $\frac{1}{2}n(n-1)$ elements. We do not discuss storage schemes that leave out the diagonal. We always store the diagonal elements, but the extended mapping function has no need to access them.

3 Arrays

3.1 General Arrays

Linear storage of the elements of a general array or rank m and dimension $(n_1, \cdots, n_m)$ can be done in many ways. First-fast and last-fast are just two special cases of the $m!$ ways in which we can nest the index loops. Since going through all possibilities is both boring and unnessary we limit ourselves to first-fast. If, for whatever reason, another permutation of the indices is needed we can first transpose the array in R, for instance using `aperm()` from the base package, or `aplTranspose()` from De Leeuw and Yajima (2016).

The R version of the mapping function is `fArrayFirst()`. Note that it is identical to the `aplDecode()` function from De Leeuw and Yajima (2016). For completeness we also give `fArrayFirstInverse()`, which is `aplEncode()`, the inverse f^{-1}, mapping $\mathcal{I}_n$ into $\mathcal{I}_{n_1} \otimes \cdots \mathcal{I}_{n_m}$. Our example is a $4 \times 3 \times 2$ array. In tabular form the index mapping is

```
##  1  1  1  ***   1
##  2  1  1  ***   2
##  3  1  1  ***   3
##  4  1  1  ***   4
##  1  2  1  ***   5
##  2  2  1  ***   6
```

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

and in array form it is

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

3.2 Super-symmetry

An array A of rank m and dimension $(n, \cdots, n)$ is *super-symmetric* if $a_{i_1 \cdots i_m}$ is invariant under permutation of indices. Think of arrays of product moments, cumulants, or partial derivatives. Super-symmetry generalizes symmetry of matrices.

In order not to drown in the $(m!) \times (m!)$ combinations of what-fast and which-triangle, we limit ourselves to first-fast and increasing (i.e. index tuples with $i_1 \leq \cdots \leq i_m$).

The number of elements we have to store is $\binom{m+n-1}{m}$, instead of n^m. This can be a considerable

saving of storage if m and/or n are at all large. For $n = 10$ and $m = 4$, for example, we save 93% of the storage. For large n the savings are approximately $100 * (1 - \frac{1}{m!})\%$.

The mapping function for a super-symmetric array, with first-fast and increasing index vectors, is `fSupSymIncreasingFirst()`. Note that this function does not need to know the dimensions of the array. Also it starts by sorting the elements of the index vector. Thus

```
fSupSymIncreasingFirstRC(c(1,2,2,3))
```

```
## [1] 8
```

and also

```
fSupSymIncreasingFirstRC(c(2,1,3,2))
```

```
## [1] 8
```

The inverse mapping is `fSupSymIncreasingFirstInverse()`.

As an example, here is the result for a four-dimensional super-symmetric array of order four. There are only 35, instead of 256, elements to consider.

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

```
##  2  2  4  4  ***  28
##  1  3  4  4  ***  29
##  2  3  4  4  ***  30
##  3  3  4  4  ***  31
##  1  4  4  4  ***  32
##  2  4  4  4  ***  33
##  3  4  4  4  ***  34
##  4  4  4  4  ***  35
```

4 Implementation

There are four versions of the two basic functions `fArrayFirst()` for general arrays and `fSupSymIncreasingFirst()` for super-symmetric arrays. Two are in C, one which passes by value and returns integers, and a second one which wraps the first one, and passes by reference and returns void. And two are in R, one in pure R, and one which uses .C() to wrap a call to the compiled C code.

The two inverse functions `fArrayFirstInverse()` and `fSupSymIncreasingFirstInverse()` have only a single C version, which passes by reference and returns void. There are again two R versions of each, one in pure R and one a wrapper around a .C() call to the compiled C code.

5 Appendix: Code

5.1 R Code

5.1.1 indexing.R

```
fArrayFirstR <- function (cell, dimension) {
  rank <- length (cell)
  k <- cumprod (c(1, dimension))[1:rank]
  return (1 + sum ((cell - 1) * k))
}

fSupSymIncreasingFirstR <- function (cell) {
  f <- function (r)
    choose (r + (cell[r] - 1) - 1, r)
  rank <- length (cell)
  cell <- sort (cell)
  return (1 + sum (sapply (1:rank, f)))
}

fArrayFirstInverseR <- function(index, dimension) {
  rank <- length (dimension)
```

```
  b <- cumprod (c(1, dimension))[1:rank]
  r <- rep(0, length(b))
  s <- index - 1
  for (j in rank:1) {
    r[j] <- s %/% b[j]
    s <- s - r[j] * b[j]
  }
  return(1 + r)
}

fSupSymIncreasingFirstInverseR <- function (dimension, rank, index) {
  last.true <- function (x) {
    w <- which (x)
    return (w[length(w)])
  }
  a <- rep (0, rank)
  v <- index - 1
  for (k in rank:1) {
    s <- choose (k + (0:(dimension - 1)) - 1, k)
    u <- last.true (v >= s)
    a[k] <- u
    v <- v - s[u]
  }
  return (a)
}
```

5.1.2 indexingC.R

```
dyn.load("indexing.so")

fArrayFirstRC <- function (cell, dimension) {
  rank <- length (cell)
  h <-
    .C(
      "fArrayFirstGlue",
      as.integer(cell),
      as.integer(dimension),
      as.integer(rank),
      index = as.integer(0)
    )
  return (h$index)
}
```

```r
fSupSymIncreasingFirstRC <- function (cell) {
  rank <- length(cell)
  h <-
    .C("fSupSymIncreasingGlue",
       as.integer(cell),
       as.integer(rank),
       index = as.integer(0))
  return (h$index)
}

fArrayFirstInverseRC <- function(index, dimension) {
  rank <- length (dimension)
  h <-
    .C(
      "fArrayFirstInverse",
      as.integer(rank),
      as.integer (dimension),
      as.integer(index),
      cell = as.integer(rep(0, rank))
    )
  return (h$cell)
}

fSupSymIncreasingFirstInverseRC <- function (dimension, rank, index) {
  h <-
    .C(
      "fSupSymIncreasingFirstInverse",
      as.integer(dimension),
      as.integer(rank),
      as.integer(index),
      cell = as.integer(rep(0, rank))
    )
  return (h$cell)
}
```

5.2 C Code

5.2.1 indexing.h

```c
#ifndef INDEXING_H
#define INDEXING_H

#include <stdlib.h>
```

```c
static inline int IMIN(const int x, const int y) { return (x < y) ? x : y; }
static inline int VINDEX(const int i) { return i - 1; }

#endif /* INDEXING_H */
```

5.2.2 indexing.c

```c
#include "indexing.h"

int binCoef(const int n, const int m) {
    int *work = (int *)calloc((size_t)m + 1, sizeof(int));
    work[0] = 1;
    for (int i = 1; i <= n; i++) {
        for (int j = IMIN(i, m); j > 0; j--) {
            work[j] = work[j] + work[j - 1];
        }
    }
    int choose = work[m];
    free(work);
    return (choose);
}

int int_cmp(const void *x, const void *y) {
    const int *ix = (const int *)x;
    const int *iy = (const int *)y;
    return (*ix - *iy);
}

int fArrayFirst(const int *cell, const int *dimension, const int rank) {
    int aux = 1, result = 1;
    for (int i = 1; i <= rank; i++) {
        result += (cell[VINDEX(i)] - 1) * aux;
        aux *= dimension[VINDEX(i)];
    }
    return (result);
}

int fSupSymIncreasing(int *cell, const int rank) {
    int result = 1;
    (void)qsort(cell, (size_t)rank, sizeof(int), int_cmp);
    for (int i = 1; i <= rank; i++) {
        int k = i + (cell[VINDEX(i)] - 1) - 1;
        result += binCoef(k, i);
```

```
    }
    return (result);
}

void fArrayFirstInverse(const int *rank, const int *dimension, const int *index,
                        int *cell) {
    int m = *rank, l = *index, b = 1;
    for (int j = 1; j < m; j++) {
        b *= dimension[VINDEX(j)];
    }
    for (int j = m - 1; j > 0; j--) {
        int k = (l - 1) / b;
        cell[j] = k + 1;
        l -= k * b;
        b /= dimension[j - 1];
    }
    cell[VINDEX(1)] = l;
    return;
}

void fSupSymIncreasingFirstInverse(const int *dimension, const int *rank,
                                   const int *index, int *cell) {
    int n = *dimension, m = *rank, l = *index, v = l - 1;
    for (int k = m; k >= 1; k--) {
        for (int j = 0; j < n; j++) {
            int sj = binCoef(k + j - 1, k), sk = binCoef(k + j, k);
            if (v < sk) {
                cell[VINDEX(k)] = j + 1;
                v -= sj;
                break;
            }
        }
    }
    return;
}

void binCoefGlue(const int *n, const int *m, int *choose) {
    *choose = binCoef(*n, *m);
}

void fArrayFirstGlue(const int *cell, const int *dimension, const int *rank,
                     int *result) {
    *result = fArrayFirst(cell, dimension, *rank);
}
```

```
void fSupSymIncreasingGlue(int *cell, const int *rank, int *result) {
    *result = fSupSymIncreasing(cell, *rank);
}
```

References

De Leeuw, J. 2017. "Higher Partials of fStress. Who Needs Them ?" http://deleeuwpdx.net/pubfolders/fStress/fStress.pdf.

De Leeuw, J., and M. Yajima. 2016. "APL in R." doi:10.13140/RG.2.1.2372.0724.