Faster Multivariate Moments

Jan de Leeuw

First created on March 24, 2020. Last update on March 26, 2020

Abstract

Efficient C code, callable from R, is given to compute all multivariate moments and product moments up to a given order.

Note: This is a working paper which will be expanded/updated frequently. All suggestions for improvement are welcome. The directory deleeuwpdx.net/pubfolders/moments 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 we have an $n \times m$ matrix with n observations of m variables. We want to compute all sample raw moments and product moments of order up to r. If we append a variable identically equal to one, then De Leeuw (2012) computes all $(m+1)^r$ product moments, stored in a super-symmetric array, using the outer() function from base R. The R function raw_moments_upto_p() is given in the appendix. It is very wasteful, both in terms of storage, and in terms of computational speed.

2 Improvement

First, for $m+1$ variables we only have to store $\binom{r+m}{r}$ moments, the number of non-negative integer solutions to $x_1 + \cdots + x_{m+1} = r$. For $m = 9$ and $n = 4$, for example, we already save almost 90% of the storage. And we only compute 10% of the elements of the array. De Leeuw (2017) gives a bijection fSupSymIncreasingFirst() that maps n-tuples of integers $(i_1, i_2, \cdots, i_n)$ for which $1 \leq i_1 \leq i_2 \leq \cdots \leq i_n \leq m$, onto the integers $\{1, 2, \cdots, \binom{n+m-1}{n}\}$. And in addition another function fSupSymIncreasingFirstInverse(), which is the inverse of the first.

Second, these indexing functions are programmed in C, and they are called by the C function moments(), which in turn can be called using the .C() interface in R. Again, all C code is given in the appendix. There are also some auxiliary functions that can be used to subtract the mean and/or make the sum of squares of the variables equal to n. Thus sample central moments can also be computed using the moments() function.

The C code is pretty idiosyncratic, because it uses pointer arithmetic instead of array indices. It uses the inline functions VINDEX() and MINDEX() to locate elements of vectors and matrices in linear storage. Also the main functions do not return a value, and all their arguments are passed by reference. This is mainly so they can be called directly from R using the .C() interface. I am not sure if these conventions are good C practice, but I am comfortable with them, and they make it easy to move between R storage/indexing and C storage/indexing.

3 Timing

To give an idea about the gain in computing time by

1. only computing and storing the upper-diagonal elements of the super-symmetric array, and
2. switching to C,

we give a simple artificial example with 10 variables and 10,000 observations. Moments of order up to 2, up to 4, and up to 6 are computed. Note that the super-symmetric array for moments up to order 6 already has 1.771561×10^6 elements, while the C function only computes 8008 elements, a mere 0.45% of the total number of array elements. The ratio of computing times for R and C is 14 for order 2, 23 for order 4, and 161 for order 6. This seems a promising improvement that will also be applied in our method to compute multivariate cumulants (see De Leeuw (2020)).

```
set.seed(12345)
x<-matrix(rnorm(10000),1000,10)
system.time(replicate(100, raw_moments_upto_p(x, 2)))
```

```
##    user  system elapsed
##   0.649   0.023   0.674
```

```
system.time(replicate(100, moments(x, 2)))
```

```
##    user  system elapsed
##   0.043   0.005   0.048
```

```
system.time(replicate(100, raw_moments_upto_p(x, 4)))
```

```
##    user  system elapsed
##   8.554   7.076  15.664
```

```
system.time(replicate(100, moments(x, 4)))
```

```
##    user  system elapsed
##   0.674   0.008   0.683
```

```
system.time(replicate(10, raw_moments_upto_p(x, 6)))
```

```
##    user  system elapsed
## 111.795  19.995 132.052
```

```
system.time(replicate(10, moments(x, 6)))
```

```
##    user  system elapsed
##   0.819   0.002   0.822
```

4 Appendix: Code

4.1 R Code

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

set.seed(12345)

data <- matrix (rnorm(400), 100, 4)

moments <- function (data, order) {
  data <- cbind(1, data)
  nvars <- ncol (data)
  nobs <- nrow(data)
  m <- choose (order + nvars - 1, order)
  h <-
    .C(
      "moments",
      as.double(data),
      as.integer(nobs),
      as.integer(nvars),
      as.integer(order),
      moments = as.double(rep(0, m))
    )
  return (h$moments)
}

center <- function(data) {
  nvars <- ncol (data)
  nobs <- nrow (data)
  h <-
    .C("center",
       as.integer(nobs),
       as.integer(nvars),
```

```
        data = as.double(data))
  return (matrix(h$data, nobs, nvars))
}

normalize <- function(data) {
  nvars <- ncol (data)
  nobs <- nrow (data)
  h <-
    .C("normalize",
       as.integer(nobs),
       as.integer(nvars),
       data = as.double(data))
  return (matrix(h$data, nobs, nvars))
}

raw_moments_upto_p <- function (x, p = 4) {
  n <- nrow (x)
  m <- ncol (x)
  if (p == 1) {
    return (c(1, apply (x, 2, mean)))
  }
  y <- array (0, rep (m + 1, p))
  for (i in 1:n) {
    xi <- c(1, x[i,])
    z <- xi
    for (s in 2:p) {
      z <- outer (z, xi)
    }
    y <- y + z
  }
  return (y / n)
}
```

4.2 C Code

4.2.1 moments.h

```
#ifndef MOMENTS_H
#define MOMENTS_H

#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
```

```c
inline int VINDEX(const int i) { return i - 1; }

inline int MINDEX(const int i, const int j, const int n) {
  return (i - 1) + (j - 1) * n;
}

inline int IMIN(const int a, const int b) { return (a > b) ? b : a; }

extern void moments(const double *, const int *, const int *, const int *, double *);

extern int binCoef(const int, const int);

extern int int_cmp(const void *, const void *);

extern void fSupSymIncreasingFirstInverse(const int *, const int *, const int *,
                                          int *);

#endif /* MOMENTS_H */
```

4.2.2 moments.c

```c
#include "moments.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);
}

void fSupSymIncreasingFirstInverse(const int *dimension, const int *order,
```

```c
                                    const int *index, int *cell) {
  int n = *dimension, m = *order, 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 moments(const double *data, const int *pnobs, const int *pnvars,
             const int *porder, double *moments) {
  int nvars = *pnvars, nobs = *pnobs, order = *porder;
  int nmax = binCoef(order + nvars - 1, order);
  int *cell = (int *)calloc((size_t)order, sizeof(int));
  int *dims = (int *)calloc((size_t)order, sizeof(int));
  for (int i = 1; i <= order; i++) {
    *(cell + VINDEX(i)) = nvars;
    *(dims + VINDEX(i)) = nvars;
  }
  for (int i = 1; i <= nmax; i++) {
    (void)fSupSymIncreasingFirstInverse(dims, porder, &i, cell);
    double s = 0.0;
    for (int k = 1; k <= nobs; k++) {
      double p = 1.0;
      for (int l = 1; l <= order; l++) {
        p *= *(data + MINDEX(k, *(cell + VINDEX(l)), nobs));
      }
      s += p;
    }
    *(moments + VINDEX(i)) = s / ((double)nobs);
  }
  free(dims);
  free(cell);
  return;
}

void center(const int *pnobs, const int *pnvars, double *data) {
  int nvars = *pnvars, nobs = *pnobs;
```

```
  double *s = (double *)calloc((size_t) nvars, sizeof(double));
  for (int j = 1; j <= nvars; j++){
    *(s + VINDEX(j))  = 0.0;
    for (int i = 1; i <= nobs; i++)  {
      *(s + VINDEX(j))  += *(data + MINDEX(i, j, nobs));
    }
    *(s + VINDEX(j))  /= ((double) nobs);
  }
  for (int j = 1; j <= nvars; j++){
    for (int i = 1; i <= nobs; i++)  {
      *(data + MINDEX(i, j, nobs)) -= *(s + VINDEX(j)) ;
    }
  }
  free(s);
  return;
}

void normalize(const int *pnobs, const int *pnvars, double *data) {
  int nvars = *pnvars, nobs = *pnobs;
  double *s = (double *)calloc((size_t) nvars, sizeof(double));
  for (int j = 1; j <= nvars; j++){
    *(s + VINDEX(j))  = 0.0;
    for (int i = 1; i <= nobs; i++)  {
      double dd = *(data + MINDEX(i, j, nobs));
      *(s + VINDEX(j)) += dd * dd;
    }
  }
  for (int j = 1; j <= nvars; j++){
    for (int i = 1; i <= nobs; i++)  {
      *(data + MINDEX(i, j, nobs)) *= sqrt(((double) nobs) / *(s + VINDEX(j))) ;
    }
  }
  free(s);
  return;
}
```

References

De Leeuw, J. 2012. "Multivariate Cumulants in R." UCLA Department of Statistics. http://deleeuwpdx.net/janspubs/2012/notes/deleeuw_U_12a.pdf.

———. 2017. "Multidimensional Array Indexing and Storage." 2017. http://deleeuwpdx.net/pubfolders/indexing/indexing.pdf.

———. 2020. "Faster Multivariate Cumulants." 2020. http://deleeuwpdx.net/pubfolders/cumulants/cumulants.pdf.