Faster Multivariate Cumulants

Jan de Leeuw

First created on February 24, 2020. Last update on April 17, 2020

Abstract

We give C code, with an interface in R, to compute multivariate cumulants of a given order. We use compact storage of the supersymmetric arrays of moments and cumulants throughout, and we use restricted growth functions to compute the necessary set partitions.

Contents

Note: This is a working paper which will be expanded/updated frequently. All suggestions for improvement are welcome. The directory deleeuwpdx.net/pubfolders/cumulants 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 $x_1, \cdots, x_m$ are m random variables, not necessarily distinct. Multivariate cumulants of order r are defined as

$$\kappa(x_1, \cdots, x_r) = \sum_{\text{partitions}} (-1)^{q-1}(q-1)! \prod_{p=1}^{q} \mu(I_p), \tag{1}$$

where the summation is over all partitions of $\{1, 2, \cdots, r\}$ and q is the number of subsets in the partition. Also $\mu(I_p)$ is the raw product-moment (around zero) of the variables in subset I_p of the partition. Thus, for example,

$$\begin{aligned}
\kappa(x_1, x_2) &= \mu_{12} - \mu_1\mu_2, \\
\kappa(x_1, x_2, x_3) &= \mu_{123} - \mu_{12}\mu_3 - \mu_{13}\mu_2 - \mu_{23}\mu_1 + 2\mu_1\mu_2\mu_3, \\
\kappa(x_1, x_2, x_3, x_4) &= \mu_{1234} - \mu_{123}\mu_4 - \mu_{124}\mu_3 - \mu_{134}\mu_2 - \mu_{234}\mu_1 \\
&\quad - \mu_{12}\mu_{34} - \mu_{13}\mu_{24} - \mu_{14}\mu_{23} \\
&\quad + 2\mu_{12}\mu_3\mu_4 + 2\mu_{13}\mu_2\mu_4 + 2\mu_{14}\mu_2\mu_3 + 2\mu_{23}\mu_1\mu_4 \\
&\quad + 2\mu_{24}\mu_1\mu_3 + 2\mu_{34}\mu_1\mu_2 \\
&\quad - 6\mu_1\mu_2\mu_3\mu_4.
\end{aligned}$$

De Leeuw (2012) gives an implementation in R that computes all cumulants of order r for m variables. As (1) shows to compute cumulants of order r we need the raw product moments of order up to r. This is done in De Leeuw (2012) by generating an r-dimensional super-symmetric array with $(m+1)^r$ elements, using the base-R function outer(). The array has $m+1$ rows, columns, slices and so on because we add a variable identically equal to one to get the moments of order less than r.

After computing the moment array, generating the partitions is done by the setparts() function from the partitions package (Hankin (2006), Hankin and West (2007)). The setparts() function, an R wrapper for C++ code, first computes the integer partitions, and subsequently the set partitions corresponding with each integer partition.

In this note we will try to improve the implementation in De Leeuw (2012) of both the moment and partition computations.

2 Improvements

As formula (1) shows, to compute the cumulants of order r for m variables we need the moments and product moments of order $1, \cdots, r$ of the m variables, and we need the partitions of the set $\{1, \cdots, r\}$.

Let's look at the moments first. We use the algorithm from De Leeuw (2020). Instead of computing all $(m+1)^r$ moments $\mu_{i_1 \cdots i_r}$ with $1 \leq i_s \leq m+1$ it suffices to compute those with

$1 \leq i_1 \leq \cdots \leq i_r \leq m+1$. The number of these ordered index tuples is equal to the number of non-negative integer solutions to $x_1 + \cdots + x_{m+1} = r$, which is $\binom{m+r}{r}$. The indexing routines from De Leeuw (2017) are used to store and access this much smaller number of moments in a single vector. Here is an example with $r = 3$ and $m = 3$, where we compute 20 moments instead of $(m+1)^r = 64$.

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

The table below shows the savings in both storage and number of moments computed, for rows $m = 1, \cdots, 20$ and columns $r = 1, \cdots 5$.

```
##        [,1]     [,2]     [,3]     [,4]     [,5]
##  [1,] 0.000000 0.250000 0.500000 0.687500 0.812500
##  [2,] 0.000000 0.333333 0.629630 0.814815 0.913580
##  [3,] 0.000000 0.375000 0.687500 0.863281 0.945312
##  [4,] 0.000000 0.400000 0.720000 0.888000 0.959680
##  [5,] 0.000000 0.416667 0.740741 0.902778 0.967593
##  [6,] 0.000000 0.428571 0.755102 0.912536 0.972511
##  [7,] 0.000000 0.437500 0.765625 0.919434 0.975830
##  [8,] 0.000000 0.444444 0.773663 0.924554 0.978205
##  [9,] 0.000000 0.450000 0.780000 0.928500 0.979980
## [10,] 0.000000 0.454545 0.785124 0.931630 0.981354
## [11,] 0.000000 0.458333 0.789352 0.934172 0.982446
## [12,] 0.000000 0.461538 0.792899 0.936277 0.983334
```

```
## [13,] 0.000000 0.464286 0.795918 0.938047 0.984069
## [14,] 0.000000 0.466667 0.798519 0.939556 0.984687
## [15,] 0.000000 0.468750 0.800781 0.940857 0.985214
## [16,] 0.000000 0.470588 0.802768 0.941991 0.985668
## [17,] 0.000000 0.472222 0.804527 0.942987 0.986063
## [18,] 0.000000 0.473684 0.806094 0.943869 0.986410
## [19,] 0.000000 0.475000 0.807500 0.944656 0.986718
## [20,] 0.000000 0.476190 0.808768 0.945362 0.986991
```

}

Next, the partitions. In this paper we use an algorithm in C based on the restricted growth function (RGF) representation of partitions, taken from section 3.2.1 of Kreher and Stinson (1998).

A restricted growth function (RGF) of length r is a sequence w of positive integers with $w_1 = 1$ and $w_i \leq 1 + \max_{j=1}^{i-1} w_j$ for all $1 < i \leq r$. In the first column of the following table are the 5 RGF's for $r = 3$, generated by the .C() function genrgf(). The second column has the corresponding partitions.

```
##   RGF Partitions
## 1 111 {123}
## 2 112 {12}{3}
## 3 121 {1,3}{2}
## 4 122 {1},{2,3}
## 5 123 {1}{2}{3}
```

Now suppose we have three variables and we want to compute the cumulant κ_{133} of order three. Our moment array is of dimension $(4, 4, 4)$, because we include a variable identically equal to one. We use this variable when looking up moments of order less than three, and we add one to the number of the other variables. The table below shows how this is done.

```
##   RGF Partitions Moments Moment_Lookup Index
## 1 111 {123}      133     244           18
## 2 112 {12}{3}    13,3    124,114       12,11
## 3 121 {1,3}{2}   13,3    124,114       12,11
## 4 122 {1},{2,3}  1,33    112,144       2,17
## 5 123 {1}{2}{3}  1,3,3   112,114,114   2,11,11
```

The C function cumulants() in the appendix computes all cumulants of order r, using the moment array and the rgf table computed by moments() and genrgf(). The R function momentsAndCumulants() calls all three C functions and returns moments of order up to r, rgfs for order r, and cumulants of order r.

3 Comparisons

De Leeuw (2012) has a general function to compute cumulants from raw moments using the partitions package. It also has a function to compute the first four cumulants directly, using the formulas we have given in the introduction to this paper. For the examples analyzed in De Leeuw (2012) the direct computation is much faster than the function that uses the setparts() function.

So to show that our new code is faster than our previous implementation we merely have to show that the direct computation of cumulants and raw moments of a given order (in supersymmetric multidimensional arrays) is slower than using RGF's and compact storage.

We use a data matrix of dimension 10,000 by 5, filled with standard normal random numbers. First the direct computation of cumulants of order four.

```
microbenchmark(make_cumulants(x, 4))
```

```
## Unit: milliseconds
##                   expr        min          lq        mean     median          uq
##  make_cumulants(x, 4) 289.335787 294.7171045 302.9636565 297.363968 304.7643685
##         max neval
##  389.321533   100
```

And now the new implementation. Same data, same order.

```
microbenchmark(momentsAndCumulants(x, 4))
```

```
## Unit: milliseconds
##                        expr      min        lq       mean   median      uq
##  momentsAndCumulants(x, 4) 5.747941 6.1413115 6.74533855 6.469289 6.87288
##        max neval
##  21.397337   100
```

As we see, in this case at least, the new implementation is about 40-50 times faster.

Next, a matrix with standard normals of dimension 10,000 by 10, and cumulants of order three.

```
microbenchmark(make_cumulants(x, 3))
```

```
## Unit: milliseconds
##                   expr        min         lq        mean      median         uq
##  make_cumulants(x, 3) 106.784105 110.103844 113.6849878 111.7400205 114.403651
##         max neval
##  187.804533   100
```

```
microbenchmark(momentsAndCumulants(x, 3))
```

```
## Unit: milliseconds
##                         expr      min       lq       mean    median        uq
##  momentsAndCumulants(x, 3) 1.918033 2.368798 2.75434945 2.4332815 2.5791985
##       max neval
##  15.96741   100
```

Again, about 40-50 times faster.

4 Appendix: Code

4.1 R Code

4.1.1 cumulants.R

```
set.seed(12345)
x <- matrix(rnorm(50000), 10000, 5)

dyn.load("cumulants.so")

tbell = c(1,   1,    2,     5,      15,     52,     203,
          877, 4140, 21147, 115975, 678570, 4213597)

listCells <- function (order, nvar) {
  n <- choose (order + nvar - 1, order)
  cells <- matrix(0, n, order)
  for (k in 1:n) {
    h <- .C(
      "fSupSymIncreasingFirstInverse",
      as.integer (nvar),
      as.integer (order),
      as.integer (k),
      cell = as.integer (rep(0, order))
    )
    cells[k, ] <- h$cell
  }
  return(cells)
}
```

```
moments <- function (data, order, upto = TRUE) {
  if (upto) {
    data <- cbind(1, data)
  }
  nvar <- ncol (data)
  nobs <- nrow (data)
  nmax <- choose (order + nvar - 1, order)
  h <-
    .C(
      "moments",
      as.double(data),
      as.integer(nobs),
      as.integer(nvar),
      as.integer(nmax),
      as.integer(order),
      moments = as.double(rep(0, nmax))
    )
}

genrgf <- function (n) {
  m <- tbell[n + 1]
  h <- .C("genrgf", as.integer (n), rgf = as.integer(rep(0, n * m)))
  return (matrix(h$rgf, m, n))
}

cumulants <- function (data, order, upto = TRUE) {
  if (upto) {
    data <- cbind(1, data)
  }
  nvar <- ncol (data)
  nobs <- nrow(data)
  m <- choose (order + nvar - 1, order)
  h <-
    .C(
      "cumulants",
      as.double(data),
      as.integer(nobs),
      as.integer(nvar),
      as.integer(order),
      cumulants = as.double(rep(0, m))
    )
  return(h$cumulants)
}
```

```
momentsAndCumulants <- function (data, order) {
  nvar <- ncol (data)
  nobs <- nrow (data)
  data <- cbind (1, data)
  ncum <- choose (order + nvar - 1, order)
  nmom <- choose (order + nvar, order)
  nrgf <- tbell[order + 1]
  m <-
    .C(
      "moments",
      as.double(data),
      as.integer(nobs),
      as.integer(nvar + 1),
      as.integer(nmom),
      as.integer(order),
      moments = as.double(rep(0, nmom))
    )
  r <-
    .C("genrgf",
       as.integer(order),
       rgf = as.integer(rep (0, nrgf * order)))
  u <-
    .C("cumulants",
      as.double(m$moments),
      as.integer(r$rgf),
      as.integer(nvar),
      as.integer(order),
      cumulants = as.double (rep(0, ncum))
    )
  return (list(
    moments = m$moments,
    rgf = r$rgf,
    cumulants = u$cumulants
  ))
}
```

4.1.2 old_cumulants.R

```
make_cumulants <- function(y, them = 1:4) {
  n <- nrow (y)
  m <- ncol (y)
  mm <- 1:m
```

```
nn <- 1:n
val <- list()
if (max(them) >= 1) {
  r1 <- colSums (y) / n
}
if (max(them) >= 2) {
  r2 <- crossprod (y) / n
}
if (max(them) >= 3) {
  r3 <- array (0, c (m, m, m))
  for (i in nn) {
    r3 <- r3 + outer (outer (y [i,], y [i,]), y [i,])
  }
  r3 <- r3 / n
}
if (max(them) >= 4) {
  r4 <- array (0, c (m, m, m, m))
  for (i in nn) {
    r4 <-
      r4 + outer (outer (y [i,], y [i,]), outer (y [i,], y [i,]))
  }
  r4 <- r4 / n
}
if (1 %in% them) {
  val <- c(val, list(r1))
}
if (2 %in% them) {
  c2 <- r2 - outer (r1, r1)
  val <- c(val, list(c2))
}
if (3 %in% them) {
  c3 <- r3
  for (i in mm)
    for (j in mm)
      for (k in mm) {
        s3 <- r3 [i, j, k]
        s21 <-
          r2 [i, j] * r1 [k] + r2 [i, k] * r1 [j] + r2 [j, k] * r1 [i]
        s111 <-
          r1 [i] * r1 [j] * r1 [k]
        c3 [i, j, k] <-
          s3 - s21 + 2 * s111
      }
  val <- c(val, list(c3))
```

```
  }
  if (4 %in% them) {
    c4 <- r4
    for (i in mm)
      for (j in mm)
        for (k in mm)
          for (l in mm) {
            s4 <- r4 [i, j, k, l]
            s31 <-
              r3 [i, j, k] * r1 [l] + r3 [i, j, l] * r1 [k] + r3 [i, k, l] * r1 [j] + r3
            s22 <-
              r2 [i, j] * r2 [k, l] + r2 [i, k] * r2 [j, l] + r2 [j, k] * r2 [i, l]
            s211 <-
              r2 [i, j] * r1 [k] * r1 [l] + r2 [i, k] * r1 [j] * r1 [l] + r2 [i, l] * r1
            s1111 <-
              r1 [i] * r1 [j] * r1 [k] * r1 [l]
            c4 [i, j, k, l] <-
              s4 - s31 - s22 + 2 * s211 - 6 * s1111
          }
    val <- c(val, list(c4))
  }
  return (val)
}
```

4.2 C code

4.2.1 cumulants.h

```
#ifndef CUMULANT_H
#define CUMULANT_H

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

#define DEBUG true

int tfact[13] = {1,    1,     2,      6,       24,       120,      720,
                 5040, 40320, 362880, 3628800, 39916800, 479001600};
int tbell[13] = {1,   1,    2,     5,      15,     52,     203,
                 877, 4140, 21147, 115975, 678570, 4213597};
```

```
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 IMAX(const int a, const int b) { return (a > b) ? a : b; }

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

inline int EVEN(const int a) { return ((a % 2) == 0) ? 1 : -1; }

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

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

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

extern int binCoef(const int, const int);

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

extern int fSupSymIncreasing(const int, int *);

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

#endif /* CUMULANT_H */
```

4.2.2 cumulants.c

```
#include "cumulants.h"

// Binomial coefficient

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];
  (void)free(work);
  return (choose);
}

// qsort aux function

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

// Given the coordinates of the cell of a supersymmetric array with order r,
// returns the index of the array element in linear storage

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

// Given the index of the element of a n ^ r supersymmetric array in linear
// storage, returns the coordinates of the cell of the array

void fSupSymIncreasingFirstInverse(const int *dimension, const int *order,
                                   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;
}

// Computes and returns all raw moments and product moments of order r
// of n variables. If you want all moments up to order r, include a
// variable identically equal to one. Returns the r dimensional array
// in compact storage.

void moments(const double *data, const int *pnobs, const int *pnvars,
             const int *pnmax, const int *porder, double *moments) {
  int nvars = *pnvars, nobs = *pnobs, order = *porder, nmax = *pnmax;
  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);
  }
  (void)free(dims);
  (void)free(cell);
  return;
}

// Generates all partitions of a set with n elements using
// the restricted growth function representation

void genrgf(const int *pn, int *rgf) {
  int n = *pn, m = tbell[n];
  int *f = (int *)calloc((size_t)n, sizeof(int));
  int *g = (int *)calloc((size_t)n, sizeof(int));
  for (int i = 1; i <= n; i++) {
    *(f + VINDEX(i)) = 1;
```

```
    *(g + VINDEX(i)) = 2;
  }
  bool done = false;
  int count = 1;
  while (!done) {
    for (int i = 1; i <= n; i++) {
      *(rgf + MINDEX(count, i, m)) = *(f + VINDEX(i));
    }
    count++;
    int j = n + 1;
    while (true) {
      j--;
      if (*(f + VINDEX(j)) != *(g + VINDEX(j))) {
        break;
      }
    }
    if (j > 1) {
      *(f + VINDEX(j)) = *(f + VINDEX(j)) + 1;
      for (int i = j + 1; i <= n; i++) {
        *(f + VINDEX(i)) = 1;
        if (*(f + VINDEX(j)) == *(g + VINDEX(j))) {
          *(g + VINDEX(i)) = *(g + VINDEX(j)) + 1;
        } else {
          *(g + VINDEX(i)) = *(g + VINDEX(j));
        }
      }
    } else {
      done = true;
    }
  }
  (void)free(f);
  (void)free(g);
  return;
}

// Returns all sample cumulants of order r, for n observations on m variables.
// The supersymmmetric array of cumulants is returned in compact storage.

void cumulants(const double *moments, const int *rgf, const int *pnvars,
               const int *porder, double *cumulants) {
  int order = *porder, nvars = *pnvars, nrgf = tbell[order];
  int nmax = binCoef(order + nvars - 1, order);
  int *npart = (int *)calloc((size_t)nrgf, sizeof(int));
  int *nfact = (int *)calloc((size_t)nrgf, sizeof(int));
```

```
int *cell = (int *)calloc((size_t)order, sizeof(int));
int *mind = (int *)calloc((size_t)order, sizeof(int));
int *jrgf = (int *)calloc((size_t)order, sizeof(int));
for (int i = 1; i <= nrgf; i++) {
  int rmax = 0;
  for (int j = 1; j <= order; j++) {
    rmax = IMAX(rmax, *(rgf + MINDEX(i, j, nrgf)));
  }
  *(npart + VINDEX(i)) = rmax;
  *(nfact + VINDEX(i)) = EVEN(rmax - 1) * tfact[rmax - 1];
}
for (int i = 1; i <= nmax; i++) {
  (void)fSupSymIncreasingFirstInverse(pnvars, porder, &i, cell);
  double cumsum = 0.0;
  for (int j = 1; j <= nrgf; j++) {
    int kmax = *(npart + VINDEX(j)), kfac = *(nfact + VINDEX(j));
    double cumprod = 1.0;
    for (int l = 1; l <= order; l++) {
      *(jrgf + VINDEX(l)) = *(rgf + MINDEX(j, l, nrgf));
    }
    for (int k = 1; k <= kmax; k++) {
      for (int l = 1; l <= order; l++) {
        *(mind + VINDEX(l)) = 1;
      }
      int nset = 1;
      for (int l = 1; l <= order; l++) {
        if (*(jrgf + VINDEX(l)) == k) {
          *(mind + VINDEX(nset)) = *(cell + VINDEX(l)) + 1;
          nset++;
        }
      }
      int kind = fSupSymIncreasing(order, mind);
      cumprod *= *(moments + VINDEX(kind));
    }
    cumsum += cumprod * ((double)kfac);
    (void)free(jrgf);
  }
  *(cumulants + VINDEX(i)) = cumsum;
}
(void)free(jrgf);
(void)free(mind);
(void)free(cell);
(void)free(npart);
(void)free(nfact);
```

```
  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 Moments." 2020. http://deleeuwpdx.net/pubfolders/moments/moments.pdf.

Hankin, R. K. S. 2006. "Additive Integer Partitions in R." *Journal of Statistical Software* 16 (Code Snippet 1): 1–3.

Hankin, R. K. S., and L. J. West. 2007. "Set Partitions in R." *Journal of Statistical Software* 23 (Code Snippet 2): 1–12.

Kreher, D. L., and D. R. Stinson. 1998. *Combinatorial Algorithms. Generation, Enumeration, and Search.* CRC Press.