NAG Library Routine Document

G05XEF

Note: before using this routine, please read the Users' Note for your implementation to check the interpretation of ***bold italicised*** terms and other implementation-dependent details.

1 Purpose

G05XEF takes a set of input times and permutes them to specify one of several predefined Brownian bridge construction orders. The permuted times can be passed to G05XAF or G05XCF to initialize the Brownian bridge generators with the chosen bridge construction order.

2 Specification

```
SUBROUTINE G05XEF (BGORD, T0, TEND, NTIMES, INTIME, NMOVE, MOVE, TIMES,    &
                   IFAIL)

INTEGER            BGORD, NTIMES, NMOVE, MOVE(NMOVE), IFAIL
REAL (KIND=nag_wp) T0, TEND, INTIME(NTIMES), TIMES(NTIMES)
```

3 Description

The Brownian bridge algorithm (see Glasserman (2004)) is a popular method for constructing a Wiener process at a set of discrete times, $t_0 < t_1 < t_2 <, \ldots, < t_N < T$, for $N \geq 1$. To ease notation we assume that T has the index $N+1$ so that $T = t_{N+1}$. Inherent in the algorithm is the notion of a *bridge construction order* which specifies the order in which the $N+2$ points of the Wiener process, X_{t_0}, X_T and X_{t_i}, for $i = 1, 2, \ldots, N$, are generated. The value of X_{t_0} is always assumed known, and the first point to be generated is always the final time X_T. Thereafter, successive points are generated iteratively by an interpolation formula, using points which were computed at previous iterations. In many cases the bridge construction order is not important, since any construction order will yield a correct process. However, in certain cases, for example when using quasi-random variates to construct the sample paths, the bridge construction order can be important.

3.1 Supported Bridge Construction Orders

G05XEF accepts as input an array of time points $t_1, t_2, \ldots, t_N, T$ at which the Wiener process is to be sampled. These time points are then permuted to construct the bridge. In all of the supported construction orders the first construction point is T which has index $N+1$. The remaining points are constructed by iteratively bisecting (sub-intervals of) the *time indices* interval $[0, N+1]$, as Figure 1 illustrates:

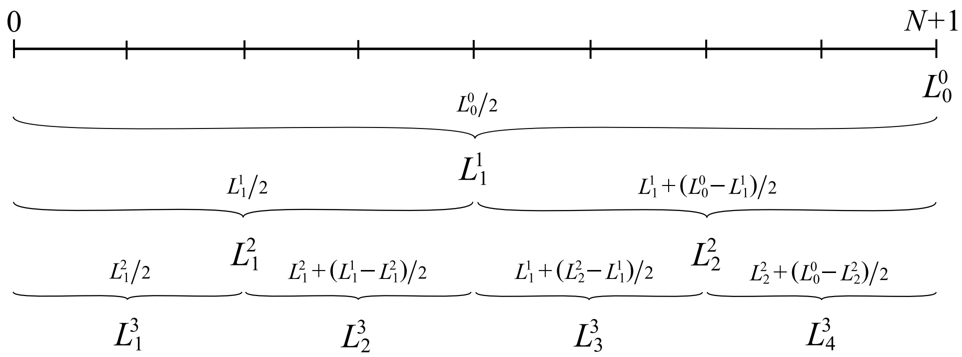

Figure 1

The time indices interval is processed in levels L^i, for $i = 1, 2, \ldots$. Each level L^i contains n_i points $L_1^i, \ldots, L_{n_i}^i$ where $n_i \leq 2^{i-1}$. The number of points at each level depends on the value of N. The points

L_j^i for $i \geq 1$ and $j = 1, 2, \ldots n_i$ are computed as follows: define $L_0^0 = N + 1$ and set

$$L_j^i = J + (K - J)/2 \qquad \text{where}$$
$$J = \max\left\{L_k^p : 1 \leq k \leq n_p,\ 0 \leq p < i \text{ and } L_k^p < L_j^i\right\} \qquad \text{and}$$
$$K = \min\left\{L_k^p : 1 \leq k \leq n_p,\ 0 \leq p < i \text{ and } L_k^p > L_j^i\right\}$$

By convention the maximum of the empty set is taken to be to be zero. Figure 1 illustrates the algorithm when $N + 1$ is a power of two. When $N + 1$ is not a power of two, one must decide how to round the divisions by 2. For example, if one rounds down to the nearest integer, then one could get the following:

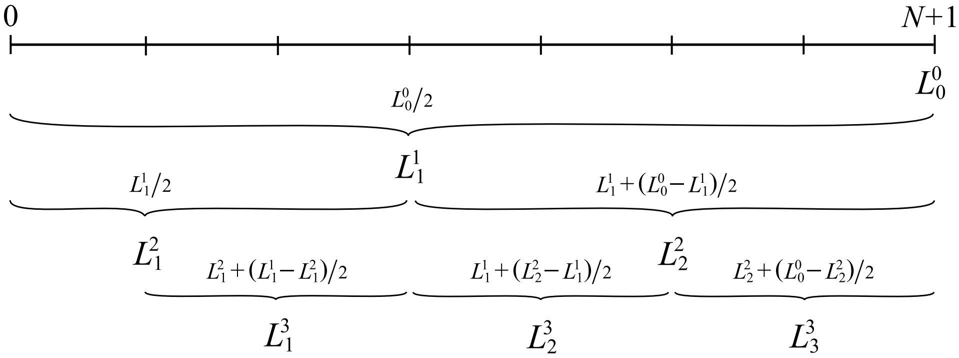

Figure 2

From the series of bisections outlined above, two ways of ordering the time indices L_j^i are supported. In both cases, levels are always processed from coarsest to finest (i.e., increasing i). Within a level, the time indices can either be processed left to right (i.e., increasing j) or right to left (i.e., decreasing j). For example, when processing left to right, the sequence of time indices could be generated as:

$$N + 1 \quad L_1^1 \quad L_1^2 \quad L_2^2 \quad L_1^3 \quad L_2^3 \quad L_3^3 \quad L_4^3 \quad \cdots$$

while when processing right to left, the same sequence would be generated as:

$$N + 1 \quad L_1^1 \quad L_2^2 \quad L_1^2 \quad L_4^3 \quad L_3^3 \quad L_2^3 \quad L_1^3 \quad \cdots$$

G05XEF therefore offers four bridge construction methods; processing either left to right or right to left, with rounding either up or down. Which method is used is controlled by the BGORD parameter. For example, on the set of times

$$t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6 \quad t_7 \quad t_8 \quad t_9 \quad t_{10} \quad t_{11} \quad t_{12} \quad T$$

the Brownian bridge would be constructed in the following orders:

BGORD $= 1$ (processing left to right, rounding down)

$T \quad t_6 \quad t_3 \quad t_9 \quad t_1 \quad t_4 \quad t_7 \quad t_{11} \quad t_2 \quad t_5 \quad t_8 \quad t_{10} \quad t_{12}$

BGORD $= 2$ (processing left to right, rounding up)

$T \quad t_7 \quad t_4 \quad t_{10} \quad t_2 \quad t_6 \quad t_9 \quad t_{12} \quad t_1 \quad t_3 \quad t_5 \quad t_8 \quad t_{11}$

BGORD $= 3$ (processing right to left, rounding down)

$T \quad t_6 \quad t_9 \quad t_3 \quad t_{11} \quad t_7 \quad t_4 \quad t_1 \quad t_{12} \quad t_{10} \quad t_8 \quad t_5 \quad t_2$

BGORD $= 4$ (processing right to left, rounding up)

$T \quad t_7 \quad t_{10} \quad t_4 \quad t_{12} \quad t_9 \quad t_6 \quad t_2 \quad t_{11} \quad t_8 \quad t_5 \quad t_3 \quad t_1\,.$

The four construction methods described above can be further modified through the use of the input array MOVE. To see the effect of this parameter, suppose that an array A holds the output of G05XEF when NMOVE $= 0$ (i.e., the bridge construction order as specified by BGORD only). Let

$$B = \left\{t_j : j = \text{MOVE}(i), i = 1, 2, \ldots, \text{NMOVE}\right\}$$

be the array of all times identified by MOVE, and let C be the array A with all the elements in B removed, i.e.,

$$C = \{A(i) : A(i) \neq B(j), i = 1, 2, \ldots, \text{NTIMES}, j = 1, 2, \ldots, \text{NMOVE}\}.$$

Then the output of G05XEF when NMOVE > 0 is given by

$$B(1) \quad B(2) \quad \cdots \quad B(\text{NMOVE}) \quad C(1) \quad C(2) \quad \cdots \quad C(\text{NTIMES} - \text{NMOVE})$$

When the Brownian bridge is used with quasi-random variates, this functionality can be used to allow specific sections of the bridge to be constructed using the lowest dimensions of the quasi-random points.

4 References

Glasserman P (2004) *Monte Carlo Methods in Financial Engineering* Springer

5 Parameters

1: BGORD – INTEGER *Input*

On entry: the bridge construction order to use.

Constraint: BGORD $= 1$, 2, 3 or 4.

2: T0 – REAL (KIND=nag_wp) *Input*

On entry: t_0, the start value of the time interval on which the Wiener process is to be constructed.

3: TEND – REAL (KIND=nag_wp) *Input*

On entry: T, the largest time at which the Wiener process is to be constructed.

4: NTIMES – INTEGER *Input*

On entry: N, the number of time points in the Wiener process, excluding t_0 and T.

Constraint: NTIMES ≥ 1.

5: INTIME(NTIMES) – REAL (KIND=nag_wp) array *Input*

On entry: the time points, $t_1, t_2, \ldots, t_N$, at which the Wiener process is to be constructed. Note that the final time T is not included in this array.

Constraints:

T0 $<$ INTIME(i) and INTIME(i) $<$ INTIME($i+1$), for $i = 1, 2, \ldots,$ NTIMES $- 1$;
INTIME(NTIMES) $<$ TEND.

6: NMOVE – INTEGER *Input*

On entry: the number of elements in the array MOVE.

Constraint: $0 \leq$ NMOVE $\leq$ NTIMES.

7: MOVE(NMOVE) – INTEGER array *Input*

On entry: the indices of the entries in INTIME which should be moved to the front of the TIMES array, with MOVE(j) $= i$ setting the jth element of TIMES to t_i. Note that i ranges from 1 to NTIMES. When NMOVE $= 0$, MOVE is not referenced.

Constraint: $1 \leq$ MOVE(j) $\leq$ NTIMES, for $j = 1, 2, \ldots,$ NMOVE.

The elements of MOVE must be unique.

8: TIMES(NTIMES) – REAL (KIND=nag_wp) array *Output*

On exit: the output bridge construction order. This should be passed to G05XAF or G05XCF.

9: IFAIL – INTEGER *Input/Output*

On entry: IFAIL must be set to 0, -1 or 1. If you are unfamiliar with this parameter you should refer to Section 3.3 in the Essential Introduction for details.

For environments where it might be inappropriate to halt program execution when an error is detected, the value -1 or 1 is recommended. If the output of error messages is undesirable, then the value 1 is recommended. Otherwise, if you are not familiar with this parameter, the recommended value is 0. **When the value -1 or 1 is used it is essential to test the value of IFAIL on exit.**

On exit: IFAIL $= 0$ unless the routine detects an error or a warning has been flagged (see Section 6).

6 Error Indicators and Warnings

If on entry IFAIL $= 0$ or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

Errors or warnings detected by the routine:

IFAIL $= 1$

On entry, BGORD $= \langle$*value*$\rangle$.
Constraint: BGORD $= 1$, 2, 3 or 4

IFAIL $= 2$

On entry, NTIMES $= \langle$*value*$\rangle$.
Constraint: NTIMES ≥ 1.

IFAIL $= 3$

On entry, NMOVE $= \langle$*value*$\rangle$ and NTIMES $= \langle$*value*$\rangle$.
Constraint: $0 \leq$ NMOVE $\leq$ NTIMES.

IFAIL $= 4$

On entry, INTIME($\langle$*value*$\rangle$) $= \langle$*value*$\rangle$ and INTIME($\langle$*value*$\rangle$) $= \langle$*value*$\rangle$.
Constraint: the elements in INTIME must be in increasing order.

On entry, INTIME(1) $= \langle$*value*$\rangle$ and T0 $= \langle$*value*$\rangle$.
Constraint: INTIME(1) $>$ T0.

On entry, NTIMES $= \langle$*value*$\rangle$, INTIME(NTIMES) $= \langle$*value*$\rangle$ and TEND $= \langle$*value*$\rangle$.
Constraint: INTIME(NTIMES) $<$ TEND.

IFAIL $= 5$

On entry, MOVE($\langle$*value*$\rangle$) $= \langle$*value*$\rangle$.
Constraint: MOVE(i) ≥ 1 for all i.

On entry, MOVE($\langle$*value*$\rangle$) $= \langle$*value*$\rangle$ and NTIMES $= \langle$*value*$\rangle$.
Constraint: MOVE(i) $\leq$ NTIMES for all i.

IFAIL $= 6$

On entry, MOVE($\langle$*value*$\rangle$) and MOVE($\langle$*value*$\rangle$) both equal $\langle$*value*$\rangle$.
Constraint: all elements in MOVE must be unique.

IFAIL $= -99$

An unexpected error has been triggered by this routine. Please contact NAG.

See Section 3.8 in the Essential Introduction for further information.

IFAIL $= -399$

Your licence key may have expired or may not have been installed correctly.

See Section 3.7 in the Essential Introduction for further information.

IFAIL $= -999$

Dynamic memory allocation failed.

See Section 3.6 in the Essential Introduction for further information.

7 Accuracy

Not applicable.

8 Parallelism and Performance

G05XEF is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

Please consult the X06 Chapter Introduction for information on how to control and interrogate the OpenMP environment used within this routine. Please also consult the Users' Note for your implementation for any additional implementation-specific information.

9 Further Comments

None.

10 Example

This example calls G05XEF, G05XAF and G05XBF to generate two sample paths of a three-dimensional free Wiener process. The array MOVE is used to ensure that a certain part of the sample path is always constructed using the lowest dimensions of the input quasi-random points. For further details on using quasi-random points with the Brownian bridge algorithm, please see Section 2.6 in the G05 Chapter Introduction.

10.1 Program Text

```
    Program g05xefe

!     G05XEF Example Program Text

!     Mark 25 Release. NAG Copyright 2014.

!     .. Use Statements ..
      Use nag_library, Only: g05xaf, g05xbf, g05xef, nag_wp
!     .. Implicit None Statement ..
      Implicit None
!     .. Parameters ..
      Integer, Parameter                   :: nout = 6
!     .. Local Scalars ..
      Real (Kind=nag_wp)                   :: t0, tend
      Integer                              :: a, bgord, d, ifail, ldb, ldc,    &
                                              ldz, nmove, npaths, ntimes, rcord
!     .. Local Arrays ..
      Real (Kind=nag_wp), Allocatable      :: b(:,:), c(:,:), intime(:),       &
                                              rcomm(:), start(:), term(:),     &
                                              times(:), z(:,:)
      Integer, Allocatable                 :: move(:)
!     .. Intrinsic Procedures ..
      Intrinsic                            :: size
!     .. Executable Statements ..
!     Get information required to set up the bridge
      Call get_bridge_init_data(bgord,t0,tend,ntimes,intime,nmove,move)
```

```
!     Make the bridge construction bgord
      Allocate (times(ntimes))
      ifail = 0
      Call g05xef(bgord,t0,tend,ntimes,intime,nmove,move,times,ifail)

!     Initialize the Brownian bridge generator
      Allocate (rcomm(12*(ntimes+1)))
      ifail = 0
      Call g05xaf(t0,tend,times,ntimes,rcomm,ifail)

!     Get additional information required by the bridge generator
      Call get_bridge_gen_data(npaths,rcord,d,start,a,term,c)

!     Generate the Z values
      Call get_z(rcord,npaths,d,a,ntimes,z,b)

!     Leading dimensions for the various input arrays
      ldz = size(z,1)
      ldc = size(c,1)
      ldb = size(b,1)

!     Call the Brownian bridge generator routine
      ifail = 0
      Call g05xbf(npaths,rcord,d,start,a,term,z,ldz,c,ldc,b,ldb,rcomm,ifail)

!     Display the results
      Call display_results(rcord,ntimes,d,b)

    Contains
      Subroutine get_bridge_init_data(bgord,t0,tend,ntimes,intime,nmove,move)

!       .. Scalar Arguments ..
        Real (Kind=nag_wp), Intent (Out)     :: t0, tend
        Integer, Intent (Out)                :: bgord, nmove, ntimes
!       .. Array Arguments ..
        Real (Kind=nag_wp), Allocatable, Intent (Out) :: intime(:)
        Integer, Allocatable, Intent (Out)   :: move(:)
!       .. Local Scalars ..
        Integer                              :: i
!       .. Intrinsic Procedures ..
        Intrinsic                            :: real
!       .. Executable Statements ..
!       Set the basic parameters for a Wiener process
        ntimes = 10
        t0 = 0.0_nag_wp
        Allocate (intime(ntimes))

!       We want to generate the Wiener process at these time points
        Do i = 1, ntimes
          intime(i) = t0 + 1.71_nag_wp*real(i,kind=nag_wp)
        End Do
        tend = t0 + 1.71_nag_wp*real(ntimes+1,kind=nag_wp)

!       We suppose the following 3 times are very important and should be
!       constructed first. Note: these are indices into INTIME
        nmove = 3
        Allocate (move(nmove))
        move(1:nmove) = (/3,5,4/)
        bgord = 3
      End Subroutine get_bridge_init_data

      Subroutine get_bridge_gen_data(npaths,rcord,d,start,a,term,c)

!       .. Use Statements ..
        Use nag_library, Only: dpotrf
!       .. Scalar Arguments ..
        Integer, Intent (Out)                :: a, d, npaths, rcord
!       .. Array Arguments ..
        Real (Kind=nag_wp), Allocatable, Intent (Out) :: c(:,:), start(:),     &
                                                         term(:)
```

```
!       .. Local Scalars ..
        Integer                             :: info
!       .. Executable Statements ..
!       Set the basic parameters for a free Wiener process
        npaths = 2
        rcord = 2
        d = 3
        a = 0

        Allocate (start(d),term(d),c(d,d))

        start(1:d) = 0.0_nag_wp
!       As A = 0, TERM need not be initialized

!       We want the following covariance matrix
        c(:,1) = (/6.0_nag_wp,1.0_nag_wp,-0.2_nag_wp/)
        c(:,2) = (/1.0_nag_wp,5.0_nag_wp,0.3_nag_wp/)
        c(:,3) = (/-0.2_nag_wp,0.3_nag_wp,4.0_nag_wp/)

!       G05XBF works with the Cholesky factorization of the covariance matrix C
!       so perform the decomposition
        Call dpotrf('Lower',d,c,d,info)
        If (info/=0) Then
          Write (nout,*) &
            'Specified covariance matrix is not positive definite: info=', &
            info
          Stop
        End If
      End Subroutine get_bridge_gen_data

      Subroutine get_z(rcord,npaths,d,a,ntimes,z,b)

!       .. Use Statements ..
        Use nag_library, Only: g05yjf
!       .. Scalar Arguments ..
        Integer, Intent (In)                :: a, d, npaths, ntimes, rcord
!       .. Array Arguments ..
        Real (Kind=nag_wp), Allocatable, Intent (Out) :: b(:,:), z(:,:)
!       .. Local Scalars ..
        Integer                             :: idim, ifail
!       .. Local Arrays ..
        Real (Kind=nag_wp), Allocatable     :: std(:), tz(:,:), xmean(:)
        Integer, Allocatable                :: iref(:), state(:)
        Integer                             :: seed(1)
!       .. Intrinsic Procedures ..
        Intrinsic                           :: transpose
!       .. Executable Statements ..
        idim = d*(ntimes+1-a)

!       Allocate Z
        If (rcord==1) Then
          Allocate (z(idim,npaths))
          Allocate (b(d*(ntimes+1),npaths))
        Else
          Allocate (z(npaths,idim))
          Allocate (b(npaths,d*(ntimes+1)))
        End If

!       We now need to generate the input quasi-random points
!       First initialize the base pseudorandom number generator
        seed(1) = 1023401
        Call initialize_prng(6,0,seed,size(seed),state)

!       Scrambled quasi-random sequences preserve the good discrepancy
!       properties of quasi-random sequences while counteracting the bias
!       some applications experience when using quasi-random sequences.
!       Initialize the scrambled quasi-random generator.
        Call initialize_scrambled_qrng(1,2,idim,state,iref)

!       Generate the quasi-random points from N(0,1)
        Allocate (xmean(idim),std(idim))
```

```
          xmean(1:idim) = 0.0_nag_wp
          std(1:idim) = 1.0_nag_wp
          If (rcord==1) Then
            Allocate (tz(npaths,idim))
            ifail = 0
            Call g05yjf(xmean,std,npaths,tz,iref,ifail)
            z(:,:) = transpose(tz)
          Else
            ifail = 0
            Call g05yjf(xmean,std,npaths,z,iref,ifail)
          End If
        End Subroutine get_z

        Subroutine initialize_prng(genid,subid,seed,lseed,state)

!         .. Use Statements ..
          Use nag_library, Only: g05kff
!         .. Scalar Arguments ..
          Integer, Intent (In)                   :: genid, lseed, subid
!         .. Array Arguments ..
          Integer, Intent (In)                   :: seed(lseed)
          Integer, Allocatable, Intent (Out)     :: state(:)
!         .. Local Scalars ..
          Integer                                :: ifail, lstate
!         .. Executable Statements ..

!         Initial call to initializer to get size of STATE array
          lstate = 0
          Allocate (state(lstate))
          ifail = 0
          Call g05kff(genid,subid,seed,lseed,state,lstate,ifail)

!         Reallocate STATE
          Deallocate (state)
          Allocate (state(lstate))

!         Initialize the generator to a repeatable sequence
          ifail = 0
          Call g05kff(genid,subid,seed,lseed,state,lstate,ifail)
        End Subroutine initialize_prng

        Subroutine initialize_scrambled_qrng(genid,stype,idim,state,iref)

!         .. Use Statements ..
          Use nag_library, Only: g05ynf
!         .. Scalar Arguments ..
          Integer, Intent (In)                   :: genid, idim, stype
!         .. Array Arguments ..
          Integer, Allocatable, Intent (Out)     :: iref(:)
          Integer, Intent (Inout)                :: state(*)
!         .. Local Scalars ..
          Integer                                :: ifail, iskip, liref, nsdigits
!         .. Executable Statements ..
          liref = 32*idim + 7
          iskip = 0
          nsdigits = 32
          Allocate (iref(liref))
          ifail = 0
          Call g05ynf(genid,stype,idim,iref,liref,iskip,nsdigits,state,ifail)
        End Subroutine initialize_scrambled_qrng

        Subroutine display_results(rcord,ntimes,d,b)

!         .. Scalar Arguments ..
          Integer, Intent (In)                   :: d, ntimes, rcord
!         .. Array Arguments ..
          Real (Kind=nag_wp), Intent (In)        :: b(:,:)
!         .. Local Scalars ..
          Integer                                :: i, j, k
!         .. Executable Statements ..
          Write (nout,*) 'G05XEF Example Program Results'
```

```
        Write (nout,*)

        Do i = 1, npaths
          Write (nout,99999) 'Weiner Path ', i, ', ', ntimes + 1, &
            ' time steps, ', d, ' dimensions'
          Write (nout,99997)(j,j=1,d)
          k = 1
          Do j = 1, ntimes + 1
            If (rcord==1) Then
              Write (nout,99998) j, b(k:k+d-1,i)
            Else
              Write (nout,99998) j, b(i,k:k+d-1)
            End If
            k = k + d
          End Do
          Write (nout,*)
        End Do
99999   Format (1X,A,I0,A,I0,A,I0,A)
99998   Format (1X,I2,1X,20(1X,F10.4))
99997   Format (1X,3X,20(9X,I2))
      End Subroutine display_results
    End Program g05xefe
```

10.2 Program Data

None.

10.3 Program Results

```
 G05XEF Example Program Results

 Weiner Path 1, 11 time steps, 3 dimensions
             1          2          3
  1     -2.1275    -2.4995    -6.0191
  2     -6.1589    -1.3257    -3.7378
  3     -5.1917    -3.1653    -6.2291
  4    -11.5557    -5.9183    -5.9062
  5     -9.2492    -5.7497    -4.2989
  6     -6.7853   -13.9759    -0.8990
  7    -12.7642   -15.6386    -3.6481
  8    -12.5245   -11.8142     3.3504
  9    -15.1995   -15.5145     0.5355
 10    -16.0360   -14.4140     0.0104
 11    -22.6719   -14.3308    -0.2418

 Weiner Path 2, 11 time steps, 3 dimensions
             1          2          3
  1     -0.0973     3.7229     0.8640
  2      0.8027     8.5041    -0.9103
  3     -3.8494     6.1062     0.1231
  4     -6.6643     4.9936    -0.1329
  5     -6.8095     9.3508     4.7022
  6     -7.7178    10.9577    -1.4262
  7     -8.0711    12.7207     4.4744
  8    -12.8353     8.8296     7.6458
  9     -7.9795    12.2399     7.3783
 10     -6.4313    10.0770     5.5234
 11     -6.6258    10.3026     6.5021
```