NAG Library Routine Document

G05XCF

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

G05XCF initializes the Brownian bridge increments generator G05XDF. It must be called before any calls to G05XDF.

2 Specification

```
SUBROUTINE G05XCF (T0, TEND, TIMES, NTIMES, RCOMM, IFAIL)
INTEGER            NTIMES, IFAIL
REAL (KIND=nag_wp) T0, TEND, TIMES(NTIMES), RCOMM(12*(NTIMES+1))
```

3 Description

3.1 Brownian Bridge Algorithm

Details on the Brownian bridge algorithm and the Brownian bridge process (sometimes also called a non-free Wiener process) can be found in Section 2.6 in the G05 Chapter Introduction. We briefly recall some notation and definitions.

Fix two times $t_0 < T$ and let $(t_i)_{1\leq i\leq N}$ be any set of time points satisfying $t_0 < t_1 < t_2 < \cdots < t_N < T$. Let $(X_{t_i})_{1\leq i\leq N}$ denote a d-dimensional Wiener sample path at these time points, and let C be any d by d matrix such that CC^{T} is the desired covariance structure for the Wiener process. Each point X_{t_i} of the sample path is constructed according to the Brownian bridge interpolation algorithm (see Glasserman (2004) or Section 2.6 in the G05 Chapter Introduction). We always start at some fixed point $X_{t_0} = x \in \mathbb{R}^d$. If we set $X_T = x + C\sqrt{T - t_0}Z$ where Z is any d-dimensional standard Normal random variable, then X will behave like a normal (free) Wiener process. However if we fix the terminal value $X_T = w \in \mathbb{R}^d$, then X will behave like a non-free Wiener process.

The Brownian bridge increments generator uses the Brownian bridge algorithm to construct sample paths for the (free or non-free) Wiener process X, and then uses this to compute the *scaled Wiener increments*

$$\frac{X_{t_1} - X_{t_0}}{t_1 - t_0}, \frac{X_{t_2} - X_{t_1}}{t_2 - t_1}, \ldots, \frac{X_{t_N} - X_{t_{N-1}}}{t_N - t_{N-1}}, \frac{X_T - X_{t_N}}{T - t_N}.$$

Such increments can be useful in computing numerical solutions to stochastic differential equations driven by (free or non-free) Wiener processes.

3.2 Implementation

Conceptually, the output of the Wiener increments generator is the same as if G05XAF and G05XBF were called first, and the scaled increments then constructed from their output. The implementation adopts a much more efficient approach whereby the scaled increments are computed directly without first constructing the Wiener sample path.

Given the start and end points of the process, the order in which successive interpolation times t_j are chosen is called the *bridge construction order*. The construction order is given by the array TIMES. Further information on construction orders is given in Section 2.6.2 in the G05 Chapter Introduction. For clarity we consider here the common scenario where the Brownian bridge algorithm is used with quasi-random points. If pseudorandom numbers are used instead, these details can be ignored.

Suppose we require the increments of P Wiener sample paths each of dimension d. The main input to the Brownian bridge increments generator is then an array of quasi-random points $Z^1, Z^2, \ldots, Z^P$ where each point $Z^p = \left(Z_1^p, Z_2^p, \ldots, Z_D^p\right)$ has dimension $D = d(N + 1)$ or $D = dN$ depending on whether a

free or non-free Wiener process is required. When G05XDF is called, the pth sample path for $1 \leq p \leq P$ is constructed as follows: if a non-free Wiener process is required set X_T equal to the terminal value w, otherwise construct X_T as

$$X_T = X_{t_0} + C\sqrt{T - t_0}\begin{bmatrix} Z_1^p \\ \vdots \\ Z_d^p \end{bmatrix}$$

where C is the matrix described in Section 3.1. The array TIMES holds the remaining time points $t_1, t_2, \ldots t_N$ in the order in which the bridge is to be constructed. For each $j = 1, \ldots, N$ set $r = \text{TIMES}(j)$, find

$$q = \max\{t_0, \text{TIMES}(i) : 1 \leq i < j, \text{TIMES}(i) < r\}$$

and

$$s = \min\{T, \text{TIMES}(i) : 1 \leq i < j, \text{TIMES}(i) > r\}$$

and construct the point X_r as

$$X_r = \frac{X_q(s-r) + X_s(r-q)}{s-q} + C\sqrt{\frac{(s-r)(r-q)}{(s-q)}}\begin{bmatrix} Z_{jd-ad+1}^p \\ \vdots \\ Z_{jd-ad+d}^p \end{bmatrix}$$

where $a = 0$ or $a = 1$ depending on whether a free or non-free Wiener process is required. The routine G05XEF can be used to initialize the TIMES array for several predefined bridge construction orders. Lastly, the scaled Wiener increments

$$\frac{X_{t_1} - X_{t_0}}{t_1 - t_0}, \frac{X_{t_2} - X_{t_1}}{t_2 - t_1}, \ldots, \frac{X_{t_N} - X_{t_{N-1}}}{t_N - t_{N-1}}, \frac{X_T - X_{t_N}}{T - t_N}$$

are computed.

4 References

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

5 Parameters

1: T0 – REAL (KIND=nag_wp) *Input*

On entry: the starting value t_0 of the time interval.

2: TEND – REAL (KIND=nag_wp) *Input*

On entry: the end value T of the time interval.

Constraint: TEND $>$ T0.

3: TIMES(NTIMES) – REAL (KIND=nag_wp) array *Input*

On entry: the points in the time interval (t_0, T) at which the Wiener process is to be constructed. The order in which points are listed in TIMES determines the bridge construction order. The routine G05XEF can be used to create predefined bridge construction orders from a set of input times.

Constraints:

T0 $<$ TIMES(i) $<$ TEND, for $i = 1, 2, \ldots,$ NTIMES;
TIMES$(i) \neq$ TIMES(j), for $i, j = 1, 2, \ldots$ NTIMES and $i \neq j$.

4: NTIMES – INTEGER *Input*

On entry: the length of TIMES, denoted by N in Section 3.1.

Constraint: NTIMES ≥ 1.

5: RCOMM$(12 \times (\text{NTIMES} + 1))$ – REAL (KIND=nag_wp) array *Communication Array*

On exit: communication array, used to store information between calls to G05XDF. This array **must not** be directly modified.

6: 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, TEND $= \langle value \rangle$ and T0 $= \langle value \rangle$.
Constraint: TEND $>$ T0.

IFAIL $= 2$

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

IFAIL $= 3$

On entry, TIMES$(\langle value \rangle) = \langle value \rangle$, T0 $= \langle value \rangle$ and TEND $= \langle value \rangle$.
Constraint: T0 $<$ TIMES(i) $<$ TEND for all i.

IFAIL $= 4$

On entry, TIMES(i) = TIMES$(j) = \langle value \rangle$, for $i = \langle value \rangle$ and $j = \langle value \rangle$.
Constraint: all elements of TIMES 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

Not applicable.

9 Further Comments

The efficient implementation of a Brownian bridge algorithm requires the use of a workspace array called the *working stack*. Since previously computed points will be used to interpolate new points, they should be kept close to the hardware processing units so that the data can be accessed quickly. Ideally the whole stack should be held in hardware cache. Different bridge construction orders may require different amounts of working stack. Indeed, a naive bridge algorithm may require a stack of size $\frac{N}{4}$ or even $\frac{N}{2}$, which could be very inefficient when N is large. G05XCF performs a detailed analysis of the bridge construction order specified by TIMES. Heuristics are used to find an execution strategy which requires a small working stack, while still constructing the bridge in the order required.

10 Example

The following example program calls G05XAF and G05XBF to generate two sample paths from a two-dimensional free Wiener process. It then calls G05XCF and G05XDF with the same input arguments to obtain the scaled increments of the Wiener sample paths. Lastly, the program prints the Wiener sample paths from G05XBF, the scaled increments from G05XDF, and the cumulative sum of the unscaled increments side by side. Note that the cumulative sum of the unscaled increments is identical to the output of G05XBF.

Please see Section 10 in G05XDF for additional examples.

10.1 Program Text

```
    Program g05xcfe

!     G05XCF Example Program Text

!     Mark 25 Release. NAG Copyright 2014.

!     .. Use Statements ..
      Use nag_library, Only: g05xaf, g05xbf, g05xcf, g05xdf, 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     :: bb(:,:), bd(:,:), c(:,:),        &
                                             diff(:), intime(:), rcommb(:),   &
                                             rcommd(:), start(:), term(:),    &
                                             times(:), zb(:,:), zd(:,:)
      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 (rcommb(12*(ntimes+1)),rcommd(12*(ntimes+1)))
      ifail = 0
      Call g05xaf(t0,tend,times,ntimes,rcommb,ifail)
      ifail = 0
      Call g05xcf(t0,tend,times,ntimes,rcommd,ifail)

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

      Call allocate_arrays(rcord,npaths,d,a,ntimes,zb,bb,zd,bd)

!     Generate the Z values
      Call get_z(rcord,npaths,d*(ntimes+1-a),zb)
      zd(:,:) = zb(:,:)

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

!     Call the Brownian bridge generator routine
      ifail = 0
      Call g05xbf(npaths,rcord,d,start,a,term,zb,ldz,c,ldc,bb,ldb,rcommb, &
        ifail)
      ifail = 0
      Call g05xdf(npaths,rcord,d,a,diff,zd,ldz,c,ldc,bd,ldb,rcommd,ifail)

!     Display the results
      Call display_results(rcord,t0,tend,ntimes,intime,d,start,bb,bd)

    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 + real(i,kind=nag_wp)
        End Do
        tend = t0 + real(ntimes+1,kind=nag_wp)

        nmove = 0
        Allocate (move(nmove))
        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 = 2
        a = 0

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

        start(1:d) = (/0.0_nag_wp,2.0_nag_wp/)
        term(1:d) = (/1.0_nag_wp,0.0_nag_wp/)

!       We want the following covariance matrix
        c(:,1) = (/6.0_nag_wp,-1.0_nag_wp/)
        c(:,2) = (/-1.0_nag_wp,5.0_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 allocate_arrays(rcord,npaths,d,a,ntimes,zb,bb,zd,bd)

!       .. Scalar Arguments ..
        Integer, Intent (In)                 :: a, d, npaths, ntimes, rcord
!       .. Array Arguments ..
        Real (Kind=nag_wp), Allocatable, Intent (Out) :: bb(:,:), bd(:,:),     &
                                                         zb(:,:), zd(:,:)
!       .. Local Scalars ..
        Integer                              :: idim
!       .. Executable Statements ..
        idim = d*(ntimes+1-a)

        If (rcord==1) Then
          Allocate (zb(idim,npaths),zd(idim,npaths))
          Allocate (bb(d*(ntimes+1),npaths),bd(d*(ntimes+1),npaths))
        Else
          Allocate (zb(npaths,idim),zd(npaths,idim))
          Allocate (bb(npaths,d*(ntimes+1)),bd(npaths,d*(ntimes+1)))
        End If

      End Subroutine allocate_arrays

      Subroutine get_z(rcord,npaths,idim,z)

!       .. Use Statements ..
        Use nag_library, Only: g05skf
!       .. Scalar Arguments ..
        Integer, Intent (In)                 :: idim, npaths, rcord
!       .. Array Arguments ..
        Real (Kind=nag_wp), Intent (Out)     :: z(npaths*idim)
!       .. Local Scalars ..
        Integer                              :: ifail
!       .. Local Arrays ..
```

```
        Real (Kind=nag_wp), Allocatable      :: ztmp(:,:), ztmp2(:,:)
        Integer                              :: seed(1)
        Integer, Allocatable                 :: state(:)
!       .. Intrinsic Procedures ..
        Intrinsic                            :: reshape, transpose
!       .. Executable Statements ..
!       We now need to generate the input pseudorandom points
!       First initialize the base pseudorandom number generator
        seed(1) = 1023401
        Call initialize_prng(6,0,seed,size(seed),state)

!       Generate the pseudorandom points from N(0,1)
        ifail = 0
        Call g05skf(idim*npaths,0.0_nag_wp,1.0_nag_wp,state,z,ifail)
        If (rcord==1) Then
          Allocate (ztmp(npaths,idim),ztmp2(idim,npaths))
          ztmp(1:npaths,1:idim) = reshape(z,(/npaths,idim/))
          ztmp2(1:idim,1:npaths) = transpose(ztmp)
          z(1:npaths*idim) = reshape(ztmp2,(/npaths*idim/))
        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 display_results(rcord,t0,tend,ntimes,intime,d,start,bb,bd)

!       .. Scalar Arguments ..
        Real (Kind=nag_wp), Intent (In)      :: t0, tend
        Integer, Intent (In)                 :: d, ntimes, rcord
!       .. Array Arguments ..
        Real (Kind=nag_wp), Intent (In)      :: bb(:,:), bd(:,:), intime(:),   &
                                                start(:)
!       .. Local Scalars ..
        Integer                              :: i, n
!       .. Local Arrays ..
        Real (Kind=nag_wp), Allocatable      :: cum(:), unscaled(:)
!       .. Executable Statements ..
        Allocate (cum(d),unscaled(d))
        Write (nout,*) 'G05XCF Example Program Results'
        Write (nout,*)

        Do n = 1, npaths
          Write (nout,99999) 'Weiner Path ', n, ', ', ntimes + 1, &
            ' time steps, ', d, ' dimensions'
          Write (nout,'(A)') &
            '      Output of G05XBF    Output of G05XDF      Sum of G05XDF'
```

```
          cum(:) = start(:)
          If (rcord==1) Then
            unscaled(:) = bd(1:d,n)*(intime(1)-t0)
            cum(:) = cum(:) + unscaled(:)
            Write (nout,99998) 1, bb(1:d,n), bd(1:d,n), cum(1:d)
          Else
            unscaled(:) = bd(n,1:d)*(intime(1)-t0)
            cum(:) = cum(:) + unscaled(:)
            Write (nout,99998) 1, bb(n,1:d), bd(n,1:d), cum(1:d)
          End If
          Do i = 2, ntimes
            If (rcord==1) Then
              unscaled(:) = bd(1+(i-1)*d:i*d,n)*(intime(i)-intime(i-1))
              cum(:) = cum(:) + unscaled(:)
              Write (nout,99998) i, bb(1+(i-1)*d:i*d,n), bd(1+(i-1)*d:i*d,n), &
                cum(1:d)
            Else
              unscaled(:) = bd(n,1+(i-1)*d:i*d)*(intime(i)-intime(i-1))
              cum(:) = cum(:) + unscaled(:)
              Write (nout,99998) i, bb(n,1+(i-1)*d:i*d), bd(n,1+(i-1)*d:i*d), &
                cum(1:d)
            End If
          End Do
          i = ntimes + 1
          If (rcord==1) Then
            unscaled(:) = bd(1+(i-1)*d:i*d,n)*(tend-intime(i-1))
            cum(:) = cum(:) + unscaled(:)
            Write (nout,99998) i, bb(1+(i-1)*d:i*d,n), bd(1+(i-1)*d:i*d,n), &
              cum(1:d)
          Else
            unscaled(:) = bd(n,1+(i-1)*d:i*d)*(tend-intime(i-1))
            cum(:) = cum(:) + unscaled(:)
            Write (nout,99998) i, bb(n,1+(i-1)*d:i*d), bd(n,1+(i-1)*d:i*d), &
              cum(1:d)
          End If

          Write (nout,*)
        End Do


99999   Format (1X,A,I0,A,I0,A,I0,A)
99998   Format (1X,I2,1X,2(1X,F8.4),2X,2(1X,F8.4),2X,2(1X,F8.4))
      End Subroutine display_results
    End Program g05xcfe
```

10.2 Program Data

None.

10.3 Program Results

```
 G05XCF Example Program Results

 Weiner Path 1, 11 time steps, 2 dimensions
      Output of G05XBF    Output of G05XDF      Sum of G05XDF
  1   -2.2323   1.6656    -2.2323  -0.3344    -2.2323   1.6656
  2   -5.2301   1.2812    -2.9978  -0.3844    -5.2301   1.2812
  3   -0.9025  -1.2421     4.3276  -2.5234    -0.9025  -1.2421
  4   -3.6799  -0.3972    -2.7774   0.8449    -3.6799  -0.3972
  5   -6.5789  -2.0358    -2.8990  -1.6386    -6.5789  -2.0358
  6  -11.2879  -1.1972    -4.7090   0.8385   -11.2879  -1.1972
  7   -8.8959  -1.6751     2.3919  -0.4779    -8.8959  -1.6751
  8   -9.7103  -2.0523    -0.8144  -0.3772    -9.7103  -2.0523
  9   -8.5720  -3.3306     1.1383  -1.2783    -8.5720  -3.3306
 10   -9.8245  -3.2035    -1.2524   0.1271    -9.8245  -3.2035
 11   -4.9941  -8.3506     4.8304  -5.1471    -4.9941  -8.3506

 Weiner Path 2, 11 time steps, 2 dimensions
      Output of G05XBF    Output of G05XDF      Sum of G05XDF
```

1	-1.4101	0.0576	-1.4101	-1.9424	-1.4101	0.0576
2	-3.5738	0.2519	-2.1637	0.1943	-3.5738	0.2519
3	-5.2528	1.7232	-1.6790	1.4713	-5.2528	1.7232
4	-0.8540	1.0897	4.3988	-0.6335	-0.8540	1.0897
5	0.4905	-0.9098	1.3445	-1.9995	0.4905	-0.9098
6	2.3322	1.3415	1.8417	2.2514	2.3322	1.3415
7	3.0105	-4.3312	0.6783	-5.6728	3.0105	-4.3312
8	2.6776	-3.4437	-0.3329	0.8875	2.6776	-3.4437
9	0.6546	-2.7291	-2.0230	0.7146	0.6546	-2.7291
10	-1.3175	-3.8166	-1.9721	-1.0875	-1.3175	-3.8166
11	-3.0214	-3.5439	-1.7039	0.2727	-3.0214	-3.5439