NAG Library Function Document

nag_rand_varma (g05pjc)

1 Purpose

nag_rand_varma (g05pjc) generates a realization of a multivariate time series from a vector autoregressive moving average (VARMA) model. The realization may be continued or a new realization generated at subsequent calls to nag_rand_varma (g05pjc).

2 Specification

```
#include <nag.h>
#include <nagg05.h>

void nag_rand_varma (Nag_OrderType order, Nag_ModeRNG mode, Integer n,
     Integer k, const double xmean[], Integer p, const double phi[],
     Integer q, const double theta[], const double var[], Integer pdv,
     double r[], Integer lr, Integer state[], double x[], Integer pdx,
     NagError *fail)
```

3 Description

Let the vector $X_t = (x_{1t}, x_{2t}, \ldots, x_{kt})^{\mathrm{T}}$, denote a k-dimensional time series which is assumed to follow a vector autoregressive moving average (VARMA) model of the form:

$$\begin{aligned} X_t - \mu = \quad & \phi_1(X_{t-1} - \mu) + \phi_2(X_{t-2} - \mu) + \cdots + \phi_p(X_{t-p} - \mu) + \\ & \epsilon_t - \theta_1\epsilon_{t-1} - \theta_2\epsilon_{t-2} - \cdots - \theta_q\epsilon_{t-q} \end{aligned} \tag{1}$$

where $\epsilon_t = (\epsilon_{1t}, \epsilon_{2t}, \ldots, \epsilon_{kt})^{\mathrm{T}}$, is a vector of k residual series assumed to be Normally distributed with zero mean and covariance matrix Σ. The components of ϵ_t are assumed to be uncorrelated at non-simultaneous lags. The ϕ_i's and θ_j's are k by k matrices of parameters. $\{\phi_i\}$, for $i = 1, 2, \ldots, p$, are called the autoregressive (AR) parameter matrices, and $\{\theta_j\}$, for $j = 1, 2, \ldots, q$, the moving average (MA) parameter matrices. The parameters in the model are thus the p k by k ϕ-matrices, the q k by k θ-matrices, the mean vector μ and the residual error covariance matrix Σ. Let

$$A(\phi) = \begin{bmatrix} \phi_1 & I & 0 & . & . & . & 0 \\ \phi_2 & 0 & I & 0 & . & . & 0 \\ . & & & . & & & \\ . & & & & . & & \\ . & & & & & . & \\ \phi_{p-1} & 0 & . & . & . & 0 & I \\ \phi_p & 0 & . & . & . & 0 & 0 \end{bmatrix}_{pk \times pk} \quad \text{and} \quad B(\theta) = \begin{bmatrix} \theta_1 & I & 0 & . & . & . & 0 \\ \theta_2 & 0 & I & 0 & . & . & 0 \\ . & & & . & & & \\ . & & & & . & & \\ . & & & & & . & \\ \theta_{q-1} & 0 & . & . & . & 0 & I \\ \theta_q & 0 & . & . & . & 0 & 0 \end{bmatrix}_{qk \times qk}$$

where I denotes the k by k identity matrix.

The model (1) must be both stationary and invertible. The model is said to be stationary if the eigenvalues of $A(\phi)$ lie inside the unit circle and invertible if the eigenvalues of $B(\theta)$ lie inside the unit circle.

For $k \geq 6$ the VARMA model (1) is recast into state space form and a realization of the state vector at time zero computed. For all other cases the function computes a realization of the pre-observed vectors $X_0, X_{-1}, \ldots, X_{1-p}$, $\epsilon_0, \epsilon_{-1}, \ldots, \epsilon_{1-q}$, from (1), see Shea (1988). This realization is then used to generate a sequence of successive time series observations. Note that special action is taken for pure MA models, that is for $p = 0$.

At your request a new realization of the time series may be generated more efficiently using the information in a reference vector created during a previous call to nag_rand_varma (g05pjc). See the description of the argument **mode** in Section 5 for details.

The function returns a realization of $X_1, X_2, \ldots, X_n$. On a successful exit, the recent history is updated and saved in the array **r** so that nag_rand_varma (g05pjc) may be called again to generate a realization of $X_{n+1}, X_{n+2}, \ldots$, etc. See the description of the argument **mode** in Section 5 for details.

Further computational details are given in Shea (1988). Note, however, that nag_rand_varma (g05pjc) uses a spectral decomposition rather than a Cholesky factorization to generate the multivariate Normals. Although this method involves more multiplications than the Cholesky factorization method and is thus slightly slower it is more stable when faced with ill-conditioned covariance matrices. A method of assigning the AR and MA coefficient matrices so that the stationarity and invertibility conditions are satisfied is described in Barone (1987).

One of the initialization functions nag_rand_init_repeatable (g05kfc) (for a repeatable sequence if computed sequentially) or nag_rand_init_nonrepeatable (g05kgc) (for a non-repeatable sequence) must be called prior to the first call to nag_rand_varma (g05pjc).

4 References

Barone P (1987) A method for generating independent realisations of a multivariate normal stationary and invertible ARMA(p, q) process *J. Time Ser. Anal.* **8** 125–130

Shea B L (1988) A note on the generation of independent realisations of a vector autoregressive moving average process *J. Time Ser. Anal.* **9** 403–410

5 Arguments

1: **order** – Nag_OrderType *Input*

On entry: the **order** argument specifies the two-dimensional storage scheme being used, i.e., row-major ordering or column-major ordering. C language defined storage is specified by **order** = Nag_RowMajor. See Section 3.2.1.3 in the Essential Introduction for a more detailed explanation of the use of this argument.

Constraint: **order** = Nag_RowMajor or Nag_ColMajor.

2: **mode** – Nag_ModeRNG *Input*

On entry: a code for selecting the operation to be performed by the function.

mode = Nag_InitializeReference
Set up reference vector and compute a realization of the recent history.

mode = Nag_GenerateFromReference
Generate terms in the time series using reference vector set up in a prior call to nag_rand_varma (g05pjc).

mode = Nag_InitializeAndGenerate
Combine the operations of **mode** = Nag_InitializeReference and Nag_GenerateFromReference.

mode = Nag_ReGenerateFromReference
A new realization of the recent history is computed using information stored in the reference vector, and the following sequence of time series values are generated.

If **mode** = Nag_GenerateFromReference or Nag_ReGenerateFromReference, then you must ensure that the reference vector **r** and the values of **k**, **p**, **q**, **xmean**, **phi**, **theta**, **var** and **pdv** have not been changed between calls to nag_rand_varma (g05pjc).

Constraint: **mode** = Nag_InitializeReference, Nag_GenerateFromReference, Nag_InitializeAndGenerate or Nag_ReGenerateFromReference.

3: **n** – Integer *Input*

On entry: n, the number of observations to be generated.

Constraint: **n** ≥ 0.

4: **k** – Integer *Input*

On entry: k, the dimension of the multivariate time series.

Constraint: $\mathbf{k} \geq 1$.

5: **xmean**[**k**] – const double *Input*

On entry: μ, the vector of means of the multivariate time series.

6: **p** – Integer *Input*

On entry: p, the number of autoregressive parameter matrices.

Constraint: $\mathbf{p} \geq 0$.

7: **phi**[$\mathbf{k} \times \mathbf{k} \times \mathbf{p}$] – const double *Input*

On entry: must contain the elements of the $\mathbf{p} \times \mathbf{k} \times \mathbf{k}$ autoregressive parameter matrices of the model, $\phi_1, \phi_2, \ldots, \phi_p$. The (i,j)th element of ϕ_l is stored in **phi**[$(l-1) \times k \times k + (j-1) \times k + i - 1$], for $l = 1, 2, \ldots, p$, $i = 1, 2, \ldots, k$ and $j = 1, 2, \ldots, k$.

Constraint: the elements of **phi** must satisfy the stationarity condition.

8: **q** – Integer *Input*

On entry: q, the number of moving average parameter matrices.

Constraint: $\mathbf{q} \geq 0$.

9: **theta**[$\mathbf{k} \times \mathbf{k} \times \mathbf{q}$] – const double *Input*

On entry: must contain the elements of the $\mathbf{q} \times \mathbf{k} \times \mathbf{k}$ moving average parameter matrices of the model, $\theta_1, \theta_2, \ldots, \theta_q$. The (i,j)th element of θ_l is stored in **theta**[$(l-1) \times k \times k + (j-1) \times k + i - 1$], for $l = 1, 2, \ldots, q$, $i = 1, 2, \ldots, k$ and $j = 1, 2, \ldots, k$.

Constraint: the elements of **theta** must be within the invertibility region.

10: **var**[*dim*] – const double *Input*

Note: the dimension, *dim*, of the array **var** must be at least $\mathbf{pdv} \times \mathbf{k}$.

Where $\mathbf{VAR}(i,j)$ appears in this document, it refers to the array element

var[$(j-1) \times \mathbf{pdv} + i - 1$] when **order** = Nag_ColMajor;
var[$(i-1) \times \mathbf{pdv} + j - 1$] when **order** = Nag_RowMajor.

On entry: $\mathbf{VAR}(i,j)$ must contain the (i,j)th element of Σ, for $i = 1, 2, \ldots, \mathbf{k}$ and $j = 1, 2, \ldots, \mathbf{k}$. Only the lower triangle is required.

Constraint: the elements of **var** must be such that Σ is positive semidefinite.

11: **pdv** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **var**.

Constraint: $\mathbf{pdv} \geq \mathbf{k}$.

12: **r**[**lr**] – double *Communication Array*

On entry: if **mode** = Nag_GenerateFromReference or Nag_ReGenerateFromReference, the array **r** as output from the previous call to nag_rand_varma (g05pjc) must be input without any change.

If **mode** = Nag_InitializeReference or Nag_InitializeAndGenerate, the contents of **r** need not be set.

On exit: information required for any subsequent calls to the function with **mode** = Nag_GenerateFromReference or Nag_ReGenerateFromReference. See Section 9.

13: **lr** – Integer *Input*

On entry: the dimension of the array **r**.

Constraints:

if $\mathbf{k} \geq 6$, $\mathbf{lr} \geq (5r^2 + 1) \times \mathbf{k}^2 + (4r + 3) \times \mathbf{k} + 4$;
if $\mathbf{k} < 6$, $\mathbf{lr} \geq \left((\mathbf{p} + \mathbf{q})^2 + 1\right) \times \mathbf{k}^2 +$
$(4 \times (\mathbf{p} + \mathbf{q}) + 3) \times \mathbf{k} + \max\left(\mathbf{k}r(\mathbf{k}r + 2), \mathbf{k}^2(\mathbf{p} + \mathbf{q})^2 + l(l + 3) + \mathbf{k}^2(\mathbf{q} + 1)\right) + 4$.

Where $r = \max(\mathbf{p}, \mathbf{q})$ and if $\mathbf{p} = 0$, $l = \mathbf{k}(\mathbf{k} + 1)/2$, or if $\mathbf{p} \geq 1$, $l = \mathbf{k}(\mathbf{k} + 1)/2 + (\mathbf{p} - 1)\mathbf{k}^2$.

See Section 9 for some examples of the required size of the array **r**.

14: **state**[*dim*] – Integer *Communication Array*

Note: the dimension, *dim*, of this array is dictated by the requirements of associated functions that must have been previously called. This array MUST be the same array passed as argument **state** in the previous call to nag_rand_init_repeatable (g05kfc) or nag_rand_init_nonrepeatable (g05kgc).

On entry: contains information on the selected base generator and its current state.

On exit: contains updated information on the state of the generator.

15: **x**[*dim*] – double *Output*

Note: the dimension, *dim*, of the array **x** must be at least

$\max(1, \mathbf{pdx} \times \mathbf{n})$ when **order** = Nag_ColMajor;
$\max(1, \mathbf{k} \times \mathbf{pdx})$ when **order** = Nag_RowMajor.

Where $\mathbf{X}(i, t)$ appears in this document, it refers to the array element

$\mathbf{x}[(t - 1) \times \mathbf{pdx} + i - 1]$ when **order** = Nag_ColMajor;
$\mathbf{x}[(i - 1) \times \mathbf{pdx} + t - 1]$ when **order** = Nag_RowMajor.

On exit: $\mathbf{X}(i, t)$ will contain a realization of the ith component of X_t, for $i = 1, 2, \ldots, k$ and $t = 1, 2, \ldots, n$.

16: **pdx** – Integer *Input*

On entry: the stride separating row or column elements (depending on the value of **order**) in the array **x**.

Constraints:

if **order** = Nag_ColMajor, $\mathbf{pdx} \geq \mathbf{k}$;
if **order** = Nag_RowMajor, $\mathbf{pdx} \geq \mathbf{n}$.

17: **fail** – NagError * *Input/Output*

The NAG error argument (see Section 3.6 in the Essential Introduction).

6 Error Indicators and Warnings

NE_ALLOC_FAIL

Dynamic memory allocation failed.
See Section 3.2.1.2 in the Essential Introduction for further information.

NE_BAD_PARAM

On entry, argument $\langle value\rangle$ had an illegal value.

NE_CLOSE_TO_STATIONARITY

The reference vector cannot be computed because the AR parameters are too close to the boundary of the stationarity region.

NE_INT

On entry, $\mathbf{k} = \langle value\rangle$.
Constraint: $\mathbf{k} \geq 1$.

On entry, **lr** is not large enough, $\mathbf{lr} = \langle value\rangle$: minimum length required $= \langle value\rangle$.

On entry, $\mathbf{n} = \langle value\rangle$.
Constraint: $\mathbf{n} \geq 0$.

On entry, $\mathbf{p} = \langle value\rangle$.
Constraint: $\mathbf{p} \geq 0$.

On entry, $\mathbf{pdv} = \langle value\rangle$.
Constraint: $\mathbf{pdv} > 0$.

On entry, $\mathbf{pdx} = \langle value\rangle$.
Constraint: $\mathbf{pdx} > 0$.

On entry, $\mathbf{q} = \langle value\rangle$.
Constraint: $\mathbf{q} \geq 0$.

NE_INT_2

On entry, $\mathbf{pdv} = \langle value\rangle$ and $\mathbf{k} = \langle value\rangle$.
Constraint: $\mathbf{pdv} \geq \mathbf{k}$.

On entry, $\mathbf{pdx} = \langle value\rangle$ and $\mathbf{k} = \langle value\rangle$.
Constraint: $\mathbf{pdx} \geq \mathbf{k}$.

On entry, $\mathbf{pdx} = \langle value\rangle$ and $\mathbf{n} = \langle value\rangle$.
Constraint: $\mathbf{pdx} \geq \mathbf{n}$.

NE_INTERNAL_ERROR

An internal error has occurred in this function. Check the function call and any array sizes. If the call is correct then please contact NAG for assistance.

An unexpected error has been triggered by this function. Please contact NAG.
See Section 3.6.6 in the Essential Introduction for further information.

NE_INVALID_STATE

On entry, **state** vector has been corrupted or not initialized.

NE_INVERTIBILITY

On entry, the moving average parameter matrices are such that the model is non-invertible.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 3.6.5 in the Essential Introduction for further information.

NE_POS_DEF

On entry, the covariance matrix **var** is not positive semidefinite to ***machine precision***.

NE_PREV_CALL

k is not the same as when **r** was set up in a previous call.
Previous value of $\mathbf{k} = \langle value\rangle$ and $\mathbf{k} = \langle value\rangle$.

NE_STATIONARY_AR

On entry, the AR parameters are outside the stationarity region.

NE_TOO_MANY_ITER

An excessive number of iterations were required by the NAG function used to evaluate the eigenvalues of the covariance matrix.

An excessive number of iterations were required by the NAG function used to evaluate the eigenvalues of the matrices used to test for stationarity or invertibility.

An excessive number of iterations were required by the NAG function used to evaluate the eigenvalues stored in the reference vector.

7 Accuracy

The accuracy is limited by the matrix computations performed, and this is dependent on the condition of the argument and covariance matrices.

8 Parallelism and Performance

nag_rand_varma (g05pjc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_rand_varma (g05pjc) makes calls to BLAS and/or LAPACK routines, which may be threaded within the vendor library used by this implementation. Consult the documentation for the vendor library for further information.

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

9 Further Comments

Note that, in reference to **fail.code** = NE_INVERTIBILITY, nag_rand_varma (g05pjc) will permit moving average parameters on the boundary of the invertibility region.

The elements of **r** contain amongst other information details of the spectral decompositions which are used to generate future multivariate Normals. Note that these eigenvectors may not be unique on different machines. For example the eigenvectors corresponding to multiple eigenvalues may be permuted. Although an effort is made to ensure that the eigenvectors have the same sign on all machines, differences in the signs may theoretically still occur.

The following table gives some examples of the required size of the array **r**, specified by the argument **lr**, for $k = 1, 2$ or 3, and for various values of p and q.

		q			
		0	1	2	3
p	0	13 36 85	20 56 124	31 92 199	46 144 310
	1	19 52 115	30 88 190	45 140 301	64 208 448
	2	35 136 397	50 188 508	69 256 655	92 340 838
	3	57 268 877	76 336 1024	99 420 1207	126 520 1426

Note that nag_tsa_arma_roots (g13dxc) may be used to check whether a VARMA model is stationary and invertible.

The time taken depends on the values of p, q and especially n and k.

10 Example

This program generates two realizations, each of length 48, from the bivariate AR(1) model

$$X_t - \mu = \phi_1(X_{t-1} - \mu) + \epsilon_t$$

with

$$\phi_1 = \begin{bmatrix} 0.80 & 0.07 \\ 0.00 & 0.58 \end{bmatrix},$$

$$\mu = \begin{bmatrix} 5.00 \\ 9.00 \end{bmatrix},$$

and

$$\Sigma = \begin{bmatrix} 2.97 & 0 \\ 0.64 & 5.38 \end{bmatrix}.$$

The pseudorandom number generator is initialized by a call to nag_rand_init_repeatable (g05kfc). Then, in the first call to nag_rand_varma (g05pjc), **mode** $=$ Nag_InitializeAndGenerate in order to set up the reference vector before generating the first realization. In the subsequent call **mode** $=$ Nag_ReGenerateFromReference and a new recent history is generated and used to generate the second realization.

10.1 Program Text

```
/* nag_rand_varma (g05pjc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 9, 2009.
 */
/* Pre-processor includes */
#include <stdio.h>
#include <math.h>
```

```
#include <nag.h>
#include <nag_stdlib.h>
#include <nagg05.h>

#define VAR(I, J)      var[(order == Nag_RowMajor)?(I*pdvar+J):(J*pdvar+I)]
#define X(I, J)        x[(order == Nag_RowMajor)?(I*pdx+J):(J*pdx+I)]
#define PHI(i, j, l)   phi[l*k*k+j*k+i]
#define THETA(i, j, l) theta[l*k*k+j*k+i]

int main(void)
{
  /* Integer scalar and array declarations */
  Integer        exit_status = 0;
  Integer        lr, x_size, var_size, i, ip, iq, j, k, l, lstate, n, tmp1,
                 tmp2, tmp3, tmp4, tmp5;
  Integer        *state = 0;
  Integer        pdx, pdvar;
  /* NAG structures */
  NagError       fail;
  Nag_ModeRNG    mode;
  /* Double scalar and array declarations */
  double         *phi = 0, *r = 0, *theta = 0, *var = 0, *x = 0, *xmean = 0;
  /* Use column major order */
  Nag_OrderType order = Nag_ColMajor;
  /* Choose the base generator */
  Nag_BaseRNG    genid = Nag_Basic;
  Integer        subid = 0;
  /* Set the seed */
  Integer        seed[] = { 1762543 };
  Integer        lseed = 1;

  /* Initialise the error structure */
  INIT_FAIL(fail);

  printf("nag_rand_varma (g05pjc) Example Program Results\n\n");

  /* Get the length of the state array */
  lstate = -1;
  nag_rand_init_repeatable(genid, subid, seed, lseed, state, &lstate, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_rand_init_repeatable (g05kfc).\n%s\n",
              fail.message);
      exit_status = 1;
      goto END;
    }

  /* Read data from a file */
  /* Skip heading*/
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif
  /* Read in initial parameters */
#ifdef _WIN32
  scanf_s("%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT"%*[^\n] ", &k, &ip,
         &iq, &n);
#else
  scanf("%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT"%"NAG_IFMT"%*[^\n] ", &k, &ip,
         &iq, &n);
#endif

  /* Calculate the size of the reference vector */
  tmp1 = (ip > iq)?ip:iq;
  if (ip == 0)
    {
      tmp2 = k * (k + 1) / 2;
    }
  else
    {
```

```
      tmp2 = k*(k+1)/2 + (ip-1)*k*k;
    }
  tmp3 = ip + iq;
  if (k >= 6)
    {
      lr = (5*tmp1*tmp1+1)*k*k + (4*tmp1+3)*k + 4;
    }
  else
    {
      tmp4 = k*tmp1*(k*tmp1+2);
      tmp5 = k*k*tmp3*tmp3+tmp2*(tmp2+3)+k*k*(iq+1);
      lr = (tmp3*tmp3+1)*k*k + (4*tmp3+3)*k + ((tmp4 > tmp5)?tmp4:tmp5) + 4;
    }
  pdvar = k;
  pdx = (order == Nag_ColMajor)?k:n;
  x_size = (order == Nag_ColMajor)?pdx * n:pdx * k;
  var_size = pdvar * k;

  pdvar = k;
  pdx = (order == Nag_ColMajor)?k:n;
  x_size = (order == Nag_ColMajor)?pdx * n:pdx * k;
  var_size = pdvar * k;

  /* Allocate arrays */
  if (!(phi = NAG_ALLOC(ip*k*k, double)) ||
      !(r = NAG_ALLOC(lr, double)) ||
      !(theta = NAG_ALLOC(iq*k*k, double)) ||
      !(var = NAG_ALLOC(var_size, double)) ||
      !(x = NAG_ALLOC(x_size, double)) ||
      !(xmean = NAG_ALLOC(k, double)) ||
      !(state = NAG_ALLOC(lstate, Integer)))
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }

  /* Read in the AR parameters */
  for (l = 0; l < ip; l++)
    {
      for (i = 0; i < k; i++)
        {
          for (j = 0; j < k; j++)
#ifdef _WIN32
            scanf_s("%lf ", &PHI(i, j, l));
#else
            scanf("%lf ", &PHI(i, j, l));
#endif
        }
    }
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  /* Read in the MA parameters */
  if (iq > 0)
    {
      for (l = 0; l < iq; l++)
        {
          for (i = 0; i < k; i++)
            {
              for (j = 0; j < k; j++)
#ifdef _WIN32
                scanf_s("%lf ", &THETA(i, j, l));
#else
                scanf("%lf ", &THETA(i, j, l));
#endif
            }
        }
```

```
#ifdef _WIN32
      scanf_s("%*[^\n] ");
#else
      scanf("%*[^\n] ");
#endif
    }

  /* Read in the means */
  for (i = 0; i < k; i++)
#ifdef _WIN32
    scanf_s("%lf ", &xmean[i]);
#else
    scanf("%lf ", &xmean[i]);
#endif
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  /* Read in the variance / covariance matrix*/
  for (i = 0; i < k; i++)
    {
      for (j = 0; j <= i; j++)
#ifdef _WIN32
        scanf_s("%lf ", &VAR(i, j));
#else
        scanf("%lf ", &VAR(i, j));
#endif
    }
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  /* Initialise the generator to a repeatable sequence */
  nag_rand_init_repeatable(genid, subid, seed, lseed, state, &lstate, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_rand_init_repeatable (g05kfc).\n%s\n",
              fail.message);
      exit_status = 1;
      goto END;
    }

  /* Generate the first realization */
  mode = Nag_InitializeAndGenerate;
  nag_rand_varma(order, mode, n, k, xmean, ip, phi, iq, theta,
                 var, pdvar, r, lr, state, x, pdx, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_rand_varma (g05pjc).\n%s\n",
              fail.message);
      exit_status = 1;
      goto END;
    }

  /* Display the results */
  printf("  Realization Number 1\n");
  for (i = 0; i < k; i++)
    {
      printf("\n   Series number %3"NAG_IFMT"\n", i+1);
      printf("   -------------\n\n ");
      for (j = 0; j < n; j++)
        printf("%9.3f%s", X(i, j), (j+1)%8?" ":"\n ");
    }
  printf("\n");

  /* Generate a second realization */
  mode = Nag_ReGenerateFromReference;
```

```
  nag_rand_varma(order, mode, n, k, xmean, ip, phi, iq, theta,
                 var, pdvar, r, lr, state, x, pdx, &fail);
  if (fail.code != NE_NOERROR)
    {
      printf("Error from nag_rand_varma (g05pjc).\n%s\n",
              fail.message);
      exit_status = 1;
      goto END;
    }

  /* Display the results */
  printf("  Realization Number 2\n");
  for (i = 0; i < k; i++)
    {
      printf("\n   Series number %3"NAG_IFMT"\n", i+1);
      printf("   -------------\n\n ");
      for (j = 0; j < n; j++)
        printf("%9.3f%s", X(i, j), (j+1)%8?" ":"\n ");
    }
  printf("\n");

 END:
  NAG_FREE(phi);
  NAG_FREE(r);
  NAG_FREE(theta);
  NAG_FREE(var);
  NAG_FREE(x);
  NAG_FREE(xmean);
  NAG_FREE(state);

  return exit_status;
}
```

10.2 Program Data

```
nag_rand_varma (g05pjc) Example Program Data
 2 1 0 48                    : k, ip, iq, n
 0.80 0.07
 0.00 0.58                   : phi(,,1)
 5.00 9.00                   : xmean
 2.97
 0.64 5.38                   : var
```

10.3 Program Results

```
nag_rand_varma (g05pjc) Example Program Results

  Realization Number 1

   Series number   1
   -------------

     4.833     2.813     3.224     3.825     1.023     1.415     2.184     3.005
     5.547     4.832     4.705     5.484     9.407    10.335     8.495     7.478
     6.373     6.692     6.698     6.976     6.200     4.458     2.520     3.517
     3.054     5.439     5.699     7.136     5.750     8.497     9.563    11.604
     9.020    10.063     7.976     5.927     4.992     4.222     3.982     7.107
     3.554     7.045     7.025     4.106     5.106     5.954     8.026     7.212

   Series number   2
   -------------

     8.458     9.140    10.866    10.975     9.245     5.054     5.023    12.486
    10.534    10.590    11.376     8.793    14.445    13.237    11.030     8.405
     7.187     8.291     5.920     9.390    10.055     6.222     7.751    10.604
    12.441    10.664    10.960     8.022    10.073    12.870    12.665    14.064
    11.867    12.894    10.546    12.754     8.594     9.042    12.029    12.557
     9.746     5.487     5.500     8.629     9.723     8.632     6.383    12.484

  Realization Number 2
```

```
Series number   1
-------------

  5.396     4.811     2.685     5.824     2.449     3.563     5.663     6.209
  3.130     4.308     4.333     4.903     1.770     1.278     1.340    -0.527
  1.745     3.211     4.478     5.170     5.365     4.852     6.080     6.464
  2.765     2.148     6.641     7.224    10.316     7.102     5.604     3.934
  4.839     3.698     5.210     5.384     7.652     7.315     7.332     7.561
  7.537     7.788     6.868     7.575     6.108     6.188     8.132    10.310

Series number   2
-------------

 11.345    10.070    13.654    12.409    11.329    13.054    12.465     9.867
 10.263    13.394    10.553    10.331     7.814     8.747    10.025    11.167
 10.626     9.366     9.607     9.662    10.492    10.766    11.512    10.813
 10.799     8.780     9.221    14.245    11.575    10.620     8.282     5.447
  9.935     9.386    11.627    10.066    11.394     7.951     7.907    12.616
 15.246     9.962    13.216    11.350    11.227     6.021     6.968    12.428
```