NAG Library Function Document

nag_rand_exp_smooth (g05pmc)

1 Purpose
nag_rand_exp_smooth (g05pmc) simulates from an exponential smoothing model, where the model uses either single exponential, double exponential or a Holt–Winters method.

2 Specification

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

void nag_rand_exp_smooth (Nag_InitialValues mode, Integer n,
                        Nag_ExpSmoothType itype, Integer p, const double param[],
                        const double init[], double var, double r[], Integer state[],
                        const double e[], Integer en, double x[], NagError *fail)
```

3 Description

nag_rand_exp_smooth (g05pmc) returns \( \{x_t : t = 1, 2, \ldots, n\} \), a realization of a time series from an exponential smoothing model defined by one of five smoothing functions:

- **Single Exponential Smoothing**
  \[
  x_t = m_{t-1} + \epsilon_t \\
  m_t = \alpha x_t + (1 - \alpha)m_{t-1}
  \]

- **Brown Double Exponential Smoothing**
  \[
  x_t = m_{t-1} + \frac{r_{t-1}}{\alpha} + \epsilon_t \\
  m_t = \alpha x_t + (1 - \alpha)m_{t-1} \\
  r_t = \alpha(m_t - m_{t-1}) + (1 - \alpha)r_{t-1}
  \]

- **Linear Holt Exponential Smoothing**
  \[
  x_t = m_{t-1} + \phi r_{t-1} + \epsilon_t \\
  m_t = \alpha x_t + (1 - \alpha)(m_{t-1} + \phi r_{t-1}) \\
  r_t = \gamma(m_t - m_{t-1}) + (1 - \gamma)\phi r_{t-1}
  \]

- **Additive Holt–Winters Smoothing**
  \[
  x_t = m_{t-1} + \phi r_{t-1} + s_{t-1-p} + \epsilon_t \\
  m_t = \alpha(x_t - s_{t-p}) + (1 - \alpha)(m_{t-1} + \phi r_{t-1}) \\
  r_t = \gamma(m_t - m_{t-1}) + (1 - \gamma)\phi r_{t-1} \\
  s_t = \beta(x_t - m_t) + (1 - \beta)s_{t-p}
  \]

- **Multiplicative Holt–Winters Smoothing**
  \[
  x_t = (m_{t-1} + \phi r_{t-1}) \times s_{t-1-p} + \epsilon_t \\
  m_t = \alpha_x/s_{t-p} + (1 - \alpha)(m_{t-1} + \phi r_{t-1}) \\
  r_t = \gamma(m_t - m_{t-1}) + (1 - \gamma)\phi r_{t-1} \\
  s_t = \beta x_t/m_t + (1 - \beta)s_{t-p}
  \]

where \( m_t \) is the mean, \( r_t \) is the trend and \( s_t \) is the seasonal component at time \( t \) with \( p \) being the seasonal order. The errors, \( \epsilon_t \) are either drawn from a normal distribution with mean zero and variance \( \sigma^2 \) or randomly sampled, with replacement, from a user-supplied vector.

4 References

5 Arguments

1: mode – Nag_InitialValues
   
   On entry: indicates if nag_rand_exp_smooth (g05pmc) is continuing from a previous call or, if not, how the initial values are computed.

   mode = Nag_InitialValuesSupplied
   Values for $m_0$, $r_0$ and $s_j$, for $j = 0, 1, \ldots, p - 1$, are supplied in init.

   mode = Nag_ContinueNoUpdate
   nag_rand_exp_smooth (g05pmc) continues from a previous call using values that are supplied in r. r is not updated.

   mode = Nag_ContinueAndUpdate
   nag_rand_exp_smooth (g05pmc) continues from a previous call using values that are supplied in r. r is updated.

   Constraint: mode = Nag_InitialValuesSupplied, Nag_ContinueNoUpdate or Nag_ContinueAndUpdate.

2: n – Integer
   
   On entry: the number of terms of the time series being generated.

   Constraint: $n \geq 0$.

3: itype – Nag_ExpSmoothType
   
   On entry: the smoothing function.

   itype = Nag_SingleExponential
   Single exponential.

   itype = Nag_BrownsExponential
   Brown’s double exponential.

   itype = Nag_LinearHolt
   Linear Holt.

   itype = Nag_AdditiveHoltWinters
   Additive Holt–Winters.

   itype = Nag_MultiplicativeHoltWinters
   Multiplicative Holt–Winters.

   Constraint: itype = Nag_SingleExponential, Nag_BrownsExponential, Nag_LinearHolt, Nag_AdditiveHoltWinters or Nag_MultiplicativeHoltWinters.

4: p – Integer
   
   On entry: if itype = Nag_AdditiveHoltWinters or Nag_MultiplicativeHoltWinters, the seasonal order, $p$, otherwise $p$ is not referenced.

   Constraint: if itype = Nag_AdditiveHoltWinters or Nag_MultiplicativeHoltWinters, $p > 1$.

5: param[dim] – const double
   
   Input

   Note: the dimension, dim, of the array param must be at least
   1 when itype = Nag_SingleExponential or Nag_BrownsExponential;
   3 when itype = Nag_LinearHolt;
   4 when itype = Nag_AdditiveHoltWinters or Nag_MultiplicativeHoltWinters.

   On entry: the smoothing parameters.

   If itype = Nag_SingleExponential or Nag_BrownsExponential, $param[0] = \alpha$ and any remaining elements of param are not referenced.


Constraints:
- if `itype` = Nag_SingleExponential, 0.0 ≤ α ≤ 1.0;
- if `itype` = Nag_BrownsExponential, 0.0 < α ≤ 1.0;
- if `itype` = Nag_LinearHolt, 0.0 ≤ α ≤ 1.0 and 0.0 ≤ γ ≤ 1.0 and φ ≥ 0.0;
- if `itype` = Nag_AdditiveHoltWinters or Nag_MultiplicativeHoltWinters, 0.0 ≤ α ≤ 1.0 and 0.0 ≤ γ ≤ 1.0 and 0.0 ≤ β ≤ 1.0 and φ ≥ 0.0.

6: `init[dim]` – const double

**Input**

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

1 when `itype` = Nag_SingleExponential;
2 when `itype` = Nag_BrownsExponential or Nag_LinearHolt;
2 + `p` when `itype` = Nag_AdditiveHoltWinters or Nag_MultiplicativeHoltWinters.

On entry: if `mode` = Nag_InitialValuesSupplied, the initial values for `m0`, `r0` and `s−j`, for `j = 0, 1, ..., p − 1`, used to initialize the smoothing.

If `itype` = Nag_SingleExponential, `init[0]` = `m0` and any remaining elements of `init` are not referenced.

If `itype` = Nag_BrownsExponential or Nag_LinearHolt, `init[0]` = `m0` and `init[1]` = `r0` and any remaining elements of `init` are not referenced.

If `itype` = Nag_AdditiveHoltWinters or Nag_MultiplicativeHoltWinters, `init[0]` = `m0`, `init[1]` = `r0` and `init[2]` to `init[2 + p − 1]` hold the values for `s−j`, for `j = 0, 1, ..., p − 1`. Any remaining elements of `init` are not referenced.

7: `var` – double

**Input**

On entry: the variance, σ² of the Normal distribution used to generate the errors εᵢ. If `var` ≤ 0.0 then Normally distributed errors are not used.

8: `r[dim]` – double

**Input/Output**

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

13 when `itype` = Nag_SingleExponential, Nag_BrownsExponential or Nag_LinearHolt;
13 + `p` when `itype` = Nag_AdditiveHoltWinters or Nag_MultiplicativeHoltWinters.

On entry: if `mode` = Nag_ContinueNoUpdate or Nag_ContinueAndUpdate, `r` must contain the values as returned by a previous call to nag_rand_exp_smooth (g05pmc), `r` need not be set otherwise.

On exit: if `mode` = Nag_ContinueNoUpdate, `r` is unchanged. Otherwise, `r` contains the information on the current state of smoothing.

Constraint: if `mode` = Nag_ContinueNoUpdate or Nag_ContinueAndUpdate, `r` must have been initialized by at least one call to nag_rand_exp_smooth (g05pmc) or nag_tsa_exp_smooth (g13amc) with `mode` ≠ Nag_ContinueNoUpdate, and `r` must not have been changed since that call.

9: `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.

10: \( e[\text{en}] \) – const double

**Input**

On entry: if \( \text{en} > 0 \) and \( \text{var} \leq 0.0 \), a vector from which the errors, \( \epsilon_t \), are randomly drawn, with replacement.

If \( \text{en} \leq 0 \), \( e \) is not referenced.

11: \( \text{en} \) – Integer

**Input**

On entry: if \( \text{en} > 0 \), then the length of the vector \( e \).

If both \( \text{var} \leq 0.0 \) and \( \text{en} \leq 0 \) then \( \epsilon_t = 0.0 \), for \( t = 1, 2, \ldots, n \).

12: \( x[n] \) – double

**Output**

On exit: the generated time series, \( x_t \), for \( t = 1, 2, \ldots, n \).

13: \( \text{fail} \) – 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 <value> had an illegal value.

**NE_ENUM_INT**

On entry, \( \text{itype} = \langle \text{value} \rangle \) and \( p = \langle \text{value} \rangle \).

Constraint: if \( \text{itype} = \text{Nag_AdditiveHoltWinters} \) or \( \text{Nag_MultiplicativeHoltWinters} \), \( p > 1 \).

On entry, \( p = \langle \text{value} \rangle \).

Constraint: if \( \text{itype} = \text{Nag_AdditiveHoltWinters} \) or \( \text{Nag_MultiplicativeHoltWinters} \), \( p \geq 2 \).

**NE_INT**

On entry, \( n = \langle \text{value} \rangle \).

Constraint: \( n \geq 0 \).

**NE_INT_ARRAY**

On entry, some of the elements of the array \( r \) have been corrupted or have not been initialized.

**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, \( \text{state} \) vector has been corrupted or not initialized.
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_REAL_ARRAY
Model unsuitable for multiplicative Holt–Winter, try a different set of parameters.

On entry, param[\textit{value}] = \langle value \rangle.
Constraint: 0 \leq \textit{param}[i] \leq 1.

Constraint: if \textit{itype} = \text{Nag_BrownsExponential}, 0 < \textit{param}[i] \leq 1.
Constraint: \textit{param}[i] \geq 0.

7 Accuracy
Not applicable.

8 Parallelism and Performance

\texttt{nag_rand_exp_smooth (g05pmc)} 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 function. Please also consult the Users’ Note for your implementation for any additional implementation-specific information.

9 Further Comments
None.

10 Example
This example reads 11 observations from a time series relating to the rate of the earth’s rotation about its polar axis and fits an exponential smoothing model using \texttt{nag_tsa_exp_smooth (g13amc)}.

\texttt{nag_rand_exp_smooth (g05pmc)} is then called multiple times to obtain simulated forecast confidence intervals.

10.1 Program Text
/* \texttt{nag_rand_exp_smooth (g05pmc)} 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 <nag_stdlib.h>
#include <nag01.h>
#include <nag05.h>
#include <nag13.h>
#define BLIM(I, J) blim[J*2 + I]
#define BSIM(I, J) bsim[J*nsim + I]
#define GLIM(I, J) glim[J*2 + I]
#define GSIM(I, J) gsim[J*nsim + I]
int main(void)
{
  /* Integer scalar and array declarations */
  Integer exit_status = 0;
  Integer en, i, ival, j, k, lstate, n, nf, nq, nsim, p;
  Integer *state = 0;
  /* NAG structures */
  NagError fail;
  Nag_TailProbability tail;
  Nag_InitialValues mode;
  Nag_ExpSmoothType itype;
  /* Double scalar and array declarations */
  double ad, alpha, dv, tmp, var, z, bvar;
  double *blim = 0, *bsim = 0, *e = 0, *fse = 0, *fv = 0;
  double *glim = 0, *gsim = 0, *init = 0, *param = 0, *r = 0;
  double *res = 0, *tsim1 = 0, *tsim2 = 0, *y = 0, *yhat = 0;
  double q[2];
  /* Character scalar and array declarations */
  char smode[40], sitype[40];
  /* 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_exp_smooth (g05pmc) 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", fail.message);
    exit_status = 1;
    goto END;
  }
  /* Skip headings in data file */
  #ifdef _WIN32
    scanf_s("%*[\n"]);
  #else
    scanf("%*[\n"]);
  #endif
  /* Read in the initial arguments and check array sizes */
  #ifdef _WIN32
    scanf_s("%39s%39s"NAG_IFMT"%NAG_IFMT"%lf%*[\n"] , smode, _countof(smode),
    sitype, _countof(sitype), &n, &nf, &nsim, &alpha);
  #else
    scanf("%39s%39s"NAG_IFMT"%NAG_IFMT"%lf%*[\n"] , smode, sitype, &n, &nf, &nsim, &alpha);
  #endif
  /* nag_enum_name_to_value (x04nac).
   * Converts NAG enum member name to value */
  mode = (Nag_InitialValues) nag_enum_name_to_value(smode);
  itype = (Nag_ExpSmoothType) nag_enum_name_to_value(sitype);
  /* Allocate arrays */
  if (!blim = NAG_ALLOC(2*nf, double) ||
      !bsim = NAG_ALLOC(nsim*nf, double) ||
      !e = NAG_ALLOC(1, double) ||
      !fse = NAG_ALLOC(nf, double) ||
      !fv = NAG_ALLOC(nf, double) ||

!(glim = NAG_ALLOC(2*nf, double)) ||
!(gsim = NAG_ALLOC(nsim*nf, double)) ||
!(res = NAG_ALLOC(n, double)) ||
!(tsim1 = NAG_ALLOC(nf, double)) ||
!(tsim2 = NAG_ALLOC(nf, double)) ||
!(y = NAG_ALLOC(n, double)) ||
!(yhat = NAG_ALLOC(n, double)) ||
!(state = NAG_ALLOC(lstate, Integer)))
{
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
}

/* 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;
}
for (i = 0; i < n; i++)
#elseif _WIN32
    scanf_s("%lf ", &y[i]);
#else
    scanf("%lf ", &y[i]);
#endif
#elseif _WIN32
    scanf_s("%*[\n ] ");
#else
    scanf("%*[\n ] ");
#endif

/* Read in the itype dependent arguments (skipping headings) */
#elseif _WIN32
    scanf_s("%*[\n ] ");
#else
    scanf("%*[\n ] ");
#endif
if (itype == Nag_SingleExponential)
{
    /* Single exponential smoothing required */
    if (!(param = NAG_ALLOC(1, double)))
    {
        printf("Allocation failure\n");
        exit_status = -1;
        goto END;
    }
#elseif _WIN32
    scanf_s("%lf*\n", &param[0]);
#else
    scanf("%lf*\n", &param[0]);
#endif
    p = 0;
    ival = 1;
} else if (itype == Nag_BrownsExponential)
{
    /* Browns exponential smoothing required */
    if (!(param = NAG_ALLOC(2, double)))
    {
        printf("Allocation failure\n");
        exit_status = -1;
        goto END;
    }
#elseif _WIN32
    scanf_s("%lf %lf*\n", &param[0], &param[1]);
#else
    scanf("%lf %lf*\n", &param[0], &param[1]);
#endif
 scanf("%lf %lf\n", &param[0], &param[1]);
#endif

 p = 0;
 ival = 2;
 }
 else if (itype == Nag_LinearHolt)
 {
  /* Linear Holt smoothing required */
  if (!(param = NAG_ALLOC(3, double)))
  {
   printf("Allocation failure\n");
   exit_status = -1;
   goto END;
  }
  #ifdef _WIN32
  scanf_s("%lf %lf %lf\n", &param[0], &param[1], &param[2]);
  #else
  scanf("%lf %lf %lf\n", &param[0], &param[1], &param[2]);
  #endif
  p = 0;
  ival = 2;
 }
 else if (itype == Nag_AdditiveHoltWinters)
 {
  /* Additive Holt Winters smoothing required */
  if (!(param = NAG_ALLOC(4, double)))
  {
   printf("Allocation failure\n");
   exit_status = -1;
   goto END;
  }
  #ifdef _WIN32
  scanf_s("%lf %lf %lf %NAG_IFMT%\n", &param[0], &param[1],
          &param[2], &param[3], &p);
  #else
  scanf("%lf %lf %lf %NAG_IFMT%\n", &param[0], &param[1],
          &param[2], &param[3], &p);
  #endif
  ival = p+2;
 }
 else if (itype == Nag_MultiplicativeHoltWinters)
 {
  /* Multiplicative Holt Winters smoothing required */
  if (!(param = NAG_ALLOC(4, double)))
  {
   printf("Allocation failure\n");
   exit_status = -1;
   goto END;
  }
  #ifdef _WIN32
  scanf_s("%lf %lf %lf %NAG_IFMT%\n", &param[0], &param[1],
          &param[2], &param[3], &p);
  #else
  scanf("%lf %lf %lf %NAG_IFMT%\n", &param[0], &param[1],
          &param[2], &param[3], &p);
  #endif
  ival = p+2;
 }
 else
{
  printf("%s is an unknown type\n", sitype);
  exit_status = -1;
  goto END;
}

 /* Allocate arrays */
 if (!((init = NAG_ALLOC(p+2, double)) ||
      (r = NAG_ALLOC(p+13, double))))
   {
    printf("Allocation failure\n");
    exit_status = -1;
   }
/* Read in the mode dependent arguments (skipping headings) */
#endif
    scanf_s("%*[\n] ");
#else
    scanf("%*[\n] ");
#endif
    if (mode == Nag_InitialValuesSupplied)
    {
        /* User supplied initial values*/
        for (i = 0; i < ival; i++)
        #ifdef _WIN32
            scanf_s("%lf ", &init[i]);
        #else
            scanf("%lf ", &init[i]);
        #endif
        #ifdef _WIN32
            scanf_s("%*[\n] ");
        #else
            scanf("%*[\n] ");
        #endif
    }
    else if (mode == Nag_ContinueAndUpdate)
    {
        /* Continuing from a previously saved R */
        for (i = 0; i < p+13; i++)
        #ifdef _WIN32
            scanf_s("%lf ", &r[i]);
        #else
            scanf("%lf ", &r[i]);
        #endif
        #ifdef _WIN32
            scanf_s("%*[\n] ");
        #else
            scanf("%*[\n] ");
        #endif
    }
    else if (mode == Nag_EstimateInitialValues)
    {
        /* Initial values calculated from first k observations */
        #ifdef _WIN32
            scanf_s("%"NAG_IFMT"%*[\n] ", &k);
        #else
            scanf("%"NAG_IFMT"%*[\n] ", &k);
        #endif
    }
    else
    {
        printf("%s is an unknown mode\n", smode);
        exit_status = -1;
        goto END;
    }

    /* Fit a smoothing model (parameter r in
    * nag_rand_exp_smooth (g05pmc) and state in g13amc are in
    * the same format) */
    nag_tsa_exp_smooth(mode, itype, p, param, n, y, k, init, nf, fv, fse, yhat,
    res, &dv, &ad, r, &fail);
    if (fail.code != NE_NOERROR)
    {
        printf("Error from nag_tsa_exp_smooth (g13amc).\n%s\n", fail.message);
        exit_status = 1;
        goto END;
    }

    /* Simulate forecast values from the model, and don’t update r */
    var = dv*dv;
    en = n;
/* Change the mode used to continue from fit model */
mode = Nag_ContinueAndUpdate;

/* Simulate nsim forecasts */
for (i = 0; i < nsim; i++)
{
    /* Simulations assuming gaussian errors */
    nag_rand_exp_smooth(mode, nf, itype, p, param, init, var, r, state,
                         e, 0, tsim1, &fail);
    if (fail.code != NE_NOERROR)
    {
        printf("Error from nag_rand_exp_smooth (g05pmc).\n%s\n", fail.message);
        exit_status = 1;
        goto END;
    }

    /* Bootstrapping errors */
    bvar = 0.0e0;
    nag_rand_exp_smooth(mode, nf, itype, p, param, init, bvar, r, state,
                         res, en, tsim2, &fail);
    if (fail.code != NE_NOERROR)
    {
        printf("Error from nag_rand_exp_smooth (g05pmc).\n%s\n", fail.message);
        exit_status = 1;
        goto END;
    }

    /* Copy and transpose the simulated values */
    for (j = 0; j < nf; j++)
    {
        GSIM(i, j) = tsim1[j];
        BSIM(i, j) = tsim2[j];
    }

    /* Calculate CI based on the quantiles for each simulated forecast */
    q[0] = alpha/2.0e0;
    q[1] = 1.0e0-q[0];
    nq = 2;
    for (i = 0; i < nf; i++)
    {
        nag_double_quantiles(nsim, &GSIM(0, i), nq, q, &GLIM(0, i), &fail);
        if (fail.code != NE_NOERROR)
        {
            printf("Error from nag_double_quantiles (g01amc).\n%s\n", fail.message);
            exit_status = 1;
            goto END;
        }

        nag_double_quantiles(nsim, &BSIM(0, i), nq, q, &BLIM(0, i), &fail);
        if (fail.code != NE_NOERROR)
        {
            printf("Error from nag_double_quantiles (g01amc).\n%s\n", fail.message);
            exit_status = 1;
            goto END;
        }
    }

    /* Display the forecast values and associated prediction intervals */
    printf("Initial values used:\n\n");
    for (i = 0; i < ival; i++)
        printf("%4"NAG_IFMT" %12.3f \n", i+1, init[i]);
    printf("\n");
    printf("Mean Deviation = %13.4e\n", dv);
    printf("Absolute Deviation = %13.4e\n", ad);
    printf("Observed 1-Step\n");
    printf(" Period Values Forecast Residual\n");
for (i = 0; i < n; i++)
    printf("%4"NAG_IFMT" %11.3f %11.3f %11.3f\n", i+1, y[i],
            yhat[i], res[i]);
printf("\n");
tail = Nag_LowerTail;
z = nag_deviates_normal(tail, q[1], &fail);
if (fail.code != NE_NOERROR)
{
    printf("Error from nag_deviates_normal (g01fac).\n%"NAG.getLine()s\n", fail.message);
    exit_status = 1;
    goto END;
}
printf(" Simulated CI\n" Simulated CI\n");
printf(" Obs. Forecast Estimated CI (Gaussian Errors) (Bootstrap Errors)\n" Obs. Forecast Estimated CI (Gaussian Errors) (Bootstrap Errors)\n");
for (i = 0; i < nf; i++)
{
    tmp = z*fse[i];
    printf("%3"NAG_IFMT" %10.3f %10.3f %10.3f %10.3f %10.3f %10.3f %10.3f\n", n+i+1, fv[i], fv[i]-tmp, fv[i]+tmp, GLIM(0, i), GLIM(1, i), BLIM(0, i), BLIM(1, i));
}
printf(" %5.1f%% CIs were produced\n", 100.0e0*(1.0e0 - alpha));
END:
NAG_FREE(blim);
NAG_FREE(bsim);
NAG_FREE(e);
NAG_FREE(fse);
NAG_FREE(fv);
NAG_FREE(glim);
NAG_FREE(qsim);
NAG_FREE(init);
NAG_FREE(param);
NAG_FREE(r);
NAG_FREE(res);
NAG_FREE(tsim1);
NAG_FREE(tsim2);
NAG_FREE(y);
NAG_FREE(yhat);
NAG_FREE(state);
return exit_status;

10.2 Program Data

nag_rand_exp_smooth (g05pmc) Example Program Data

Nag_EstimateInitialValues Nag_LinearHolt
11 5 100 0.05 : mode,itype,n,nf,nsim,alpha
180 135 213 181 148 228 225 198 200 187 : y
dependent arguments for itype=Nag_LinearHolt
0.01 1.0 1.0 : param[0],param[1],param[2]
dependent arguments for mode=Nag_ContinueAndUpdate
11 : k

10.3 Program Results

nag_rand_exp_smooth (g05pmc) Example Program Results

Initial values used:
  1  168.018
  2  3.800

Mean Deviation = 2.5473e+01
Absolute Deviation = 2.1233e+01

Observed 1-Step
<table>
<thead>
<tr>
<th>Period</th>
<th>Values</th>
<th>Forecast</th>
<th>Residual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>180.000</td>
<td>171.818</td>
<td>8.182</td>
</tr>
<tr>
<td>2</td>
<td>135.000</td>
<td>175.782</td>
<td>-40.782</td>
</tr>
<tr>
<td>3</td>
<td>213.000</td>
<td>178.848</td>
<td>34.152</td>
</tr>
<tr>
<td>4</td>
<td>181.000</td>
<td>183.005</td>
<td>-2.005</td>
</tr>
<tr>
<td>5</td>
<td>148.000</td>
<td>186.780</td>
<td>-38.780</td>
</tr>
<tr>
<td>6</td>
<td>204.000</td>
<td>189.800</td>
<td>14.200</td>
</tr>
<tr>
<td>7</td>
<td>228.000</td>
<td>193.492</td>
<td>34.508</td>
</tr>
<tr>
<td>8</td>
<td>225.000</td>
<td>197.732</td>
<td>27.268</td>
</tr>
<tr>
<td>9</td>
<td>198.000</td>
<td>202.172</td>
<td>-4.172</td>
</tr>
<tr>
<td>10</td>
<td>200.000</td>
<td>206.256</td>
<td>-6.256</td>
</tr>
<tr>
<td>11</td>
<td>187.000</td>
<td>210.256</td>
<td>-23.256</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obs.</th>
<th>Forecast</th>
<th>Estimated CI</th>
<th>Simulated CI (Gaussian Errors)</th>
<th>Simulated CI (Bootstrap Errors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>213.854</td>
<td>163.928</td>
<td>263.781</td>
<td>161.431 258.001 173.073 248.363</td>
</tr>
<tr>
<td>13</td>
<td>217.685</td>
<td>167.748</td>
<td>267.622</td>
<td>172.660 262.100 177.311 252.638</td>
</tr>
<tr>
<td>14</td>
<td>221.516</td>
<td>171.556</td>
<td>271.475</td>
<td>169.259 263.107 179.344 256.921</td>
</tr>
<tr>
<td>15</td>
<td>225.346</td>
<td>175.347</td>
<td>275.345</td>
<td>180.721 272.776 183.672 260.804</td>
</tr>
<tr>
<td>16</td>
<td>229.177</td>
<td>179.115</td>
<td>279.238</td>
<td>184.790 263.591 186.398 264.173</td>
</tr>
</tbody>
</table>

95.0% CIs were produced

---

**Example Program**

Exponential Smoothing

(95% confidence intervals (CIs) are shown)