NAG Library Function Document

nag_2d_spline_fit_ts_scat (e02jdc)

Note: *this function uses* **optional parameters** *to define choices in the problem specification and in the details of the algorithm. If you wish to use* default *settings for all of the optional parameters, you need only read Sections 1 to 10 of this document. If, however, you wish to reset some or all of the settings please refer to Section 11 for a detailed description of the specification of the optional parameters produced by the function*.

1 Purpose

nag_2d_spline_fit_ts_scat (e02jdc) computes a spline approximation to a set of scattered data using a two-stage approximation method.

The computational complexity of the method grows linearly with the number of data points; hence large datasets are easily accommodated.

2 Specification

```
#include <nag.h>
#include <nage02.h>
void nag_2d_spline_fit_ts_scat (Integer n, const double x[],
     const double y[], const double f[], Integer lsminp, Integer lsmaxp,
     Integer nxcels, Integer nycels, Integer lcoefs, double coefs[],
     Integer iopts[], double opts[], NagError *fail)
```

Before calling nag_2d_spline_fit_ts_scat (e02jdc), nag_fit_opt_set (e02zkc) must be called with **optstr** set to `"Initialize = e02jdc"`. Settings for optional algorithmic arguments may be specified by calling nag_fit_opt_set (e02zkc) before a call to nag_2d_spline_fit_ts_scat (e02jdc).

3 Description

nag_2d_spline_fit_ts_scat (e02jdc) determines a smooth bivariate spline approximation to a set of data points (x_i, y_i, f_i), for $i = 1, 2, \ldots, n$. Here, 'smooth' means C^1 or C^2. (You may select the degree of smoothing using the optional parameter **Global Smoothing Level**.)

The approximation domain is the bounding box $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$, where $x_{\min}$ (respectively $y_{\min}$) and $x_{\max}$ (respectively $y_{\max}$) denote the lowest and highest data values of the (x_i) (respectively (y_i)).

The spline is computed by local approximations on a uniform triangulation of the bounding box. These approximations are extended to a smooth spline representation of the surface over the domain. The local approximation scheme is controlled by the optional parameter **Local Method**. The schemes provided are: by least squares polynomial approximation (Davydov and Zeilfelder (2004)); by hybrid polynomial and radial basis function (RBF) approximation (Davydov *et al.* (2006)); or by pure RBF approximation (Davydov *et al.* (2005)).

The two-stage approximation method employed by nag_2d_spline_fit_ts_scat (e02jdc) is derived from the TSFIT package of O. Davydov and F. Zeilfelder.

Values of the computed spline can subsequently be computed by calling nag_2d_spline_ts_eval (e02jec) or nag_2d_spline_ts_eval_rect (e02jfc).

4 References

Davydov O, Morandi R and Sestini A (2006) Local hybrid approximation for scattered data fitting with bivariate splines *Comput. Aided Geom. Design* **23** 703–721

Davydov O, Sestini A and Morandi R (2005) Local RBF approximation for scattered data fitting with bivariate splines *Trends and Applications in Constructive Approximation* M. G. de Bruin, D. H. Mache, and J. Szabados, Eds **ISNM Vol. 151** Birkhauser 91–102

Davydov O and Zeilfelder F (2004) Scattered data fitting by direct extension of local polynomials to bivariate splines *Advances in Comp. Math.* **21** 223–271

5 Arguments

1: **n** – Integer *Input*

On entry: n, the number of data values to be fitted.

Constraint: $\mathbf{n} > 1$.

2: **x**[**n**] – const double *Input*
3: **y**[**n**] – const double *Input*
4: **f**[**n**] – const double *Input*

On entry: the (x_i, y_i, f_i) data values to be fitted.

Constraint: $\mathbf{x}[j-1] \neq \mathbf{x}[0]$ for some $j = 2, \ldots, n$ and $\mathbf{y}[k-1] \neq \mathbf{y}[0]$ for some $k = 2, \ldots, n$; i.e., there are at least two distinct x and y values.

5: **lsminp** – Integer *Input*
6: **lsmaxp** – Integer *Input*

On entry: are control parameters for the local approximations.

Each local approximation is computed on a local domain containing one of the triangles in the discretization of the bounding box. The size of each local domain will be adaptively chosen such that if it contains fewer than **lsminp** sample points it is expanded, else if it contains greater than **lsmaxp** sample points a thinning method is applied. **lsmaxp** mainly controls computational cost (in that working with a thinned set of points is cheaper and may be appropriate if the input data is densely distributed), while **lsminp** allows handling of different types of scattered data.

Setting **lsmaxp** $<$ **lsminp**, and therefore forcing either expansion or thinning, may be useful for computing initial coarse approximations. In general smaller values for these arguments reduces cost.

A calibration procedure (experimenting with a small subset of the data to be fitted and validating the results) may be needed to choose the most appropriate values for **lsminp** and **lsmaxp**.

Constraints:

$1 \leq \mathbf{lsminp} \leq \mathbf{n}$;
$\mathbf{lsmaxp} \geq 1$.

7: **nxcels** – Integer *Input*
8: **nycels** – Integer *Input*

On entry: **nxcels** (respectively **nycels**) is the number of cells in the x (respectively y) direction that will be used to create the triangulation of the bounding box of the domain of the function to be fitted.

Greater efficiency generally comes when **nxcels** and **nycels** are chosen to be of the same order of magnitude and are such that **n** is $O(\mathbf{nxcels} \times \mathbf{nycels})$. Thus for a 'square' triangulation — when $\mathbf{nxcels} = \mathbf{nycels}$ — the quantities $\sqrt{\mathbf{n}}$ and **nxcels** should be of the same order of magnitude. See also Section 9.

Constraints:

nxcels ≥ 1;
nycels ≥ 1.

9: **lcoefs** – Integer *Input*

10: **coefs**[**lcoefs**] – double *Output*

On exit: if **fail**.**code** = NE_NOERROR on exit, **coefs** contains the computed spline coefficients.

Constraints:

if **Global Smoothing Level** $= 1$,
lcoefs $\geq (((\mathbf{nxcels}+2)\times(\mathbf{nycels}+2)+1)/2)\times 10+1$;
if **Global Smoothing Level** $= 2$,
lcoefs $\geq 28\times(\mathbf{nxcels}+2)\times(\mathbf{nycels}+2)\times 4+1$.

11: **iopts**[*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 **iopts** in the previous call to nag_fit_opt_set (e02zkc).

On entry: the contents of **iopts** MUST NOT be modified in any way either directly or indirectly, by further calls to nag_fit_opt_set (e02zkc), before calling either or both of the evaluation routines nag_2d_spline_ts_eval (e02jec) and nag_2d_spline_ts_eval_rect (e02jfc).

12: **opts**[*dim*] – double *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 **opts** in the previous call to nag_fit_opt_set (e02zkc).

On entry: the contents of **opts** MUST NOT be modified in any way either directly or indirectly, by further calls to nag_fit_opt_set (e02zkc), before calling either or both of the evaluation routines nag_2d_spline_ts_eval (e02jec) and nag_2d_spline_ts_eval_rect (e02jfc).

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

The NAG error argument (see Section 2.7 in How to Use the NAG Library and its Documentation).

6 Error Indicators and Warnings

NE_ALL_ELEMENTS_EQUAL

On entry, all elements of **x** or of **y** are equal.

NE_ALLOC_FAIL

Dynamic memory allocation failed.
See Section 3.2.1.2 in How to Use the NAG Library and its Documentation for further information.

NE_BAD_PARAM

On entry, argument ⟨*value*⟩ had an illegal value.

NE_INITIALIZATION

Option arrays are not initialized or are corrupted.

NE_INT

On entry, **lcoefs** $= \langle value \rangle$.
Constraint:
if **Global Smoothing Level** $= 1$,
lcoefs $\geq (((\mathbf{nxcels}+2) \times (\mathbf{nycels}+2)+1)/2) \times 10+1$;
if **Global Smoothing Level** $= 2$,
lcoefs $\geq 28 \times (\mathbf{nxcels}+2) \times (\mathbf{nycels}+2) \times 4+1$.

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

On entry, **n** $= \langle value \rangle$.
Constraint: **n** > 1.

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

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

NE_INT_2

On entry, **lsminp** $= \langle value \rangle$ and **n** $= \langle value \rangle$.
Constraint: $1 \leq$ **lsminp** $\leq$ **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 algorithmic failure was encountered. Please contact NAG.

An unexpected error has been triggered by this function. Please contact NAG.
See Section 3.6.6 in How to Use the NAG Library and its Documentation for further information.

NE_INVALID_OPTION

The selected radial basis function cannot be used with the RBF local method.

The value of optional parameter **Polynomial Starting Degree** was invalid.

NE_NO_LICENCE

Your licence key may have expired or may not have been installed correctly.
See Section 3.6.5 in How to Use the NAG Library and its Documentation for further information.

7 Accuracy

Technical results on error bounds can be found in Davydov and Zeilfelder (2004), Davydov *et al.* (2006) and Davydov *et al.* (2005).

Local approximation by polynomials of degree d for n data points has optimal approximation order $n^{-(d+1)/2}$. The improved approximation power of hybrid polynomial/RBF and of pure RBF approximations is shown in Davydov *et al.* (2006) and Davydov *et al.* (2005).

The approximation error for C^1 global smoothing is $O\left(n^{-2}\right)$. For C^2 smoothing the error is $O\left(n^{-7/2}\right)$ when **Supersmooth C2** $=$ YES and $O\left(n^{-3}\right)$ when **Supersmooth C2** $=$ NO.

Whether maximal accuracy is achieved depends on the distribution of the input data and the choices of the algorithmic parameters. The references above contain extensive numerical tests and further technical discussions of how best to configure the method.

8 Parallelism and Performance

nag_2d_spline_fit_ts_scat (e02jdc) is threaded by NAG for parallel execution in multithreaded implementations of the NAG Library.

nag_2d_spline_fit_ts_scat (e02jdc) 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' Notefor your implementation for any additional implementation-specific information.

9 Further Comments

n-linear complexity and memory usage can be attained for sufficiently dense input data if the triangulation parameters **nxcels** and **nycels** are chosen as recommended in their descriptions above. For sparse input data on such triangulations, if many expansion steps are required (see **lsminp**) the complexity may rise to be loglinear.

Parts of the pure RBF method used when **Local Method** = RBF have n-quadratic memory usage.

Note that if **Local Method** = HYBRID and an initial hybrid approximation is deemed unreliable (see the description of optional parameter **Minimum Singular Value LHA**), a pure polynomial approximation will be used instead on that local domain.

10 Example

The Franke function

$$\begin{aligned} f(x,y) \quad = \quad & 0.75\exp\Big(-\Big((9x-2)^2+(9y-2)^2\Big)/4\Big)+ \\ & 0.75\exp\Big(-(9x+1)^2/49-(9y+1)/10\Big)+ \\ & 0.5\exp\Big(-\Big((9x-7)^2+(9y-3)^2\Big)/4\Big)- \\ & 0.2\exp\Big(-(9x-4)^2-(9y-7)^2\Big) \end{aligned}$$

is widely used for testing surface-fitting methods. The example program randomly generates a number of points on this surface. From these a spline is computed and then evaluated at a vector of points and on a mesh.

10.1 Program Text

```
/* nag_2d_spline_fit_ts_scat (e02jdc) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 */

#include <math.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nage02.h>
#include <nagf16.h>
#include <nagg05.h>
#include <string.h>

static Integer generate_data(Integer *n, double **x, double **y, double **f,
                             Integer *lsminp, Integer *lsmaxp,
                             Integer *nxcels, Integer *nycels,
                             Integer *lcoefs, double **coefs,
                             Integer *gsmoothness);
```

```
static Integer handle_options(Integer gsmoothness,
                              Integer iopts[], const Integer liopts,
                              double opts[], const Integer lopts);
static Integer evaluate_at_vector(const double coefs[], const Integer iopts[],
                                  const double opts[], const double pmin[],
                                  const double pmax[]);
static Integer evaluate_on_mesh(const double coefs[], const Integer iopts[],
                                const double opts[], const double pmin[],
                                const double pmax[]);

int main(void)
{
  /* Scalars */
  Integer exit_status = 0;
  Integer liopts = 100, lopts = 100;
  Integer gsmoothness, i, lcoefs, lsmaxp, lsminp, n, nxcels, nycels;
  /* Arrays */
  double *coefs = 0, *f = 0, *x = 0, *y = 0;
  double opts[100], pmax[2], pmin[2];
  Integer iopts[100];
  /* Nag Types */
  NagError fail;

  INIT_FAIL(fail);

  printf("nag_2d_spline_fit_ts_scat (e02jdc) Example Program Results\n");

  /* Generate the data to fit and set the compulsory algorithmic control
   * parameters.
   */
  exit_status = generate_data(&n, &x, &y, &f, &lsminp, &lsmaxp, &nxcels,
                              &nycels, &lcoefs, &coefs, &gsmoothness);
  if (exit_status != 0)
    goto END;

  /* Initialize the options arrays and set/get some options. */
  exit_status = handle_options(gsmoothness, iopts, liopts, opts, lopts);
  if (exit_status != 0)
    goto END;

  /* Compute the spline coefficients using
   * nag_2d_spline_fit_ts_scat (e02jdc).
   */
  nag_2d_spline_fit_ts_scat(n, x, y, f, lsminp, lsmaxp, nxcels, nycels,
                            lcoefs, coefs, iopts, opts, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_2d_spline_fit_ts_scat (e02jdc).\n%s\n",
           fail.message);
    exit_status = 1;
    goto END;
  }

  /* pmin and pmax form the bounding box of the spline. We must not attempt to
   * evaluate the spline outside this box. Use nag_dmin_val (f16jpc) and
   * nag_dmax_val (f16jnc) to obtain the min and max values.
   */
  nag_dmin_val(n, x, 1, &i, &pmin[0], &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_dmin_val (f16jpc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }
  nag_dmin_val(n, y, 1, &i, &pmin[1], &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_dmin_val (f16jpc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }
  nag_dmax_val(n, x, 1, &i, &pmax[0], &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_dmax_val (f16jnc).\n%s\n", fail.message);
```

```
    exit_status = 1;
    goto END;
  }
  nag_dmax_val(n, y, 1, &i, &pmax[1], &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_dmax_val (f16jnc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  NAG_FREE(x);
  NAG_FREE(y);
  NAG_FREE(f);

  /* Evaluate the approximation at a vector of values. */
  exit_status = evaluate_at_vector(coefs, iopts, opts, pmin, pmax);
  if (exit_status != 0)
    goto END;

  /* Evaluate the approximation on a mesh. */
  exit_status = evaluate_on_mesh(coefs, iopts, opts, pmin, pmax);
  if (exit_status != 0)
    goto END;

END:

  NAG_FREE(coefs);

  return exit_status;
}

static Integer generate_data(Integer *n, double **x, double **y, double **f,
                             Integer *lsminp, Integer *lsmaxp,
                             Integer *nxcels, Integer *nycels,
                             Integer *lcoefs, double **coefs,
                             Integer *gsmoothness)
{
  /* Reads n from a data file and then generates an x and a y vector of n
   * pseudorandom uniformly distributed values on (0,1]. These are passed
   * to the bivariate function of R. Franke to create the data set to fit.
   * The remaining input data for the fitter are set to suitable values for
   * this problem, as discussed by Davydov and Zeilfelder.
   * Reads the global smoothing level from a data file. This value determines
   * the minimum required length of the array of spline coefficients, coefs.
   */

  /* Scalars */
  Integer exit_status = 0;
  Integer lseed = 4, mstate = 21;
  Integer i, lstate, subid;
  /* Arrays */
  Integer seed[4], state[21];
  /* Nag Types */
  NagError fail;

  INIT_FAIL(fail);

  /* Read the size of the data set to be generated and fitted.
   * (Skip the heading in the data file.)
   */
#ifdef _WIN32
  scanf_s("%*[^\n]");
#else
  scanf("%*[^\n]");
#endif
#ifdef _WIN32
  scanf_s("%" NAG_IFMT "%*[^\n] ", n);
#else
  scanf("%" NAG_IFMT "%*[^\n] ", n);
#endif
```

```
  if (!(*x = NAG_ALLOC(*n, double)) ||
      !(*y = NAG_ALLOC(*n, double)) || !(*f = NAG_ALLOC(*n, double)))
  {
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
  }

  /* Initialize the random number generator using
   * nag_rand_init_repeatable (g05kfc) and then generate the data using
   * nag_rand_basic (g05sac).
   */
  subid = 53;
  seed[0] = 32958;
  seed[1] = 39838;
  seed[2] = 881818;
  seed[3] = 45812;
  lstate = mstate;
  nag_rand_init_repeatable(Nag_WichmannHill_I, 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;
  }
  nag_rand_basic(*n, state, *x, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_rand_basic (g05sac).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }
  nag_rand_basic(*n, state, *y, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_rand_basic (g05sac).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  for (i = 0; i < *n; i++)
    (*f)[i] = 0.75 *
           exp(-(pow((9. * (*x)[i] - 2.), 2) + pow((9. * (*y)[i] - 2.), 2)) /
               4.) + 0.75 * exp(-pow((9. * (*x)[i] + 1.),
                                     2) / 49. - (9. * (*y)[i] + 1.) / 10.) +
           0.5 *
           exp(-(pow((9. * (*x)[i] - 7.), 2) + pow((9. * (*y)[i] - 3.), 2)) /
               4.) - 0.2 * exp(-pow((9. * (*x)[i] - 4.),
                                    2) - pow((9. * (*y)[i] - 7.), 2));

  /* Set the grid size for the approximation. */
  *nxcels = 6;
  *nycels = *nxcels;

  /* Read the required level of global smoothing. */
#ifdef _WIN32
  scanf_s("%" NAG_IFMT "%*[^\n] ", gsmoothness);
#else
  scanf("%" NAG_IFMT "%*[^\n] ", gsmoothness);
#endif

  /* Identify the computation. */
  printf("\n"
         "Computing the coefficients of a C^%" NAG_IFMT
         " spline approximation to" " Franke's function\n" "Using a %"
         NAG_IFMT " by %" NAG_IFMT " grid\n", *gsmoothness, *nxcels, *nycels);

  /* Set the local-approximation control parameters. */
  *lsminp = 3;
  *lsmaxp = 100;

  /* Set up the array to hold the computed spline coefficients. */
```

```
  switch (*gsmoothness) {
  case 1:
    *lcoefs = (((*nxcels + 2) * (*nycels + 2) + 1) / 2) * 10 + 1;
    break;
  case 2:
    *lcoefs = 28 * (*nxcels + 2) * (*nycels + 2) * 4 + 1;
    break;
  default:
    *lcoefs = 0;
  }

  if (!(*coefs = NAG_ALLOC(*lcoefs, double)))
  {
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
  }

END:

  return exit_status;
}

static Integer handle_options(Integer gsmoothness,
                              Integer iopts[], const Integer liopts,
                              double opts[], const Integer lopts)
{
  /* Auxiliary routine for initializing the options arrays and
   * for demonstrating how to set and get optional parameters using
   * nag_fit_opt_set (e02zkc) and nag_fit_opt_get (e02zlc) respectively.
   */

  /* Scalars */
  Integer exit_status = 0;
  double rvalue;
  Integer ivalue, lcvalue;
  /* Arrays */
  char cvalue[16 + 1], optstr[80 + 1], supersmooth[9 + 1];
  /* Nag Types */
  Nag_Boolean supersmooth_enum;
  Nag_VariableType optype;
  NagError fail;

  INIT_FAIL(fail);

  nag_fit_opt_set("Initialize = e02jdc", iopts, liopts, opts, lopts, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_fit_opt_set (e02zkc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* Configure the global approximation method. */
#ifdef _WIN32
  sprintf_s(optstr, (unsigned)_countof(optstr),
            "Global Smoothing Level = %1" NAG_IFMT, gsmoothness);
#else
  sprintf(optstr, "Global Smoothing Level = %1" NAG_IFMT, gsmoothness);
#endif
  nag_fit_opt_set(optstr, iopts, liopts, opts, lopts, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_fit_opt_set (e02zkc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* If C^2 smoothing is requested, compute the spline using additional
   * super-smoothness constraints?
   * (The default is 'No'.)
   */
#ifdef _WIN32
```

```
  scanf_s("%9s%*[^\n] ", supersmooth, (unsigned)_countof(supersmooth));
#else
  scanf("%9s%*[^\n] ", supersmooth);
#endif
  supersmooth_enum = (Nag_Boolean) nag_enum_name_to_value(supersmooth);
  if (gsmoothness == 2 && supersmooth_enum == Nag_TRUE) {
    nag_fit_opt_set("Supersmooth C2 = Yes", iopts, liopts, opts, lopts,
                    &fail);
    if (fail.code != NE_NOERROR) {
      printf("Error from nag_fit_opt_set (e02zkc).\n%s\n", fail.message);
      exit_status = 1;
      goto END;
    }
  }

  nag_fit_opt_set("Averaged Spline = Yes", iopts, liopts, opts, lopts, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_fit_opt_set (e02zkc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* Configure the local approximation method.
   * (The default is 'Polynomial'.)
   */
  nag_fit_opt_set("Local Method = Polynomial", iopts, liopts, opts, lopts,
                  &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_fit_opt_set (e02zkc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

#ifdef _WIN32
  sprintf_s(optstr, (unsigned)_countof(optstr),
            "Minimum Singular Value LPA = %16.9e", 1. / 32.);
#else
  sprintf(optstr, "Minimum Singular Value LPA = %16.9e", 1. / 32.);
#endif
  nag_fit_opt_set(optstr, iopts, liopts, opts, lopts, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_fit_opt_set (e02zkc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  switch (gsmoothness) {
  case 1:
#ifdef _WIN32
    sprintf_s(optstr, (unsigned)_countof(optstr),
              "Polynomial Starting Degree = 3");
#else
    sprintf(optstr, "Polynomial Starting Degree = 3");
#endif
    break;
  case 2:
    if (supersmooth_enum == Nag_TRUE) {
      /* We can benefit from starting with local polynomials of greater
       * degree than with regular C^2 smoothing.
       */
      printf("Using super-smoothing\n");
#ifdef _WIN32
      sprintf_s(optstr, (unsigned)_countof(optstr),
                "Polynomial Starting Degree = 6");
#else
      sprintf(optstr, "Polynomial Starting Degree = 6");
#endif
    }
    else
#ifdef _WIN32
      sprintf_s(optstr, (unsigned)_countof(optstr),
```

```
                "Polynomial Starting Degree = 5");
#else
      sprintf(optstr, "Polynomial Starting Degree = 5");
#endif
    break;
  }
  nag_fit_opt_set(optstr, iopts, liopts, opts, lopts, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_fit_opt_set (e02zkc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  /* As an example of how to get the value of an optional parameter,
   * display whether averaging of local approximations is in operation.
   */
  lcvalue = 16 + 1;
  nag_fit_opt_get("Averaged Spline", &ivalue, &rvalue, cvalue, lcvalue,
                  &optype, iopts, opts, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_fit_opt_get (e02zlc).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  if (!strcmp(cvalue, "YES"))
    printf("Using an averaged local approximation\n");

END:
  return exit_status;
}

static Integer evaluate_at_vector(const double coefs[], const Integer iopts[],
                                  const double opts[], const double pmin[],
                                  const double pmax[])
{
  /* Evaluates the approximation at a vector of values using
   * nag_2d_spline_ts_eval (e02jec).
   */

  /* Scalars */
  Integer exit_status = 0;
  Integer i, nevalv;
  /* Arrays */
  double *fevalv = 0, *xevalv = 0, *yevalv = 0;
  /* Nag Types */
  NagError fail;

  INIT_FAIL(fail);

#ifdef _WIN32
  scanf_s("%" NAG_IFMT "%*[^\n] ", &nevalv);
#else
  scanf("%" NAG_IFMT "%*[^\n] ", &nevalv);
#endif

  if (!(xevalv = NAG_ALLOC(nevalv, double)) ||
      !(yevalv = NAG_ALLOC(nevalv, double)) ||
      !(fevalv = NAG_ALLOC(nevalv, double)))
  {
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
  }

#ifdef _WIN32
  for (i = 0; i < nevalv; i++)
    scanf_s("%lf%lf%*[^\n]", &xevalv[i], &yevalv[i]);
#else
  for (i = 0; i < nevalv; i++)
    scanf("%lf%lf%*[^\n]", &xevalv[i], &yevalv[i]);
```

```
#endif

  /* Force the points to be within the bounding box of the spline. */
  for (i = 0; i < nevalv; i++) {
    xevalv[i] = MAX(xevalv[i], pmin[0]);
    xevalv[i] = MIN(xevalv[i], pmax[0]);
    yevalv[i] = MAX(yevalv[i], pmin[1]);
    yevalv[i] = MIN(yevalv[i], pmax[1]);
  }

  /* Evaluate using nag_2d_spline_ts_eval (e02jec). */
  nag_2d_spline_ts_eval(nevalv, xevalv, yevalv, coefs, fevalv, iopts, opts,
                        &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_2d_spline_ts_eval (e02jec).\n%s\n", fail.message);
    exit_status = 1;
    goto END;
  }

  printf("\nValues of computed spline at (x_i,y_i):\n"
         "\n%12s%12s%12s\n", "x_i", "y_i", "f(x_i,y_i)");

  for (i = 0; i < nevalv; i++)
    printf("%12.2f%12.2f%12.2f\n", xevalv[i], yevalv[i], fevalv[i]);

END:

  NAG_FREE(xevalv);
  NAG_FREE(yevalv);
  NAG_FREE(fevalv);

  return exit_status;
}

static Integer evaluate_on_mesh(const double coefs[], const Integer iopts[],
                                const double opts[], const double pmin[],
                                const double pmax[])
{
  /* Evaluates the approximation on a mesh of n_x * n_y values. */

  /* Scalars */
  Integer exit_status = 0;
  Integer i, j, nxeval, nyeval;
  /* Arrays */
  char print_mesh[9 + 1];
  double *fevalm = 0, *xevalm = 0, *yevalm = 0;
  double h[2], ll_corner[2], ur_corner[2];
  /* Nag Types */
  Nag_Boolean print_mesh_enum;
  NagError fail;

  INIT_FAIL(fail);

#ifdef _WIN32
  scanf_s("%" NAG_IFMT "%" NAG_IFMT "%*[^\n] ", &nxeval, &nyeval);
#else
  scanf("%" NAG_IFMT "%" NAG_IFMT "%*[^\n] ", &nxeval, &nyeval);
#endif

  if (!(xevalm = NAG_ALLOC(nxeval, double)) ||
      !(yevalm = NAG_ALLOC(nyeval, double)) ||
      !(fevalm = NAG_ALLOC(nxeval * nyeval, double)))
  {
    printf("Allocation failure\n");
    exit_status = -1;
    goto END;
  }

  /* Define the mesh by its lower-left and upper-right corners, which must
   * lie within the bounding box of the spline.
   */
```

```
#ifdef _WIN32
  scanf_s("%lf%lf%*[^\n] ", &ll_corner[0], &ll_corner[1]);
#else
  scanf("%lf%lf%*[^\n] ", &ll_corner[0], &ll_corner[1]);
#endif
#ifdef _WIN32
  scanf_s("%lf%lf%*[^\n] ", &ur_corner[0], &ur_corner[1]);
#else
  scanf("%lf%lf%*[^\n] ", &ur_corner[0], &ur_corner[1]);
#endif

  for (i = 0; i < 2; i++) {
    ll_corner[i] = MAX(ll_corner[i], pmin[i]);
    ur_corner[i] = MIN(ur_corner[i], pmax[i]);
  }

  /* Set the mesh spacing and the evaluation points. */
  h[0] = (ur_corner[0] - ll_corner[0]) / (double) (nxeval - 1);
  h[1] = (ur_corner[1] - ll_corner[1]) / (double) (nyeval - 1);

  for (i = 0; i < nxeval; i++)
    xevalm[i] = ll_corner[0] + (double) i *h[0];
  for (j = 0; j < nyeval; j++)
    yevalm[j] = ll_corner[1] + (double) j *h[1];

  /* Evaluate using nag_2d_spline_ts_eval_rect (e02jfc). */
  nag_2d_spline_ts_eval_rect(nxeval, nyeval, xevalm, yevalm, coefs, fevalm,
                             iopts, opts, &fail);
  if (fail.code != NE_NOERROR) {
    printf("Error from nag_2d_spline_ts_eval_rect (e02jfc).\n%s\n",
           fail.message);
    exit_status = 1;
    goto END;
  }

  /* Output the computed function values? */
#ifdef _WIN32
  scanf_s("%9s%*[^\n] ", print_mesh, (unsigned)_countof(print_mesh));
#else
  scanf("%9s%*[^\n] ", print_mesh);
#endif
  print_mesh_enum = (Nag_Boolean) nag_enum_name_to_value(print_mesh);

  printf("\n");
  if (!print_mesh_enum) {
    printf("Outputting of the function values on the mesh is disabled\n");
  }
  else {
    printf("Values of computed spline at (x_i,y_j):\n"
           "\n%12s%12s%12s\n", "x_i", "y_j", "f(x_i,y_j)");
    for (j = 0; j < nyeval; j++)
      for (i = 0; i < nxeval; i++)
        printf("%12.2f%12.2f%12.2f\n", xevalm[i], yevalm[j],
               fevalm[j * nxeval + i]);
  }

END:

  NAG_FREE(xevalm);
  NAG_FREE(yevalm);
  NAG_FREE(fevalm);

  return exit_status;
}
```

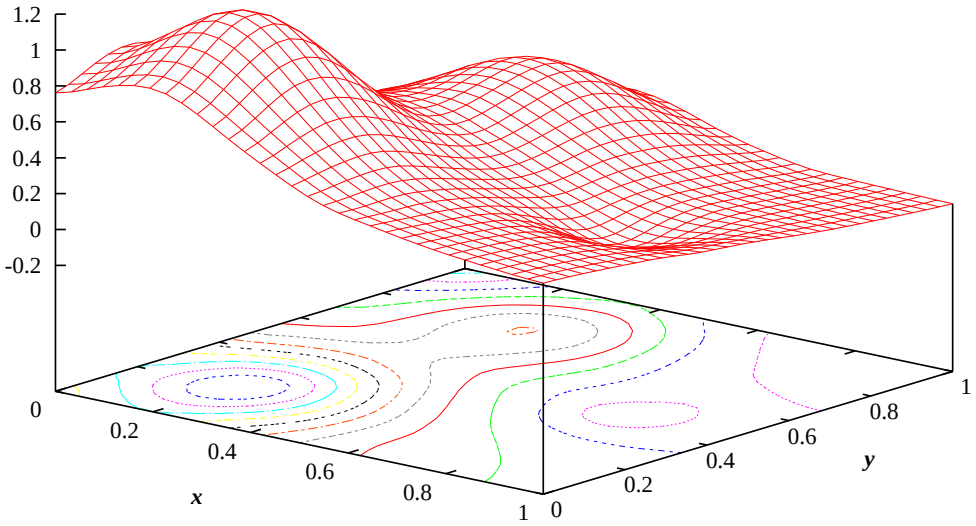
10.2 Program Data

```
nag_2d_spline_fit_ts_scat (e02jdc) Example Program Data
 100                                          : number of data points to fit
  1                                           : global smoothing level
  Nag_FALSE                                   : if C^2 smoothing, supersmooth?
  1                                           : no. points for vector evaluation
  0  0                                        : (x_i,y_i) vector to eval.
  101  101                                    : (n_x,n_y) size for mesh eval.
  0  0                                        : mesh lower-left corner
  1  1                                        : mesh upper-right corner
  Nag_FALSE                                   : display the computed mesh vals?
```

10.3 Program Results

```
nag_2d_spline_fit_ts_scat (e02jdc) Example Program Results

Computing the coefficients of a C^1 spline approximation to Franke's function
Using a 6 by 6 grid
Using an averaged local approximation

Values of computed spline at (x_i,y_i):

         x_i          y_i  f(x_i,y_i)
        0.01         0.01        0.73

Outputting of the function values on the mesh is disabled
```

Example Program
Calculation and Evaluation of Bivariate Spline Fit
from Scattered Data using Two-Stage Approximation



11 Optional Parameters

Several optional parameters in nag_2d_spline_fit_ts_scat (e02jdc) control aspects of the algorithm, methodology used, logic or output. Their values are contained in the arrays **iopts** and **opts**; these must be initialized before calling nag_2d_spline_fit_ts_scat (e02jdc) by first calling nag_fit_opt_set (e02zkc) with **optstr** set to `"Initialize = e02jdc"`.

Each optional parameter has an associated default value; to set any of them to a non-default value, or to reset any of them to the default value, use nag_fit_opt_set (e02zkc). The current value of an optional parameter can be queried using nag_fit_opt_get (e02zlc).

The remainder of this section can be skipped if you wish to use the default values for all optional parameters.

The following is a list of the optional parameters available. A full description of each optional parameter is provided in Section 11.1.

Averaged Spline
Global Smoothing Level
Interpolation Only RBF
Local Method
Minimum Singular Value LHA
Minimum Singular Value LPA
Polynomial Starting Degree
Radial Basis Function
Scaling Coefficient RBF
Separation LRBFA
Supersmooth C2

11.1 Description of the Optional Parameters

For each option, we give a summary line, a description of the optional parameter and details of constraints.

The summary line contains:

the keywords;

a parameter value, where the letters a, i and r denote options that take character, integer and real values respectively;

the default value.

Keywords and character values are case insensitive.

For nag_2d_spline_fit_ts_scat (e02jdc) the maximum length of the parameter **cvalue** used by nag_fit_opt_get (e02zlc) is 16.

Averaged Spline a Default $=$ NO

When the bounding box is triangulated there are 8 equivalent configurations of the mesh. Setting **Averaged Spline** $=$ YES will use the averaged value of the 8 possible local polynomial approximations over each triangle in the mesh. This usually gives better results but at (about 8 times) higher computational cost.

Constraint: **Averaged Spline** $=$ YES or NO.

Global Smoothing Level i Default $= 1$

The smoothness level for the global spline approximation.

Global Smoothing Level $= 1$
Will use C^1 piecewise cubics.

Global Smoothing Level $= 2$
Will use C^2 piecewise sextics.

Constraint: **Global Smoothing Level** $= 1$ or 2.

Interpolation Only RBF a Default = YES

If **Interpolation Only RBF** = YES, each local RBF approximation is computed by interpolation.

If **Interpolation Only RBF** = NO, each local RBF approximation is computed by a discrete least squares approach. This is likely to be more accurate and more expensive than interpolation.

If **Local Method** = HYBRID or POLYNOMIAL, this option setting is ignored.

Constraint: **Interpolation Only RBF** = YES or NO.

Local Method a Default = POLYNOMIAL

The local approximation scheme to use.

Local Method = POLYNOMIAL
Uses least squares polynomial approximations.

Local Method = HYBRID
Uses hybrid polynomial and RBF approximations.

Local Method = RBF
Uses pure RBF approximations.

In general POLYNOMIAL is less computationally expensive than HYBRID is less computationally expensive than RBF with the reverse ordering holding for accuracy of results.

Constraint: **Local Method** = POLYNOMIAL, HYBRID or RBF.

Minimum Singular Value LHA r Default = 1.0

A tolerance measure for accepting or rejecting a local hybrid approximation (LHA) as reliable.

The solution of a local least squares problem solved on each triangle subdomain is accepted as reliable if the minimum singular value σ of the collocation matrix (of polynomial and radial basis function terms) associated with the least squares problem satisfies **Minimum Singular Value LHA** $\leq \sigma$.

In general the approximation power will be reduced as **Minimum Singular Value LHA** is reduced. (A small σ indicates that the local data has hidden redundancies which prevent it from carrying enough information for a good approximation to be made.) Setting **Minimum Singular Value LHA** very large may have the detrimental effect that only approximations of low degree are deemed reliable.

A calibration procedure (experimenting with a small subset of the data to be fitted and validating the results) may be needed to choose the most appropriate value for this parameter.

If **Local Method** = POLYNOMIAL or RBF, this option setting is ignored.

Constraint: **Minimum Singular Value LHA** ≥ 0.0.

Minimum Singular Value LPA r Default = 1.0

A tolerance measure for accepting or rejecting a local polynomial approximation (LPA) as reliable. Clearly this setting is relevant when **Local Method** = POLYNOMIAL, but it also may be used when **Local Method** = HYBRID (see Section 9.)

The solution of a local least squares problem solved on each triangle subdomain is accepted as reliable if the minimum singular value σ of the matrix (of Bernstein polynomial values) associated with the least squares problem satisfies **Minimum Singular Value LPA** $\leq \sigma$.

In general the approximation power will be reduced as **Minimum Singular Value LPA** is reduced. (A small σ indicates that the local data has hidden redundancies which prevent it from carrying enough information for a good approximation to be made.) Setting **Minimum Singular Value LPA** very large may have the detrimental effect that only approximations of low degree are deemed reliable.

Minimum Singular Value LPA will have no effect if **Polynomial Starting Degree** = 0, and it will have little effect if the input data is 'smooth' (e.g., from a known function).

A calibration procedure (experimenting with a small subset of the data to be fitted and validating the results) may be needed to choose the most appropriate value for this parameter.

If **Local Method** = RBF, this option setting is ignored.

Constraint: **Minimum Singular Value LPA** ≥ 0.0.

Polynomial Starting Degree i Default $= 5$ if **Local Method** = HYBRID, Default $= 1$ otherwise

The degree to be used for the polynomial part in the initial step of each local approximation.

At this initial step the method will attempt to fit with a local approximation having polynomial part of degree **Polynomial Starting Degree**. If **Local Method** = POLYNOMIAL and the approximation is deemed unreliable (according to **Minimum Singular Value LPA**), the degree will be decremented by one and a new local approximation computed, ending with a constant approximation if no other is reliable. If **Local Method** = HYBRID and the approximation is deemed unreliable (according to **Minimum Singular Value LHA**), a pure polynomial approximation of this degree will be tried instead. The method then proceeds as in the POLYNOMIAL case.

Polynomial Starting Degree is bounded from above by the maximum possible spline degree, which is 6 (when performing C^2 global super-smoothing). Note that the best-case approximation error (see Section 7) for C^2 smoothing with **Supersmooth C2** = NO is achieved for local polynomials of degree 5; that is, for this level of global smoothing no further benefit is gained by setting **Polynomial Starting Degree** = 6.

The default value gives a good compromise between efficiency and accuracy. In general the best approximation can be obtained by setting:

If **Local Method** = POLYNOMIAL

if **Global Smoothing Level** = 1, **Polynomial Starting Degree** = 3;

if **Global Smoothing Level** = 2;

if **Supersmooth C2** = NO, **Polynomial Starting Degree** = 5;

otherwise **Polynomial Starting Degree** = 6.

If **Local Method** = HYBRID, **Polynomial Starting Degree** as small as possible.

If **Local Method** = RBF, this option setting is ignored.

Constraints:

if **Local Method** = HYBRID,

if **Radial Basis Function** = MQ2, MQ3, TPS or POLYHARMONIC3,
Polynomial Starting Degree ≥ 1;
if **Radial Basis Function** = TPS4 or POLYHARMONIC5,
Polynomial Starting Degree ≥ 2;
if **Radial Basis Function** = TPS6 or POLYHARMONIC7,
Polynomial Starting Degree ≥ 3;
if **Radial Basis Function** = POLYHARMONIC9,
Polynomial Starting Degree ≥ 4.;
otherwise **Polynomial Starting Degree** ≥ 0;
if **Local Method** = POLYNOMIAL and **Global Smoothing Level** = 1,
Polynomial Starting Degree ≤ 3;
otherwise **Polynomial Starting Degree** ≤ 6.

Radial Basis Function a Default $=$ MQ
Scaling Coefficient RBF r Default $= 1.0$

Radial Basis Function selects the RBF to use in each local RBF approximation, while **Scaling Coefficient RBF** selects the scale factor to use in its evaluation, as described below.

A calibration procedure (experimenting with a small subset of the data to be fitted and validating the results) may be needed to choose the most appropriate scale factor and RBF.

If **Local Method** = POLYNOMIAL, these option settings are ignored.

If **Local Method** = HYBRID or RBF, the following (conditionally) positive definite functions may be chosen.

Define $R = \sqrt{x^2 + y^2}$ and $\rho = R/r$.

GAUSS	Gaussian $\exp(-\rho^2)$
IMQ	inverse multiquadric $1/\sqrt{r^2 + R^2}$
IMQ2	inverse multiquadric $1/(r^2 + R^2)$
IMQ3	inverse multiquadric $1/(r^2 + R^2)^{(3/2)}$
IMQ0_5	inverse multiquadric $1/(r^2 + R^2)^{(1/4)}$
WENDLAND31	H. Wendland's C^2 function $\max(0, 1-\rho)^4(4\rho + 1)$
WENDLAND32	H. Wendland's C^4 function $\max(0, 1-\rho)^6(35\rho^2 + 18\rho + 3)$
WENDLAND33	H. Wendland's C^6 function $\max(0, 1-\rho)^8(32\rho^3 + 25\rho^2 + 8\rho + 1)$
BUHMANNC3	M. Buhmann's C^3 function $112/45\rho^{(9/2)} + 16/3\rho^{(7/2)} - 7\rho^4 - 14/15\rho^2 + 1/9$ if $\rho \leq 1$, 0 otherwise
MQ	multiquadric $\sqrt{r^2 + R^2}$
MQ1_5	multiquadric $(r^2 + R^2)^{(1.5/2)}$
POLYHARMONIC1_5	polyharmonic spline $\rho^{1.5}$
POLYHARMONIC1_75	polyharmonic spline $\rho^{1.75}$

If **Local Method** = HYBRID the following conditionally positive definite functions may also be chosen.

MQ2	multiquadric $(r^2 + R^2)\log(r^2 + R^2)$
MQ3	multiquadric $(r^2 + R^2)^{(3/2)}$
TPS	thin plate spline $\rho^2 \log \rho^2$
POLYHARMONIC3	polyharmonic spline ρ^3
TPS4	thin plate spline $\rho^4 \log \rho^2$
POLYHARMONIC5	polyharmonic spline ρ^5
TPS6	thin plate spline $\rho^6 \log \rho^2$
POLYHARMONIC7	polyharmonic spline ρ^7
POLYHARMONIC9	polyharmonic spline ρ^9

Constraints:

if **Radial Basis Function** = MQ2, MQ3, TPS or POLYHARMONIC3,
Local Method = HYBRID and **Polynomial Starting Degree** ≥ 1;
if **Radial Basis Function** = TPS4 or POLYHARMONIC5,
Local Method = HYBRID and **Polynomial Starting Degree** ≥ 2;
if **Radial Basis Function** = TPS6 or POLYHARMONIC7,
Local Method = HYBRID and **Polynomial Starting Degree** ≥ 3;
if **Radial Basis Function** = POLYHARMONIC9,
Local Method = HYBRID and **Polynomial Starting Degree** ≥ 4;
Scaling Coefficient RBF > 0.0.

Separation LRBFA r Default $= 16.0/$**Scaling Coefficient RBF**

A knot-separation parameter used to control the condition number of the matrix used in each local RBF approximation (LRBFA). A smaller value may mean greater numerical stability but fewer knots.

If **Local Method** = HYBRID or POLYNOMIAL, this option setting is ignored.

Constraint: **Separation LRBFA** > 0.0.

Supersmooth C2 a Default = NO

If **Supersmooth C2** = YES, the C^2 spline is generated using additional smoothness constraints. This usually gives better results but at higher computational cost.

If **Global Smoothing Level** = 1 this option setting is ignored.

Constraint: **Supersmooth C2** = YES or NO.