NAG Library Function Document

nag_fit_1dspline_deriv_vector (e02bfc)

1 Purpose

nag_fit_1dspline_deriv_vector (e02bfc) evaluates a cubic spline and up to its first three derivatives from its B-spline representation at a vector of points. nag_fit_1dspline_deriv_vector (e02bfc) can be used to compute the values and derivatives of cubic spline fits and interpolants produced by reference to nag_1d_spline_interpolant (e01bac), nag_1d_spline_fit_knots (e02bac) and nag_1d_spline_fit (e02bec).

2 Specification

```
#include <nag.h>
#include <nage02.h>
```

```
void nag_fit_1dspline_deriv_vector (Nag_SplineVectorSort start,
     Nag_Spline *spline, Nag_DerivType deriv, Nag_Boolean xord,
     const double x[], Integer ixloc[], Integer nx, double s[], Integer pds,
     Integer iwrk[], Integer liwrk, NagError *fail)
```

3 Description

nag_fit_1dspline_deriv_vector (e02bfc) evaluates the cubic spline $s(x)$ and optionally derivatives up to order 3 for a vector of points x_j, for $j = 1, 2, \dots, n_x$. It is assumed that $s(x)$ is represented in terms of its B-spline coefficients c_i, for $i = 1, 2, \dots, \bar{n} + 3$, and (augmented) ordered knot set λ_i, for $i = 1, 2, \dots, \bar{n} + 7$, (see nag_1d_spline_fit_knots (e02bac) and nag_1d_spline_fit (e02bec)), i.e.,

$$s(x) = \sum_{i=1}^{q} c_i N_i(x).$$

Here $q = \bar{n} + 3$, $\bar{n}$ is the number of intervals of the spline and $N_i(x)$ denotes the normalized B-spline of degree 3 (order 4) defined upon the knots $\lambda_i, \lambda_{i+1}, \dots, \lambda_{i+4}$. The knots $\lambda_5, \lambda_6, \dots, \lambda_{\bar{n}+3}$ are the interior knots. The remaining knots, λ_1, λ_2, λ_3, λ_4 and $\lambda_{\bar{n}+4}$, $\lambda_{\bar{n}+5}$, $\lambda_{\bar{n}+6}$, $\lambda_{n\bar{+}7}$ are the exterior knots. The knots λ_4 and $\lambda_{\bar{n}+4}$ are the boundaries of the spline.

Only abscissae satisfying,

$$\lambda_4 \leq x_j \leq \lambda_{\bar{n}+4},$$

will be evaluated. At a simple knot λ_i (i.e., one satisfying $\lambda_{i-1} < \lambda_i < \lambda_{i+1}$), the third derivative of the spline is, in general, discontinuous. At a multiple knot (i.e., two or more knots with the same value), lower derivatives, and even the spline itself, may be discontinuous. Specifically, at a point $x = u$ where (exactly) r knots coincide (such a point is termed a knot of multiplicity r), the values of the derivatives of order $4 - j$, for $j = 1, 2, \dots, r$, are, in general, discontinuous. (Here $1 \leq r \leq 4$; $r > 4$ is not meaningful.) The maximum order of the derivatives to be evaluated D_{ord}, and the left- or right-handedness of the computation when an abscissa corresponds exactly to an interior knot, are determined by the value of **deriv**.

Each abscissa (point at which the spline is to be evaluated) x_j contained in **x** has an associated enclosing interval number, $ixloc_j$ either supplied or returned in **ixloc** (see argument **start**). A simple call to nag_fit_1dspline_deriv_vector (e02bfc) would set **start** = Nag_SplineVectorSort_Sorted and the contents of **ixloc** need never be set nor referenced, and the following description on modes of operation can be ignored. However, where efficiency is an important consideration, the following description will help to choose the appropriate mode of operation.

The interval numbers are used to determine which B-splines must be evaluated for a given abscissa, and are defined as

$$ixloc_j = \begin{pmatrix} \leq 0 & x_j < \lambda_1 & \\ 4 & \lambda_4 = x_j & \\ k & \lambda_k < x_j < \lambda_{k+1} & \\ k & \lambda_4 < \lambda_k = x_j & \text{left derivatives} \\ k & x_j = \lambda_{k+1} < \lambda_{\bar{n}+4} & \text{right derivatives or no derivatives} \\ \bar{n}+4 & \lambda_{\bar{n}+4} = x_j & \\ > \bar{n}+7 & x_j > \lambda_{\bar{n}+7} & \end{pmatrix} \qquad (1)$$

The algorithm has two modes of vectorization, termed here sorted and unsorted, which are selectable by the argument **start**.

Furthermore, if the supplied abscissae are sufficiently ordered, as indicated by the argument **xord**, the algorithm will take advantage of significantly faster methods for the determination of both the interval numbers and the subsequent spline evaluations.

The sorted mode has two phases, a sorting phase and an evaluation phase. This mode is recommended if there are many abscissae to evaluate relative to the number of intervals of the spline, or the abscissae are distributed relatively densely over a subsection of the spline. In the first phase, $ixloc_j$ is determined for each x_j and a permutation is calculated to sort the x_j by interval number. The first phase may be either partially or completely by-passed using the argument **start** if the enclosing segments and/or the subsequent ordering are already known *a priori*, for example if multiple spline coefficients **spline→c** are to be evaluated over the same set of knots **spline→lamda**.

In the second phase of the sorted mode, spline approximations are evaluated by segment, so that non-abscissa dependent calculations over a segment may be reused in the evaluation for all abscissae belonging to a specific segment. For example, all third derivatives of all abscissae in the same segment will be identical.

In the unsorted mode of vectorization, no *a priori* segment sorting is performed, and if the abscissae are not sufficiently ordered, the evaluation at an abscissa will be independent of evaluations at other abscissae; also non-abscissa dependent calculations over a segment will be repeated for each abscissa in a segment. This may be quicker if the number of abscissa is small in comparison to the number of knots in the spline, and they are distributed sparsely throughout the domain of the spline. This is effectively a direct vectorization of nag_1d_spline_evaluate (e02bbc) and nag_1d_spline_deriv (e02bcc), although if the enclosing interval numbers $ixloc_j$ are known, these may again be provided.

If the abscissae are sufficiently ordered, then once the first abscissa in a segment is known, an efficient algorithm will be used to determine the location of the final abscissa in this segment. The spline will subsequently be evaluated in a vectorized manner for all the abscissae indexed between the first and last of the current segment.

If no derivatives are required, the spline evaluation is calculated by taking convex combinations due to de Boor (1972). Otherwise, the calculation of $s(x)$ and its derivatives is based upon,

(i) evaluating the nonzero B-splines of orders 1, 2, 3 and 4 by recurrence (see Cox (1972) and Cox (1978)),

(ii) computing all derivatives of the B-splines of order 4 by applying a second recurrence to these computed B-spline values (see de Boor (1972)),

(iii) multiplying the fourth-order B-spline values and their derivative by the appropriate B-spline coefficients, and summing, to yield the values of $s(x)$ and its derivatives.

The method of convex combinations is significantly faster than the recurrence based method. If higher derivatives of order 2 or 3 are not required, as much computation as possible is avoided.

4 References

Cox M G (1972) The numerical evaluation of B-splines *J. Inst. Math. Appl.* **10** 134–149

Cox M G (1978) The numerical evaluation of a spline from its B-spline representation *J. Inst. Math. Appl.* **21** 135–143

de Boor C (1972) On calculating with B-splines *J. Approx. Theory* **6** 50–62

5 Arguments

1: **start** – Nag_SplineVectorSort *Input*

On entry: indicates the completion state of the first phase of the algorithm.

start = Nag_SplineVectorSort_Sorted
The enclosing interval numbers $ixloc_j$ for the abscissae x_j contained in **x** have not been determined, and you wish to use the sorted mode of vectorization.

start = Nag_SplineVectorSort_Sorted_Indexed
The enclosing interval numbers $ixloc_j$ have been determined and are provided in **ixloc**, however the required permutation and interval related information has not been determined and you wish to use the sorted mode of vectorization.

start = Nag_SplineVectorSort_Sorted_Indexed_Perm
You wish to use the sorted mode of vectorization, and the entire first phase has been completed, with the enclosing interval numbers supplied in **ixloc**, and the required permutation and interval related information provided in **iwrk** (from a previous call to nag_fit_1dspline_deriv_vector (e02bfc)).

start = Nag_SplineVectorSort_Unsorted
The enclosing interval numbers $ixloc_j$ for the abscissae x_j contained in **x** have not been determined, and you wish to use the unsorted mode of vectorization.

start = Nag_SplineVectorSort_Unsorted_Indexed
The enclosing interval numbers $ixloc_j$ for the abscissae x_j contained in **x** have been supplied in **ixloc**, and you wish to use the unsorted mode of vectorization.

Constraint: **start** = Nag_SplineVectorSort_Sorted, Nag_SplineVectorSort_Sorted_Indexed, Nag_SplineVectorSort_Sorted_Indexed_Perm, Nag_SplineVectorSort_Unsorted or Nag_SplineVectorSort_Unsorted_Indexed.

Additional: **start** = Nag_SplineVectorSort_Sorted or Nag_SplineVectorSort_Unsorted should be used unless you are sure that the knot set is unchanged between calls.

2: **spline** – Nag_Spline *

Pointer to structure of type Nag_Spline with the following members:

n – Integer *Input*

On entry: $\bar{n}+7$, where $\bar{n}$ is the number of intervals of the spline (which is one greater than the number of interior knots, i.e., the knots strictly within the range λ_4 to $\lambda_{\bar{n}+4}$ over which the spline is defined).

Constraint: **spline→n** ≥ 8.

lamda – double * *Input*

On entry: a pointer to which memory of size **spline→n** must be allocated. **spline→lamda**$[k-1]$ must be set to the value of the kth member of the complete set of knots, λ_k, for $k = 1, 2, \ldots, \bar{n}+7$.

Constraint: the λ_k must be in nondecreasing order with **spline→lamda**[**spline→n** $-$ 4] $>$ **spline→lamda**[3].

c – double * *Input*

On entry: a pointer to which memory of size **spline→n** $-$ 4 must be allocated. **spline→c** holds the coefficient c_i of the B-spline $N_i(x)$, for $i = 1, 2, \ldots, \bar{n}+3$.

Under normal usage, the call to function nag_fit_1dspline_deriv_vector (e02bfc) will follow at least one call to nag_1d_spline_interpolant (e01bac), nag_1d_spline_fit_knots (e02bac) or nag_1d_spline_fit (e02bec)). In that case, the structure spline will have been set up correctly for input to nag_fit_1dspline_deriv_vector (e02bfc). If multiple sets of B-spline co-efficients are required for the same set of knots λ and the same set of abscissae x, multiple calls to nag_fit_1dspline_deriv_vector (e02bfc) may be made with **spline**→**c** pointing to different coefficient sets, with **start** set appropriately for efficiency.

3: **deriv** – Nag_DerivType *Input*

On entry: determines the maximum order of derivatives required, D_{ord}, as well as the computational behaviour when absicssae correspond exactly to interior knots.

For abscissae satisfying $x_j = \lambda_4$ or $x_j = \lambda_{\bar{n}+4}$ only right-handed or left-handed computation will be used respectively. For abscissae which do not coincide exactly with a knot, the handedness of the computation is immaterial.

deriv = Nag_NoDerivs
No derivatives required. $D_{\text{ord}} = 0$. Only right-handed computation will be used at interior knots.

deriv = Nag_LeftDerivs_1 or Nag_RightDerivs_1
Only $s(x)$ and its first derivative are required. $D_{\text{ord}} = 1$.

deriv = Nag_LeftDerivs_2 or Nag_RightDerivs_2
Only $s(x)$ and its first and second derivatives are required. $D_{\text{ord}} = 2$.

deriv = Nag_LeftDerivs_3 or Nag_RightDerivs_3
$s(x)$ and its first, second and third derivatives are required. $D_{\text{ord}} = 3$.

Constraint: **deriv** = Nag_NoDerivs, Nag_LeftDerivs_1, Nag_RightDerivs_1, Nag_LeftDerivs_2, Nag_RightDerivs_2, Nag_LeftDerivs_3 or Nag_RightDerivs_3.

Additional: if left-handed computation of the spline s is required, a value of **deriv** must be chosen which computes at least the first derivative in a left-handed manner. As mentioned in Section 3, the handedness of the computation of s will only have an effect if at least 4 interior knots are identical.

4: **xord** – Nag_Boolean *Input*

On entry: indicates whether **x** is supplied in a sufficiently ordered manner. If **x** is sufficiently ordered nag_fit_1dspline_deriv_vector (e02bfc) will complete faster.

xord = Nag_TRUE
The abscissae in **x** are ordered at least by ascending interval, in that any two abscissae contained in the same interval are only separated by abscissae in the same interval. For example, $x_j < x_{j+1}$, for $j = 1, 2, \ldots, \mathbf{nx} - 1$.

xord = Nag_FALSE
The abscissae in **x** are not sufficiently ordered.

5: **x**[**nx**] – const double *Input*

On entry: the abscissae x_j, for $j = 1, 2, \ldots, n_x$. If **start** = Nag_SplineVectorSort_Sorted or Nag_SplineVectorSort_Unsorted then evaluations will only be performed for these x_j satisfying $\lambda_4 \leq x_j \leq \lambda_{\bar{n}+4}$. Otherwise evaluation will be performed unless the corresponding element of **ixloc** contains an invalid interval number. Please note that if the **ixloc**[j] is a valid interval number then no check is made that **x**[j] actually lies in that interval.

Constraint: at least one abscissa must fall between **spline**→**lamda**[3] and **spline**→**lamda**[**spline**→**n** − 4].

6: **ixloc**[**nx**] – Integer *Input/Output*

On entry: if **start** = Nag_SplineVectorSort_Sorted_Indexed, Nag_SplineVectorSort_Sorted_Indexed_Perm or Nag_SplineVectorSort_Unsorted_Indexed, if you wish x_j to be evaluated, **ixloc**[$j - 1$] must be the enclosing interval number $ixloc_j$ of the abscissae x_j (see (1)). If you do not wish x_j to be evaluated, you may set the interval number to be either less than 4 or greater than $\bar{n} + 4$.

Otherwise, **ixloc** need not be set.

On exit: if **start** = Nag_SplineVectorSort_Sorted_Indexed, Nag_SplineVectorSort_Sorted_Indexed_Perm or Nag_SplineVectorSort_Unsorted_Indexed, **ixloc** is unchanged on exit.

Otherwise, **ixloc**$[j-1]$, contains the enclosing interval number $ixloc_j$, for the abscissa supplied in **x**$[j-1]$, for $j=1,2,\ldots,n_x$. Evaluations will only be performed for abscissae x_j satisfying $\lambda_4 \leq x_j \leq \lambda_{\bar{n}+4}$. If evaluation is not performed **ixloc**$[j-1]$ is set to 0 if $x_j < \lambda_4$ or $\bar{n}+7$ if $x_j > \lambda_{\bar{n}+4}$.

Constraint: if **start** = Nag_SplineVectorSort_Sorted_Indexed, Nag_SplineVectorSort_Sorted_Indexed_Perm or Nag_SplineVectorSort_Unsorted_Indexed, at least one element of **ixloc** must be between 4 and **spline**→**n** − 3.

7: **nx** – Integer *Input*

On entry: n_x, the total number of abscissae contained in **x**, including any that will not be evaluated.

Constraint: **nx** ≥ 1.

8: **s**$[dim]$ – double *Output*

Note: the dimension, *dim*, of the array **s** must be at least **pds** $\times (D_{\mathrm{ord}}+1)$, see **deriv** for the definition of D_{ord}.

On exit: if x_j is valid, $\mathbf{S}(j,d)$ will contain the $(d-1)$th derivative of $s(x)$, for $d=1,2,\ldots,D_{\mathrm{ord}}+1$ and $j=1,2,\ldots,n_x$. In particular, $\mathbf{S}(j,1)$ will contain the approximation of $s(x_j)$ for all legal values in **x**.

9: **pds** – Integer *Input*

On entry: the stride separating row elements in the two-dimensional data stored in the array **s**.

Constraint: **pds** $\geq$ **nx**, regardless of the acceptability of the elements of **x**.

10: **iwrk**[**liwrk**] – Integer *Input/Output*

On entry: if **start** = Nag_SplineVectorSort_Sorted_Indexed_Perm, **iwrk** must be unchanged from a previous call to nag_fit_1dspline_deriv_vector (e02bfc) with **start** = Nag_SplineVectorSort_Sorted or Nag_SplineVectorSort_Sorted_Indexed.

Otherwise, **iwrk** need not be set. Furthermore, **iwrk** may be **NULL** if **start** = Nag_SplineVectorSort_Unsorted or Nag_SplineVectorSort_Unsorted_Indexed.

On exit: if **start** = Nag_SplineVectorSort_Unsorted or Nag_SplineVectorSort_Unsorted_Indexed, **iwrk** is unchanged on exit.

Otherwise, **iwrk** contains the required permutation of elements of **x**, if any, and information related to the division of the abscissae x_j between the intervals derived from **spline**→**lamda**.

11: **liwrk** – Integer *Input*

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

Constraint: if **start** = Nag_SplineVectorSort_Sorted, Nag_SplineVectorSort_Sorted_Indexed or Nag_SplineVectorSort_Sorted_Indexed_Perm, **liwrk** $\geq 3 + 3 \times$ **nx**.

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

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

6 Error Indicators and Warnings

NE_ABSCI_OUTSIDE_KNOT_INTVL

On entry, all elements of **x** had enclosing interval numbers in **ixloc** outside the domain allowed by the provided spline.
$\langle value \rangle$ entries of **x** were indexed below the lower bound $\langle value \rangle$.
$\langle value \rangle$ entries of **x** were indexed above the upper bound $\langle value \rangle$.

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_INT

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

On entry, **spline→n** $= \langle value \rangle$.
Constraint: **spline→n** ≥ 8.

NE_INT_2

On entry, **liwrk** $= \langle value \rangle$.
Constraint: **liwrk** $\geq 3 \times$ **nx** $+ 3 = \langle value \rangle$.

On entry, **pds** $= \langle value \rangle$.
Constraint: **pds** $\geq$ **nx** $= \langle value \rangle$.

NE_INT_CHANGED

On entry, **start** = Nag_SplineVectorSort_Sorted_Indexed_Perm and **nx** is not consistent with the previous call to nag_fit_1dspline_deriv_vector (e02bfc).
On entry, **nx** $= \langle value \rangle$.
Constraint: **nx** $= \langle value \rangle$.

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_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_SPLINE_RANGE_INVALID

On entry, **spline→lamda**[3] $= \langle value \rangle$, **spline→n** $= \langle value \rangle$ and
spline→lamda[**spline→n** $- 4$] $= \langle value \rangle$.
Constraint: **spline→lamda**[3] $<$ **spline→lamda**[**spline→n** $- 4$].

NW_SOME_SOLUTIONS

On entry, at least one element of **x** has an enclosing interval number in **ixloc** outside the set allowed by the provided spline. The spline has been evaluated for all **x** with enclosing interval numbers inside the allowable set.

$\langle$*value*$\rangle$ entries of **x** were indexed below the lower bound $\langle$*value*$\rangle$.
$\langle$*value*$\rangle$ entries of **x** were indexed above the upper bound $\langle$*value*$\rangle$.

7 Accuracy

The computed value of $s(x)$ has negligible error in most practical situations. Specifically, this value has an absolute error bounded in modulus by $18 \times cmax \times$ ***machine precision***, where $cmax$ is the largest in modulus of c_j, $c_j + 1$, $c_j + 2$ and $c_j + 3$, and j is an integer such that $\lambda_j + 3 < x \leq \lambda_j + 4$. If c_j, $c_j + 1$, $c_j + 2$ and $c_j + 3$ are all of the same sign, then the computed value of $s(x)$ has relative error bounded by $20 \times$ ***machine precision***. For full details see Cox (1978).

No complete error analysis is available for the computation of the derivatives of $s(x)$. However, for most practical purposes the absolute errors in the computed derivatives should be small. Note that this is in comparison to the derivatives of the spline, which may or may not be comparable to the derivatives of the function that has been approximated by the spline.

8 Parallelism and Performance

nag_fit_1dspline_deriv_vector (e02bfc) 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

If using the sorted mode of vectorization, the time required for the first phase to determine the enclosing intervals is approximately proportional to $O(n_x \log(\bar{n}))$. The time required to then generate the required permutations and interval information is $O(n_x)$ if **x** is ordered sufficiently, or at worst $O(n_x \min(n_x, \bar{n}) \log(\min(n_x, \bar{n})))$ if **x** is not ordered. The time required by the second phase is then proportional to $O(n_x)$.

If using the unsorted mode of vectorization, the time required is proportional to $O(n_x \log(\bar{n}))$ if the enclosing interval numbers are not provided, or $O(n_x)$ if they are provided. However, the repeated calculation of various quantities will typically make this slower than the sorted mode when the ratio of abscissae to knots is high, or the abscissae are densely distributed over a relatively small subset of the intervals of the spline.

Note: the function does not test all the conditions on the knots given in the description of **spline→lamda** in Section 5, since to do this would result in a computation time with a linear dependency upon $\bar{n}$ instead of $\log(\bar{n})$. All the conditions are tested in nag_1d_spline_fit_knots (e02bac) and nag_1d_spline_fit (e02bec), however.

10 Example

This example fits a spline through a set of data points using nag_1d_spline_fit (e02bec) and then evaluates the spline at a set of supplied abscissae.

10.1 Program Text

```
/* nag_fit_1dspline_deriv_vector (e02bfc) Example Program.
 *
 * Copyright 2014 Numerical Algorithms Group.
 *
 * Mark 24, 2013.
 */

#include <nag.h>
#include <stdio.h>
#include <nag_stdlib.h>
#include <nage02.h>
```

```
int main(void)
{
#define S(I,J) s[(J-1)*pds + I-1]
  Integer               exit_status = 0;
  double                fp, sfac;
  Integer               pds, liwrk, m, nest, nx, d, j;
  double                *s = 0, *wdata = 0, *x = 0, *xdata = 0, *ydata = 0;
  Integer               *iwrk = 0, *ixloc = 0;
  Nag_Comm              warmstartinf;
  Nag_Spline            spline;
  Nag_Start             start_e02bec;
  Nag_SplineVectorSort  start;
  Nag_Boolean           xord;
  Nag_DerivType         deriv;
  NagError              fail;

  printf("nag_fit_1dspline_deriv_vector (e02bfc) Example Program Results\n");

  INIT_FAIL(fail);

  /* Initialize spline */
  spline.lamda = 0;
  spline.c = 0;
  warmstartinf.nag_w =0;
  warmstartinf.nag_iw = 0;

  /* Skip heading in data file*/
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  /* Input the number of data points for the spline,*/
  /* followed by the data points (xdata), the function values (ydata)*/
  /* and the weights (wdata).*/
#ifdef _WIN32
  scanf_s("%"NAG_IFMT"", &m);
#else
  scanf("%"NAG_IFMT"", &m);
#endif
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif
  nest = m + 4;
  if (m >= 4)
    {
      if (!(wdata = NAG_ALLOC(m, double)) ||
          !(xdata = NAG_ALLOC(m, double)) ||
          !(ydata = NAG_ALLOC(m, double)))
        {
          printf("Allocation failure\n");
          exit_status = -1;
          goto END;
        }
    }
  else
    {
      printf("Invalid m.\n");
      exit_status = 1;
      return exit_status;
    }
  start_e02bec = Nag_Cold;

  for (j=0; j<m; j++)
    {
#ifdef _WIN32
      scanf_s("%lf", &xdata[j]);
```

```
#else
      scanf("%lf", &xdata[j]);
#endif
#ifdef _WIN32
      scanf_s("%lf", &ydata[j]);
#else
      scanf("%lf", &ydata[j]);
#endif
#ifdef _WIN32
      scanf_s("%lf", &wdata[j]);
#else
      scanf("%lf", &wdata[j]);
#endif
    }
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  /* Read in the requested smoothing factor.*/
#ifdef _WIN32
  scanf_s("%lf", &sfac);
#else
  scanf("%lf", &sfac);
#endif
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  /* Determine the spline approximation.
   * nag_1d_spline_fit (e02bec).
   * Least squares cubic spline curve fit, automatic knot placement,
   * one variable.
   */
  nag_1d_spline_fit(start_e02bec, m, xdata, ydata, wdata, sfac, nest,
                    &fp, &warmstartinf, &spline, &fail);

  if (fail.code!=NE_NOERROR)
    {
      printf("Error from nag_1d_spline_fit (e02bec).\n%s\n",
             fail.message);
      exit_status = 2;
      goto END;
    }

  /* Read in the number of sample points requested.*/
#ifdef _WIN32
  scanf_s("%"NAG_IFMT"", &nx);
#else
  scanf("%"NAG_IFMT"", &nx);
#endif
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  /* Allocate memory for sample point locations and*/
  /* function and derivative approximations.*/
  pds = nx;
  liwrk = 3 + 3 * nx;
  if (
      !(x = NAG_ALLOC(nx, double))||
      !(s = NAG_ALLOC(pds*4, double))||
      !(ixloc = NAG_ALLOC(nx, Integer))||
      !(iwrk = NAG_ALLOC(liwrk, Integer))
      )
    {
      printf("Allocation failure\n");
      exit_status = -1;
      goto END;
    }

  /* Read in sample points.*/
  for (j=0; j<nx; j++)
#ifdef _WIN32
    scanf_s("%lf", &x[j]);
#else
    scanf("%lf", &x[j]);
#endif
#ifdef _WIN32
  scanf_s("%*[^\n] ");
#else
  scanf("%*[^\n] ");
#endif

  xord  = Nag_FALSE;
  start = Nag_SplineVectorSort_Sorted;
  deriv = Nag_RightDerivs_3;
  /*
   * nag_fit_1dspline_deriv_vector (e02bfc).
   * Evaluation of fitted cubic spline, function and optionally derivatives
   * at a vector of points.
   */
  nag_fit_1dspline_deriv_vector(start, &spline, deriv, xord, x, ixloc, nx,
                                s, pds, iwrk, liwrk, &fail);
  switch (fail.code)
    {
    case NE_NOERROR:
    case NW_SOME_SOLUTIONS:
      {
        /* Output the results.*/
        printf("\n");
        printf("      x   ixloc         s(x) ");
        printf("      ds/dx      d2s/dx2      d3s/dx3\n");
        for (j=0; j<nx; j++)
          {
            if (ixloc[j]>=4 && ixloc[j]<= spline.n - 3)
              {
                printf("%8.4f %7"NAG_IFMT" ",x[j],ixloc[j]);
                for (d=0; d<4; d++)
                  printf("%12.4e ", S(j+1,d+1));
                printf("\n");
              }
            else
              printf("%f %"NAG_IFMT"\n", x[j],ixloc[j]);
          }
        break;
      }
    default:
      {
        printf("Error from nag_fit_1dspline_deriv_vector (e02bfc).\n%s\n",
               fail.message);
        exit_status = 3;
        goto END;
      }
    }
 END:

  NAG_FREE(xdata);
  NAG_FREE(ydata);
  NAG_FREE(wdata);
  NAG_FREE(warmstartinf.nag_w);
  NAG_FREE(warmstartinf.nag_iw);
  NAG_FREE(spline.lamda);
  NAG_FREE(spline.c);
  NAG_FREE(x);
  NAG_FREE(ixloc);
  NAG_FREE(s);
  NAG_FREE(iwrk);

  return exit_status;
}
```

10.2 Program Data

```
nag_fit_1dspline_deriv_vector (e02bfc) Example Program Data
  15                                  : M, the number of data points.
  0.0000E+00  -1.1000E+00   1.00
  5.0000E-01  -3.7200E-01   1.00
  1.0000E+00   4.3100E-01   1.50
  1.5000E+00   1.6900E+00   1.00
  2.0000E+00   2.1100E+00   1.00
  2.5000E+00   3.1000E+00   1.00
  3.0000E+00   4.2300E+00   1.00
  4.0000E+00   4.3500E+00   1.00
  4.5000E+00   4.8100E+00   1.00
  5.0000E+00   4.6100E+00   1.00
  5.5000E+00   4.7900E+00   1.00
  6.0000E+00   5.2300E+00   1.00
  7.0000E+00   6.3500E+00   1.00
  7.5000E+00   7.1900E+00   1.00
  8.0000E+00   7.9700E+00   1.00  : xdata(1:m), ydata(1:m), wdata(1:m)
  0.001                               : S, smoothing factor.

  20                                  : NX, the number of evaluation points.
  6.5178  7.2463  1.0159  7.3070
  5.0589  0.7803  2.2280  4.3751
  7.6601  7.7191  1.2609  7.7647
  7.6573  3.8830  6.4022  1.1351
  3.3741  7.3259  6.3377  7.6759  : Unordered evaluation points x(1:nx).
```

10.3 Program Results

```
nag_fit_1dspline_deriv_vector (e02bfc) Example Program Results

       x    ixloc         s(x)         ds/dx       d2s/dx2       d3s/dx3
     6.5178   14   5.7418e+00   1.0741e+00   5.6736e-01   1.3065e+00
     7.2463   15   6.7486e+00   1.7074e+00   4.9054e-01  -2.8697e+00
     1.0159    5   4.7469e-01   2.4179e+00   3.8175e+00  -2.2171e+01
     7.3070   15   6.8531e+00   1.7319e+00   3.1634e-01  -2.8697e+00
     5.0589   12   4.6105e+00  -1.0363e-01   2.9075e+00  -4.4467e+00
     0.7803    4   6.6885e-03   1.6216e+00   2.5007e+00   7.5980e+00
     2.2280    7   2.4751e+00   1.9559e+00   3.0615e+00  -6.6690e+00
     4.3751   10   4.7199e+00   8.5194e-01  -3.0718e+00  -1.9866e+01
     7.6601   15   7.4633e+00   1.6647e+00  -6.9696e-01  -2.8697e+00
     7.7191   15   7.5602e+00   1.6186e+00  -8.6627e-01  -2.8697e+00
     1.2609    5   1.1273e+00   2.6878e+00  -1.6146e+00  -2.2171e+01
     7.7647   15   7.6330e+00   1.5761e+00  -9.9713e-01  -2.8697e+00
     7.6573   15   7.4586e+00   1.6667e+00  -6.8892e-01  -2.8697e+00
     3.8830    9   4.3152e+00   1.6458e-01   3.1754e+00   1.0296e+01
     6.4022   14   5.6211e+00   1.0172e+00   4.1633e-01   1.3065e+00
     1.1351    5   7.8376e-01   2.7154e+00   1.1746e+00  -2.2171e+01
     3.3741    9   4.4165e+00  -1.1809e-01  -2.0644e+00   1.0296e+01
     7.3259   15   6.8859e+00   1.7374e+00   2.6211e-01  -2.8697e+00
     6.3377   14   5.5563e+00   9.9310e-01   3.3206e-01   1.3065e+00
     7.6759   15   7.4895e+00   1.6534e+00  -7.4230e-01  -2.8697e+00
```