Module 8.2: nag_spline_1d
One-dimensional Spline Fitting

`nag_spline_1d` provides procedures for computing and evaluating spline approximations to arbitrary data sets in one dimension.

Contents

Introduction

This module is concerned with a one-dimensional cubic spline $s(x)$, defined for $x \in [a, b]$ and expressed in its B-spline series representation

$$s(x) = \sum_{i=1}^{p-4} \kappa_i N_i(x),$$

where $N_i(x)$ denotes the normalized cubic B-spline Hayes [15], defined on the knots $\lambda_i, \lambda_{i+1}, \ldots, \lambda_{i+4}$. The derived type `nag_spline_1d_comm_wp` is used in this module to represent $s(x)$ in the above form, which is the most compact form possible, requiring only the $p - 4$ coefficients, and p knot positions, to define $s(x)$ fully.

Such a spline may be used to interpolate (pass exactly through) a given set of data points (x_i, f_i), for $i = 1, 2, \ldots, m$. The procedure `nag_spline_1d_interp` generates such an interpolant. Alternatively, a spline may be used to approximate the points, without actually passing through them. In the latter case it is useful to have some measure of the accuracy of the fit. For this purpose we define the sum of squares of the weighted residuals

$$\theta = \sum_{i=1}^{m} w_i^2 (f_i - s(x_i))^2,$$

where the weights w_i, $i = 1, 2, \ldots, m$ may be used to ensure that f-values known to be more accurate than others have a greater influence on θ.

A *least-squares* spline approximation is one for which the coefficients κ_i have been chosen in order to minimise θ. Typically, a least-squares spline approximation involves significantly fewer coefficients than the corresponding interpolating spline. Its use is much less liable to produce unwanted fluctuations, and so can often provide a better approximation to the function underlying the data. The procedure `nag_spline_1d_lsq_fit` computes a weighted least-squares fit with given interior knots.

A much more automatic fitting procedure can be derived by choosing both the interior knots and the coefficients κ_i in order to optimize some measure of the smoothness of $s(x)$, subject to θ being less than a given threshold. The procedure `nag_spline_1d_auto_fit` implements an algorithm of this type.

The spline is assumed to have a total of p knots $\lambda_1, \lambda_2, \ldots, \lambda_p$. Of these, the first four and the last four are defined by

$$\lambda_1 = \lambda_2 = \lambda_3 = \lambda_4 = a, \qquad \lambda_{p-3} = \lambda_{p-2} = \lambda_{p-1} = \lambda_p = b.$$

The remaining *interior* knots $\lambda_5, \lambda_6, \ldots, \lambda_{p-4}$ are either automatically selected or specified through input arguments, depending on the spline generation procedure used. Coincident knots are permitted as long as their multiplicity does not exceed four. At a knot of multiplicity one (the usual case), $s(x)$ and its first two derivatives are continuous. At a knot of multiplicity two, $s(x)$ and its first derivative are continuous. At a knot of multiplicity three, $s(x)$ is continuous, and at a knot of multiplicity four, $s(x)$ is generally discontinuous.

In addition to the derived type and procedures mentioned above, the module also provides procedures for the evaluation of the spline, its derivatives, and its definite integral.

Procedure: nag_spline_1d_auto_fit

1 Description

nag_spline_1d_auto_fit determines a smooth cubic spline approximation $s(x)$ to the set of data points (x_i, f_i) with weights w_i, for $i = 1, 2, \ldots, m$. The data points are assumed to be ordered such that $x_1 < x_2 < \cdots < x_m$ and the spline is defined on the interval $[a, b] = [x_1, x_m]$. The weights are by default set to unity, but you may specify non-default values by supplying the optional argument `wt`.

The total number of knots p, and their values $\lambda_1, \ldots, \lambda_p$ are chosen automatically by the procedure. The balance between closeness of fit and smoothness of the approximation $s(x)$ is controlled by means of the *smoothing factor* S. If S is too large the spline will be too smooth and information will be lost (underfit); for very large S the procedure returns the weighted least-squares cubic polynomial. If S is too small the spline will pick up too much noise (overfit); for $S = 0$ an interpolating spline is generated. Experimenting with values between these two extremes should result in a good compromise. (See Section 6.4 for advice.)

2 Usage

```
USE nag_spline_1d
CALL nag_spline_1d_auto_fit(start, x, f, smooth, spline  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array `x` must have exactly n elements.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

$m \geq 4$ — the number of data points

3.1 Mandatory Arguments

start — character(len=1), intent(in)

Input: specifies whether a *cold* or *warm* start is required.

If `start` = `'C'` or `'c'`, the procedure will build up the knot set starting from no interior knots. For this *cold* start no initialization of the argument `spline` is required.

If `start` = `'W'` or `'w'`, the procedure will restart the knot-placing strategy using the knots found in a previous call to the procedure. For this *warm* start the structure `spline` must be unchanged from that previous call.

Note: a warm start can save much time in searching for a satisfactory value of `smooth`.

Constraints: `start` = `'C'` or `'c'` on the first call to the procedure; `start` = `'C'`, `'c'`, `'W'`, or `'w'` on subsequent calls.

x(m) — real(kind=*wp*), intent(in)

Input: the data points x_i, for $i = 1, 2, \ldots, m$.

Constraints: `x(1)` $<$ `x(2)` $< \cdots <$ `x(`m`)`.

f(m) — real(kind=*wp*), intent(in)

Input: the values f_i, for $i = 1, 2, \ldots, m$.

smooth — real(kind=*wp*), intent(in)

Input: the smoothing factor S.

If $0.0 \leq$ `smooth` $<$ `EPSILON(1.0_`*wp*`)` the procedure returns an interpolating spline.

Note: for advice on the choice of `smooth` see Section 6.4.

Constraints: `smooth` ≥ 0.0.

spline — type(nag_spline_1d_comm_*wp*), intent(inout)

Input: a structure representing the spline.

If `start` = `'C'` or `'c'`, no initialization of `spline` is required.

If `start` = `'W'` or `'w'`, the structure must be as output from a previous call to this procedure.

Output: a structure containing details of the spline $s(x)$ generated. This structure may be passed to `nag_spline_1d_eval` to evaluate $s(x)$ at given points, or to `nag_spline_1d_intg` to compute its definite integral.

Note: to reduce the risk of corrupting the data accidentally, the components of this structure are private; details of the spline may be extracted by calling `nag_spline_1d_extract`.

The procedure allocates approximately $22m$ real(kind=*wp*) elements, and m integer elements of storage to the structure. If you wish to deallocate this storage when the structure is no longer required, you must call the procedure `nag_deallocate`, as illustrated in Example 1 of this module document.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

wt(*m*) — real(kind=*wp*), intent(in), optional

Input: the values w_i of the weights, for $i = 1, 2, \ldots, m$.

Note: Section 3.3 of the Chapter Introduction gives advice on the choice of weights.

Default: `wt` = 1.0.

Constraints: `wt(`i`)` > 0.0, for $i = 1, 2, \ldots, m$.

p — integer, intent(out), optional

Output: the total number of knots p chosen by this procedure.

theta — real(kind=*wp*), intent(out), optional

Output: the sum of squares of weighted residuals θ, as described in the Module Introduction.

Note: if `theta` = 0.0, this is an interpolating spline. `theta` should equal `smooth` within a relative tolerance of 0.001 unless $p = 8$, when the spline has no interior knots and so is simply a cubic polynomial. For knots to be inserted, `smooth` must be set to a value below the value of `theta` produced in this case.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
303	Array arguments have inconsistent shapes.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	An iterative process has failed to converge. The iterative process used to compute the coefficients of the approximating spline has failed to converge. This error exit may occur if `smooth` has been set very small. If the error persists with increased `smooth`, consult NAG.

If `error%level` = 2 a spline approximation is computed, but fails to satisfy the fitting criterion (see Section 6.1), perhaps by only a small amount, however. If you wish to use this approximation you must supply the optional argument `error` with `error%halt_level` set to 3.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 1 of this module document.

6 Further Comments

6.1 Algorithmic Detail

The coefficients $\kappa_1, \kappa_2, \ldots, \kappa_{p-4}$ are determined as the solution of the following constrained minimisation problem:

$$\text{minimize } \eta = \sum_{i=5}^{p-4} \delta_i^2, \text{ subject to } \theta = \sum_{i=1}^{m} \epsilon_i^2 \leq S,$$

where δ_i stands for the discontinuity jump in the third-order derivative of $s(x)$ at the interior knot λ_i, $\epsilon_i = w_i(f_i - s(x_i))$ denotes the weighted residual, and S is the non-negative value supplied in `smooth`.

The quantity η can be considered as a measure of the lack of smoothness in $s(x)$, while closeness of fit is measured through θ. By means of the parameter S, the *smoothing factor*, you may control the balance between these two (usually conflicting) properties.

If $S = 0$, the requisite number of knots is known in advance, i.e., $p = m + 4$; the interior knots are located immediately as $\lambda_i = x_{i-2}$, for $i = 5, 6, \ldots, p - 4$. The corresponding least-squares spline (see `nag_spline_1d_lsq_fit`) is then an interpolating spline and therefore a solution of the problem.

If $S > 0$, a suitable knot set is built up in stages (starting with no interior knots in the case of a cold start but with the knot set found in a previous call if a warm start is chosen). At each stage, a spline is fitted to the data by least-squares (see `nag_spline_1d_lsq_fit`) and θ, the weighted sum of squares of residuals, is computed. If $\theta > S$, new knots are added to the knot set to reduce θ at the next stage. The number of knots added depends on the value of S and on the progress made so far in reducing θ. The new knots are located in intervals where the fit is particularly poor. At some point in the computation the condition $\theta \leq S$ will be satisfied, and at that point the knot set is accepted. The procedure then goes on to compute the (unique) spline which has this knot set and which satisfies the full fitting criterion specified above. The theoretical solution has $\theta = S$. The procedure computes the spline by an iterative

scheme which is ended when $\theta = S$ within a relative tolerance of 0.001. The main part of each iteration consists of a linear least-squares computation of a special form, done in a similarly stable and efficient manner as in `nag_spline_1d_lsq_fit`.

An exception occurs when the procedure finds at the start that, even with no interior knots ($p = 8$), the least-squares spline already has its weighted sum of squares of residuals $\leq S$. In this case, since this spline (which is simply a cubic polynomial) also has an optimal value for the smoothness measure η, namely zero, it is returned at once as the (trivial) solution. It will usually mean that S has been chosen too large.

For further details of the algorithm and its use, see Dierckx [9], Dierckx [10] and Dierckx [11].

6.2 Accuracy

On successful exit, the approximation returned is such that its residual norm is equal to the smoothing factor S, up to a relative tolerance of 0.001, except that if $p = 8$, it may be significantly less than S; in this case the computed spline is simply a weighted least-squares cubic polynomial approximation, i.e., a spline with no interior knots.

6.3 Timing

The time taken for a call of this procedure depends on the complexity of the shape of the data, the value of the smoothing factor S, and the number of data points. If the procedure is to be called for different values of S, much time can be saved by setting `start = 'W'` after the first call.

6.4 Choice of S

If the weights have been correctly chosen (see Section 3.3 of the Chapter Introduction), the standard deviation of $w_i f_i$ would be the same for all i, equal to σ, say. In this case, choosing the smoothing factor S in the range $[\sigma^2(m-\sqrt{2m}), \sigma^2(m+\sqrt{2m})]$, as suggested by Reinsch [16], is likely to give a good start in the search for a satisfactory value. Otherwise, experimenting with different values of S will be required from the start, taking account of the remarks in Section 1. In that case, in view of computation time and memory requirements, it is recommended to start with a value of S which is so large that no interior knots are inserted ($p = 8$). The spline generated in this case is the least-squares cubic polynomial, and the value returned for `theta`, call it θ_0, gives an upper bound for S. Then progressively decrease the value of S to obtain closer fits, say by a factor of 10 in the beginning, i.e., $S = \theta_0/10$, $S = \theta_0/100$, and so on, and more carefully as the approximation shows more details.

The number of knots of the spline returned, and their location, generally depend on the value of S and on the behaviour of the function underlying the data. If this procedure is called with a warm start, however, the knots returned may also depend on the smoothing factors of the previous calls. Therefore if, after a number of trials with different values of S and warm starts, a fit can finally be accepted as satisfactory, it may be worthwhile to call the procedure once more with the selected value for S but now using `start = 'C'`. Often, this procedure then returns an approximation with the same quality of fit but with fewer knots, which is therefore better if data reduction is also important.

Procedure: nag_spline_1d_lsq_fit

1 Description

nag_spline_1d_lsq_fit determines a least-squares cubic spline approximation $s(x)$ to the set of data points (x_i, f_i) with weights w_i, for $i = 1, 2, \ldots, m$. The data points are assumed to be ordered such that $x_1 \leq x_2 \leq \cdots \leq x_m$ and the spline is defined on the interval $[a, b] = [x_1, x_m]$. The weights are by default set to unity, but you may specify non-default values by supplying the optional argument wt. The spline has a total of p knots, of which you must specify the $p - 8$ *interior* knots λ_5, $\lambda_6, \ldots, \lambda_{p-4}$. Multiple knots are permitted as long as their multiplicity does not exceed four. As mentioned in the Module Introduction, the use of multiple knots generally results in discontinuity of $s(x)$ or its derivatives.

$s(x)$ has the property that it minimizes the sum of squares of weighted residuals

$$\theta = \sum_{i=1}^{m} \epsilon_i^2,$$

where $\epsilon_i = w_i(f_i - s(x_i))$ is the weighted residual at x_i.

2 Usage

```
USE nag_spline_1d
CALL nag_spline_1d_lsq_fit(x, f, lambda, spline  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

This procedure derives the values of the following problem parameters from the shape of the supplied arrays.

m — the number of data points
p — the total number of knots

m and p must satisfy the constraints

$$m \geq m_d \geq 4, \quad 8 \leq p \leq m_d + 4,$$

where m_d is the number of *distinct* values in x.

3.1 Mandatory Arguments

x(m) — real(kind=*wp*), intent(in)
Input: the data points x_i, for $i = 1, 2, \ldots, m$.
Constraints: x(1) $\leq$ x(2) $\leq \cdots \leq$ x(m).

f(m) — real(kind=*wp*), intent(in)
Input: the values f_i, for $i = 1, 2, \ldots, m$.

lambda($p - 8$) — real(kind=*wp*), intent(in)
Input: lambda(i) must contain the interior knot λ_{i+4}, for $i = 1, 2, \ldots, p - 8$.
Note: for advice on the choice of knots see the Further Details section of this module document.
Constraints: x(1) $<$ lambda(1) $\leq \cdots \leq$ lambda($p - 8$) $<$ x(m), and lambda(i) must not have multiplicity > 4.

spline — type(nag_spline_1d_comm_*wp*), intent(out)

Output: a structure containing details of the spline $s(x)$ generated. This structure may be passed to `nag_spline_1d_eval` to evaluate $s(x)$ at given points, or to `nag_spline_1d_intg` to compute its definite integral.

Note: to reduce the risk of corrupting the data accidentally, the components of this structure are private; details of the spline may be extracted by calling `nag_spline_1d_extract`.

The procedure allocates $2p$ real(kind=*wp*) elements of storage to the structure. If you wish to deallocate this storage when the structure is no longer required, you must call the procedure `nag_deallocate`, as illustrated in Example 2 of this module document.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

wt(m) — real(kind=*wp*), intent(in), optional

Input: the values w_i of the weights, for $i = 1, 2, \ldots, m$.

Note: Section 3.3 of the Chapter Introduction gives advice on the choice of weights.

Default: `wt` = 1.0.

Constraints: `wt(`i`)` > 0.0, for $i = 1, 2, \ldots, m$.

theta — real(kind=*wp*), intent(out), optional

Output: the sum of squares of weighted residuals θ.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
303	Array arguments have inconsistent shapes.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	There is no unique solution. The conditions specified by Schoenberg and Whitney [17] fail to hold for at least one subset of the distinct data abscissae. That is, there is no subset of $p-4$ strictly increasing values `x(`r_1`)`, `x(`r_2`)`,..., `x(`r_{p-4}`)` among the abscissae such that $$\begin{array}{ccccc} \mathtt{x}(r_1) & < & \lambda_1 & < & \mathtt{x}(r_5), \\ \mathtt{x}(r_2) & < & \lambda_2 & < & \mathtt{x}(r_6), \\ & & \ldots & & \\ \mathtt{x}(r_{p-8}) & < & \lambda_{p-8} & < & \mathtt{x}(r_{p-4}). \end{array}$$ This means that there is no unique solution: there are regions containing too many knots compared with the number of data points.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 2 of this module document.

6 Further Comments

6.1 Algorithmic Detail

The method employed involves forming and then computing the least-squares solution of a set of m linear equations in the coefficients κ_i, for $i = 1, 2, \ldots, p-4$. The equations are formed using a recurrence relation for B-splines that is unconditionally stable (see Cox [1], De Boor [8]), even for multiple (coincident) knots. The least-squares solution is also obtained in a stable manner by using orthogonal transformations, namely a variant of Givens rotations (see Gentleman [13] and Gentleman [14]). This requires only one equation to be stored at a time. Full advantage is taken of the structure of the equations, there being at most four non-zero values of $N_i(x)$ for any value of x and hence at most four coefficients in each equation. For further details of the algorithm and its use see Cox [2], Cox [4] and Cox and Hayes [7].

6.2 Accuracy

The rounding errors committed are such that the computed coefficients are exact for a slightly perturbed set of ordinates $f_i + \delta f_i$. The ratio of the root-mean-square value for the δf_i to the root-mean-square value of the f_i can be expected to be less than a small multiple of $C \times m \times$`EPSILON(1.0_wp)`, where C is a condition number for the problem (see Cox [4]). In general we would not expect C to be large unless the choice of knots results in near-violation of the Schoenberg–Whitney conditions (see Schoenberg and Whitney [17]).

A cubic spline which adequately fits the data and is free from spurious oscillations is more likely to be obtained if the knots are chosen to be grouped more closely in regions where the function (underlying the data) or its derivatives change more rapidly than elsewhere.

6.3 Timing

The time taken by the procedure is proportional to $2m + p$.

Procedure: nag_spline_1d_interp

1 Description

nag_spline_1d_interp determines a cubic spline $s(x)$, defined in the range $[a, b] = [x_1, x_m]$, which interpolates (passes exactly through) the set of data points (x_i, f_i), for $i = 1, 2, \ldots, m$, where $m \geq 4$ and $x_1 < x_2 < \ldots < x_m$. The spline has a total of $p = m + 4$ knots, of which the $m - 4$ interior knots $\lambda_5, \lambda_6, \ldots, \lambda_m$ are set to the values of $x_3, x_4, \ldots, x_{m-2}$ respectively.

2 Usage

```
USE nag_spline_1d
CALL nag_spline_1d_interp(x, f, spline  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

$m \geq 4$ — the number of data points

3.1 Mandatory Arguments

x(m) — real(kind=*wp*), intent(in)

Input: the data points x_i, for $i = 1, 2, \ldots, m$.

Constraints: x(1) < x(2) < ⋯ < x(m).

f(m) — real(kind=*wp*), intent(in)

Input: the values f_i, for $i = 1, 2, \ldots, m$.

spline — type(nag_spline_1d_comm_*wp*), intent(out)

Output: a structure containing details of the spline $s(x)$ generated. This structure may be passed to nag_spline_1d_eval to evaluate $s(x)$ at given points, or to nag_spline_1d_intg to compute its definite integral.

Note: to reduce the risk of corrupting the data accidentally, the components of this structure are private; details of the spline may be extracted by calling nag_spline_1d_extract.

The procedure allocates approximately $2m$ real(kind=*wp*) elements of storage to the structure. If you wish to deallocate this storage when the structure is no longer required, you must call the procedure nag_deallocate, as illustrated in Example 3 of this module document.

3.2 Optional Argument

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document nag_error_handling (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to nag_set_error before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
303	Array arguments have inconsistent shapes.
320	The procedure was unable to allocate enough memory.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 3 of this module document.

6 Further Comments

6.1 Algorithmic Detail

Unlike some other spline interpolation algorithms, derivative end conditions are not imposed on the spline. All the x_i are used as knot positions except x_2 and x_{m-1}. This choice of knots (see Cox [5]) means that $s(x)$ is composed of $m-3$ cubic arcs as follows. If $m = 4$, there is just a single arc space spanning the whole interval $[x_1, x_4]$. If $m \geq 5$, the first arc spans the interval $[x_1, x_3]$ and the last arc spans $[x_{m-2}, x_m]$. Additionally, if $m \geq 6$, the ith arc, for $i = 2, 3, \ldots, m-4$, spans $[x_{i+1}, x_{i+2}]$.

The algorithm for determining the coefficients is as described in Cox [3], except that QR factorization is used instead of LU decomposition. The implementation of the algorithm involves setting up appropriate information for the related procedure `nag_spline_1d_lsq_fit`, followed by a call to that procedure. For further details of `nag_spline_1d_lsq_fit`, see the procedure document .

6.2 Accuracy

The rounding errors committed are such that the computed coefficients are exact for a slightly perturbed set of ordinates $f_i + \delta f_i$. The ratio of the root-mean-square value for the δf_i to the root-mean-square value of the f_i is no greater than a small multiple of `EPSILON(1.0_wp)`.

6.3 Timing

The time taken by the procedure is approximately proportional to m.

Procedure: nag_spline_1d_eval

1 Description

nag_spline_1d_eval evaluates a cubic spline $s(x)$, and optionally its first three derivatives, at a set of points u_i, $i = 1, 2, \ldots, n$. Since $s(x)$ is a piecewise cubic function the higher derivatives are all zero.

Each evaluation point must lie within the region of definition $[a, b]$ of the spline, as described in Section 1 of the procedure document for its generation procedure.

If an evaluation point coincides with one or more of the knots of the spline the required values of s and its derivatives are not in general continuous. By default, at points of discontinuity, left-hand values are calculated; this may be overridden by means of the optional argument right_hand.

2 Usage

```
USE nag_spline_1d
CALL nag_spline_1d_eval(spline, u, s  [, optional arguments])
```

2.1 Interfaces

Distinct interfaces are provided for the following cases.

Evaluation at an array of points / at a single point

Array of points: u, s and the optional arguments sd1, sd2 and sd3 are all rank-1 arrays.

Single point: u, s and the optional arguments sd1, sd2 and sd3 are all scalar.

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

$n \geq 1$ — the number of evaluation points

3.1 Mandatory Arguments

spline — type(nag_spline_1d_comm_*wp*), intent(in)

Input: a structure containing details of the spline $s(x)$ to be evaluated.

Constraints: spline must be as output from a previous call to nag_spline_1d_auto_fit, nag_spline_1d_lsq_fit, nag_spline_1d_interp, or nag_spline_1d_set.

u(n) / u — real(kind=*wp*), intent(in)

Input: the point(s) u_i, for $i = 1, 2, \ldots, n$, at which $s(x)$ is to be evaluated.

Constraints: $a \leq$ u(i) $\leq b$, where $[a, b]$ is the interval of definition of the spline, as described in the procedure document for its generation procedure.

Note: if $n = 1$, u may be declared as a scalar. In this case the constraint is $a \leq$ u $\leq b$.

s(n) / s — real(kind=*wp*), intent(out)

Output: the value(s) of the spline $s(u_i)$, for $i = 1, 2, \ldots, n$.

Note: s must have the same rank as u.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

sd1(n) **/ sd1** — real(kind=*wp*), intent(out), optional

Output: the value(s) of the first derivative of the spline $s'(u_i)$, for $i = 1, 2, \ldots, n$.

Note: `sd1` must have the same rank as `u`.

sd2(n) **/ sd2** — real(kind=*wp*), intent(out), optional

Output: the value(s) of the second derivative of the spline $s''(u_i)$, for $i = 1, 2, \ldots, n$.

Note: `sd2` must have the same rank as `u`.

sd3(n) **/ sd3** — real(kind=*wp*), intent(out), optional

Output: the value(s) of the third derivative of the spline $s'''(u_i)$, for $i = 1, 2, \ldots, n$.

Note: `sd3` must have the same rank as `u`.

right_hand — logical, intent(in), optional

Input: determines whether left-hand or right-hand function and derivative values are computed should any u_i coincide with a knot.

If `right_hand` = `.true.` right-hand values are computed.

If `right_hand` = `.false.` left-hand values are computed.

Default: `right_hand` = `.false.`.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
303	Array arguments have inconsistent shapes.

5 Examples of Usage

Complete examples of the use of this procedure appear in Examples 3 and 4 of this module document.

6 Further Comments

6.1 Algorithmic Detail

The method of evaluation for each point u_i is based on the following steps.

1. Carry out a binary search for the knot interval containing the point u_i, (see Cox [6]).
2. Evaluate the non-zero B-splines of orders 1, 2, 3 and 4 by recurrence (see Cox [1], Cox [6]).
3. Compute all derivatives of the B-splines of order 4 by applying a second recurrence to these computed B-spline values (see Cox [1]).

4. Multiply the fourth-order B-spline values and their derivatives by the appropriate B-spline coefficients, and sum, to yield the value of the spline and its derivatives at u_i.

If derivatives are not required the procedure uses a more efficient method of taking convex combinations due to De Boor [8]. For further details of the algorithm and its use see Cox [1].

6.2 Accuracy

The computed values of $s(x)$ have negligible error in most practical situations. Specifically, a value has an *absolute* error bounded in modulus by $18\kappa_{\max}\times$`EPSILON(1.0_wp)`, where $\kappa_{\max}$ is the largest in modulus of κ_j, κ_{j+1}, κ_{j+2} and κ_{j+3}, and j is an integer such that $\lambda_{j+3} \leq x \leq \lambda_{j+4}$. If κ_j, κ_{j+1}, κ_{j+2} and κ_{j+3} are all of the same sign, then the computed value of $s(x)$ has a *relative* error not exceeding $20\times$`EPSILON(1.0_wp)` in modulus. For further details see Cox [6].

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.

6.3 Timing

The time required to evaluate the spline and its derivatives at a given point varies linearly with $\log(p)$, where p is the total number of knots. If the value of p is not known it can be determined by calling `nag_spline_1d_extract`. If no derivatives are required a faster method of evaluation is used.

Procedure: nag_spline_1d_intg

1 Description

The function nag_spline_1d_intg evaluates the definite integral

$$I = \int_{\alpha}^{\beta} s(x)\, dx$$

of a cubic spline $s(x)$.

By default the region of integration $[\alpha, \beta]$ is taken as the region of definition $[a, b]$ of $s(x)$, as determined by the particular procedure used to produce the spline. Details may be found in Section 1 of the relevant procedure document.

Integration may be performed over non-default regions lying within $[a, b]$ by supplying one or both of the optional arguments alpha and beta.

2 Usage

```
USE nag_spline_1d
[value =] nag_spline_1d_intg(spline  [, optional arguments])
```

The function result value is a scalar of type real(kind=*wp*).

3 Arguments

3.1 Mandatory Argument

spline — type(nag_spline_1d_comm_*wp*), intent(in)

Input: a structure containing details of the spline $s(x)$ to be integrated.

Constraints: spline must be as output from a previous call to nag_spline_1d_auto_fit, nag_spline_1d_lsq_fit, nag_spline_1d_interp, or nag_spline_1d_set.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

alpha — real(kind=*wp*), intent(in), optional

beta — real(kind=*wp*), intent(in), optional

Input: the lower and upper limits α and β of the integral.

Constraints: $a \leq$ alpha $\leq b$ and $a \leq$ beta $\leq b$. See Section 1.

Note: it is *not* required that alpha $<$ beta.

Default: alpha $= a$, beta $= b$.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document nag_error_handling (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to nag_set_error before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.

5 Examples of Usage

Complete examples of the use of this procedure appear in Examples 2 and 4 of this module document.

6 Further Comments

6.1 Algorithmic Detail

The method employed is described in Section 1.3.3 of Dierckx [12].

6.2 Timing

The time taken by this function is approximately proportional to the total number of knots p. If the value of p is not known it can be determined by calling `nag_spline_1d_extract`.

Procedure: nag_spline_1d_set

1 Description

nag_spline_1d_set creates a structure of type nag_spline_1d_comm_*wp* containing details of a given cubic spline

$$s(x) = \sum_{i=1}^{p-4} \kappa_i N_i(x), \quad x \in [a, b],$$

where $N_i(x)$ is the normalized cubic B-spline defined on the knots $\lambda_i, \ldots, \lambda_{i+4}$. The interior knots $\lambda_5, \cdots, \lambda_{p-4}$ must satisfy

$$a < \lambda_5 \leq \lambda_6 \leq \cdots \leq \lambda_{p-3} \leq \lambda_{p-4} < b,$$

and each interior knot must have maximum multiplicity four. The knot set is completed by setting the first four knots to a and the last four to b.

Note that this procedure cannot be used in conjunction with nag_spline_1d_auto_fit with a warm start.

2 Usage

```
USE nag_spline_1d
CALL nag_spline_1d_set(a, b, lambda, kappa, spline  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

$p \geq 8$ — the total number of knots

3.1 Mandatory Arguments

a — real(kind=*wp*), intent(in)

b — real(kind=*wp*), intent(in)

Input: the lower and upper limits a and b of the region on which the spline is defined.

Constraints: a < b.

lambda($p-8$) — real(kind=*wp*), intent(in)

Input: lambda(i) must contain the interior knot λ_{i+4}, for $i = 1, 2, \ldots, p-8$.

Constraints: a < lambda(1) $\leq \cdots \leq$ lambda($p-8$) < b, and lambda(i) must not have multiplicity > 4.

kappa($p-4$) — real(kind=*wp*), intent(in)

Input: kappa(i) must contain the coefficient κ_i, for $i = 1, 2, \ldots, p-4$, in the B-spline representation of $s(x)$.

spline — type(nag_spline_1d_comm_*wp*), intent(out)

Output: a structure containing details of the spline $s(x)$ generated. This structure may be passed to `nag_spline_1d_eval` to evaluate $s(x)$ at given points, or to `nag_spline_1d_intg` to compute its definite integral.

Note: to reduce the risk of corrupting the data accidentally, the components of this structure are private; details of the spline may be extracted by calling `nag_spline_1d_extract`.

The procedure allocates $2p$ real(kind=*wp*) elements of storage to the structure. If you wish to deallocate this storage when the structure is no longer required, you must call the procedure `nag_deallocate`, as illustrated in Example 4 of this module document.

3.2 Optional Argument

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
303	Array arguments have inconsistent shapes.
320	The procedure was unable to allocate enough memory.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 4 of this module document.

6 Further Comments

6.1 Algorithmic Detail

This procedure may be used for example to initialize a spline with data which has been read from file. The procedure `nag_spline_1d_extract` may be used to extract the data to be written to file.

Procedure: nag_spline_1d_extract

1 Description

Given a structure of type nag_spline_1d_comm_*wp* representing a cubic spline

$$s(x) = \sum_{i=1}^{p-4} \kappa_i N_i(x), \quad x \in [a, b],$$

this procedure optionally returns the total number of knots p, the interval on which the spline is defined $[a, b]$, the interior knots $\lambda_5, \ldots, \lambda_{p-4}$, and the B-spline coefficients $\kappa_1, \ldots, \kappa_{p-4}$.

Since the number of knots may not be known prior to a call to this procedure the arguments which return the knots and B-spline coefficients are pointers, which are allocated internally. It is your responsibility to deallocate this storage.

2 Usage

```
USE nag_spline_1d
CALL nag_spline_1d_extract(spline  [, optional arguments])
```

3 Arguments

3.1 Mandatory Argument

spline — type(nag_spline_1d_comm_*wp*), intent(in)

Input: a structure containing details of the spline $s(x)$.

Constraints: spline must be as output from a previous call to nag_spline_1d_auto_fit, nag_spline_1d_lsq_fit, nag_spline_1d_interp, or nag_spline_1d_set.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

p — integer, intent(out), optional

Output: the total number of knots p in the spline.

a — real(kind=*wp*), intent(out), optional

b — real(kind=*wp*), intent(out), optional

Output: the lower and upper limits a and b of the region of definition of $s(x)$ respectively.

lambda(:) — real(kind=*wp*), pointer, optional

Output: lambda(i) holds the interior knot λ_{i+4}, for $i = 1, 2, \ldots, p-8$.

Note: this array is allocated by this procedure. It should be deallocated when no longer required.

kappa(:) — real(kind=*wp*), pointer, optional

Output: kappa(i) holds the coefficient κ_i, for $i = 1, 2, \ldots, p-4$.

Note: this array is allocated by this procedure. It should be deallocated when no longer required.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document nag_error_handling (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to nag_set_error before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 1 of this module document.

6 Further Comments

6.1 Algorithmic Detail

This procedure may be used for example to extract data from the structure spline in order to write to file. The procedure nag_spline_1d_set may then be used to initialize a new spline with this data.

If called with no optional arguments this procedure merely checks that the structure spline has been created by one of the spline generation procedures of this module.

Derived Type: nag_spline_1d_comm_*wp*

Note. The names of derived types containing real/complex components are precision dependent. For double precision the name of this type is `nag_spline_1d_comm_dp`. For single precision the name is `nag_spline_1d_comm_sp`. Please read the Users' Note for your implementation to check which precisions are available.

1 Description

The derived type `nag_spline_1d_comm_wp` is used to represent a one-dimensional cubic spline $s(x)$, in B-spline series form, as described in the Module Introduction.

The procedures `nag_spline_1d_auto_fit`, `nag_spline_1d_lsq_fit`, `nag_spline_1d_interp` and `nag_spline_1d_set` return structures of this type suitable for passing to `nag_spline_1d_eval`, `nag_spline_1d_intg` and `nag_spline_1d_extract`.

These generation procedures allocate storage to the pointer components of the structure. For details of the amount of storage allocated see the description of the argument `spline` in the relevant procedure document.

If you wish to deallocate the storage when the structure is no longer required, you must call the generic deallocation procedure `nag_deallocate`, which is described in the module document `nag_lib_support` (1.1).

The generation procedures check whether the structure has already had storage allocated to it in a previous call; if it has, that storage is deallocated before allocating the storage required for the new call.

The components of this type are private.

2 Type Definition

```
type nag_spline_1d_comm_wp
  private
  .
  .
  .
end type nag_spline_1d_comm_wp
```

3 Components

In order to reduce the risk of accidental data corruption the components of this type are private and may not be accessed directly.

The procedures `nag_spline_1d_set` and `nag_spline_1d_extract` may be used to initialize and extract data from structures of the type.

Example 1: Spline fitting with automatic knot selection

Generate a spline approximation to a set of data points with automatic knot selection, using several different values of the smoothing factor S. For each value of S output the total number of knots, the values of the interior knots, and the residual sum of squares.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
PROGRAM nag_spline_1d_ex01

  ! Example Program Text for nag_spline_1d
  ! NAG fl90, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_spline_1d, ONLY : nag_spline_1d_comm_wp => nag_spline_1d_comm_dp &
   , nag_spline_1d_auto_fit, nag_spline_1d_extract, nag_deallocate
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: m, p
  REAL (wp) :: smooth, theta
  CHARACTER (1) :: start
  TYPE (nag_spline_1d_comm_wp) :: spline
  ! .. Local Arrays ..
  REAL (wp), ALLOCATABLE :: f(:), wt(:), x(:)
  REAL (wp), POINTER :: lambda(:)
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_spline_1d_ex01.'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) m

  ALLOCATE (x(m),f(m),wt(m))   ! Allocate storage

  READ (nag_std_in,*) x
  READ (nag_std_in,*) f
  READ (nag_std_in,*) wt

  start = 'cold start'

  DO

    ! Read in successive values of smooth and generate spline for
    ! each.
    READ (nag_std_in,*,end=20) smooth

    ! Determine the spline approximation.

    CALL nag_spline_1d_auto_fit(start,x,f,smooth,spline,wt=wt,theta=theta)

    ! Extract the knots.

    CALL nag_spline_1d_extract(spline,p=p,lambda=lambda)
```

```
        WRITE (nag_std_out,'(//1X,A,1P,E13.6)') &
         'Calling with smoothing factor =', smooth
        WRITE (nag_std_out,'(/1X,A,I8)') 'Total number of knots =', p
        WRITE (nag_std_out,'(1X,A/(2X,1PE12.6))') 'Interior knots', lambda
        WRITE (nag_std_out,'(/1X,A,1PE12.4)') 'Residual sum of squares =', &
         theta

        start = 'warm start'

      END DO

20    CONTINUE

      DEALLOCATE (x,f,wt,lambda)   ! Deallocate storage

      NULLIFY (lambda)

      CALL nag_deallocate(spline)  ! Free structure allocated by NAG fl90

    END PROGRAM nag_spline_1d_ex01
```

2 Program Data

```
Example Program Data for nag_spline_1d_ex01
  15                                                          : m
  0.0000E+00  5.0000E-01  1.0000E+00  1.5000E+00  2.0000E+00
  2.5000E+00  3.0000E+00  4.0000E+00  4.5000E+00  5.0000E+00
  5.5000E+00  6.0000E+00  7.0000E+00  7.5000E+00  8.0000E+00 : End of x
 -1.1000E+00 -3.7200E-01  4.3100E-01  1.6900E+00  2.1100E+00
  3.1000E+00  4.2300E+00  4.3500E+00  4.8100E+00  4.6100E+00
  4.7900E+00  5.2300E+00  6.3500E+00  7.1900E+00  7.9700E+00 : End of f
  1.00  2.00  1.50  1.00  3.00  1.00  0.50  1.00
  2.00  2.50  1.00  3.00  1.00  2.00  1.00                    : End of wt
  1.0                                                  : 1st smooth value
  0.5                                                  : 2nd smooth value
  0.1                                                  : 3rd smooth value
```

3 Program Results

```
 Example Program Results for nag_spline_1d_ex01.


 Calling with smoothing factor = 1.000000E+00

 Total number of knots =       9
 Interior knots
  4.000000E+00

 Residual sum of squares =  1.0003E+00


 Calling with smoothing factor = 5.000000E-01

 Total number of knots =      13
 Interior knots
  1.000000E+00
  2.000000E+00
  4.000000E+00
  5.000000E+00
  6.000000E+00
```

```
Residual sum of squares =  5.0010E-01


Calling with smoothing factor = 1.000000E-01

Total number of knots =      16
Interior knots
 1.000000E+00
 1.500000E+00
 2.000000E+00
 3.000000E+00
 4.000000E+00
 4.500000E+00
 5.000000E+00
 6.000000E+00

Residual sum of squares =  1.0000E-01
```

Example 2: Least-squares spline fitting

Determine a weighted least-squares cubic spline approximation with 12 knots (four interior knots) to a set of 14 given data points. Calculate the residual sum of squares and the definite integral of the spline over the interval [x(1),x(14)], on which it is defined.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
PROGRAM nag_spline_1d_ex02

  ! Example Program Text for nag_spline_1d
  ! NAG fl90, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_spline_1d, ONLY : nag_spline_1d_comm_wp => nag_spline_1d_comm_dp &
   , nag_spline_1d_lsq_fit, nag_spline_1d_intg, nag_deallocate
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: m, p
  REAL (wp) :: integral, theta
  TYPE (nag_spline_1d_comm_wp) :: spline
  ! .. Local Arrays ..
  REAL (wp), ALLOCATABLE :: f(:), lambda(:), wt(:), x(:)
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_spline_1d_ex02'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) m, p

  ALLOCATE (x(m),f(m),wt(m),lambda(p-8)) ! Allocate storage

  READ (nag_std_in,*) x
  READ (nag_std_in,*) f
  READ (nag_std_in,*) wt
  READ (nag_std_in,*) lambda

  ! Fit spline and output residual norm.

  CALL nag_spline_1d_lsq_fit(x,f,lambda,spline,wt=wt,theta=theta)

  WRITE (nag_std_out,'(/1X,A,E12.4)') 'Residual sum of squares = ', theta

  ! Evaluate spline integral over region of definition.

  integral = nag_spline_1d_intg(spline)

  WRITE (nag_std_out,'(/1X,A,E12.4)') 'Spline integral = ', integral

  DEALLOCATE (x,f,wt,lambda)   ! Deallocate storage

  CALL nag_deallocate(spline)  ! Free structure allocated by NAG fl90

END PROGRAM nag_spline_1d_ex02
```

2 Program Data

```
Example Program Data for nag_spline_1d_ex02
14 12                                 : m, p
0.20 0.47 0.74 1.09 1.60 1.90 2.60
3.10 4.00 5.15 6.17 8.00 10.00 12.00 : End of x
0.00 2.00 4.00 6.00 8.00 8.62 9.10
8.90 8.15 7.00 6.00 4.54 3.39 2.56   : End of f
0.20 0.20 0.30 0.70 0.90 1.00 1.00
1.00 0.80 0.50 0.70 1.00 1.00 1.00   : End of wt
1.50 2.60 4.00 8.00                  : End of lambda
```

3 Program Results

```
 Example Program Results for nag_spline_1d_ex02

 Residual sum of squares =   0.1783E-02

 Spline integral =   0.6617E+02
```

Example 3: Spline interpolation

Interpolate the exponential function from 7 values lying in the interval $[0, 1]$. Evaluate the spline at the abscissae and at points lying halfway between them.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
PROGRAM nag_spline_1d_ex03

  ! Example Program Text for nag_spline_1d
  ! NAG fl90, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_spline_1d, ONLY : nag_spline_1d_comm_wp => nag_spline_1d_comm_dp &
   , nag_spline_1d_interp, nag_spline_1d_eval, nag_deallocate
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC EXP, KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, m, n
  TYPE (nag_spline_1d_comm_wp) :: spline
  ! .. Local Arrays ..
  REAL (wp), ALLOCATABLE :: f(:), s(:), u(:), x(:)
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_spline_1d_ex03'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) m
  n = 2*m - 1

  ALLOCATE (x(m),f(m),u(n),s(n)) ! Allocate storage

  READ (nag_std_in,*) x
  f = EXP(x)

  ! Construct interpolating spline.

  CALL nag_spline_1d_interp(x,f,spline)

  ! Calculate values of the spline at the x(i)
  ! and at points halfway between them.
  u(1:n:2) = x(1:m)
  u(2:n-1:2) = 0.5_wp*(x(1:m-1)+x(2:m))

  CALL nag_spline_1d_eval(spline,u,s)

  WRITE (nag_std_out,'(/,7X,''x'',11X,''s(x)'')')
  DO i = 1, n
    WRITE (nag_std_out,'(2(3X,E10.4))') u(i), s(i)
  END DO

  DEALLOCATE (x,f,u,s)         ! Deallocate storage

  CALL nag_deallocate(spline)  ! Free structure allocated by NAG fl90
```

```
    END PROGRAM nag_spline_1d_ex03
```

2 Program Data

```
Example Program Data for nag_spline_1d_ex03
7                                   : m
0.00 0.20 0.40 0.60 0.75 0.90 1.00 : End of x
```

3 Program Results

```
 Example Program Results for nag_spline_1d_ex03

       x           s(x)
   0.0000E+00   0.1000E+01
   0.1000E+00   0.1105E+01
   0.2000E+00   0.1221E+01
   0.3000E+00   0.1350E+01
   0.4000E+00   0.1492E+01
   0.5000E+00   0.1649E+01
   0.6000E+00   0.1822E+01
   0.6750E+00   0.1964E+01
   0.7500E+00   0.2117E+01
   0.8250E+00   0.2282E+01
   0.9000E+00   0.2460E+01
   0.9500E+00   0.2586E+01
   0.1000E+01   0.2718E+01
```

Example 4: Initializing a spline

Initialize a spline defined on a given region $[a, b]$ with interior knots $\lambda_5, \ldots, \lambda_{p-4}$ and B-spline coefficients $\kappa_1, \ldots, \kappa_{p-4}$. Evaluate the left and right limits of the spline and its derivatives on a uniform mesh and compute its integral on a region $[\alpha, \beta]$.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
PROGRAM nag_spline_1d_ex04

  ! Example Program Text for nag_spline_1d
  ! NAG fl90, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_spline_1d, ONLY : nag_spline_1d_comm_wp => nag_spline_1d_comm_dp &
   , nag_spline_1d_set, nag_spline_1d_eval, nag_spline_1d_intg, &
   nag_deallocate
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: i, n, p
  REAL (wp) :: a, alpha, b, beta, integral, s, sd1, sd2, sd3, x
  TYPE (nag_spline_1d_comm_wp) :: spline
  ! .. Local Arrays ..
  REAL (wp), ALLOCATABLE :: kappa(:), lambda(:)
  ! .. Executable Statements ..

  WRITE (nag_std_out,*) 'Example Program Results for nag_spline_1d_ex04'

  READ (nag_std_in,*)          ! Skip heading in data file
  READ (nag_std_in,*) a, b
  READ (nag_std_in,*) p

  ALLOCATE (lambda(p-8),kappa(p-4)) ! Allocate storage

  READ (nag_std_in,*) lambda
  READ (nag_std_in,*) kappa
  READ (nag_std_in,*) n
  READ (nag_std_in,*) alpha, beta

  ! Initialize spline.

  CALL nag_spline_1d_set(a,b,lambda,kappa,spline)

  ! Calculate values of the spline and its derivatives on a uniform
  ! mesh.
  WRITE (nag_std_out,'(/,6X,A,13X,A,6X,A,3X,A,3X,A)') 'x', 'spline', &
   '1st deriv', '2nd deriv', '3rd deriv'
  x = 0.0_wp
  DO i = 1, n

    CALL nag_spline_1d_eval(spline,x,s,sd1=sd1,sd2=sd2,sd3=sd3)

    WRITE (nag_std_out,'(/E12.4,1X,A,4E12.4)') x, 'LEFT ', s, sd1, sd2, &
```

```
        sd3

        CALL nag_spline_1d_eval(spline,x,s,sd1=sd1,sd2=sd2,sd3=sd3, &
         right_hand=.TRUE.)

        WRITE (nag_std_out,'(E12.4,1X,A,4E12.4)') x, 'RIGHT', s, sd1, sd2, sd3
        x = x + 1.0_wp
      END DO

      ! Evaluate spline integral on (alpha, beta).

      integral = nag_spline_1d_intg(spline,alpha=alpha,beta=beta)

      WRITE (nag_std_out,'(/1X,A,E12.4)') 'Spline integral = ', integral

      DEALLOCATE (lambda,kappa)    ! Deallocate storage

      CALL nag_deallocate(spline)  ! Free structure allocated by NAG fl90

    END PROGRAM nag_spline_1d_ex04
```

2 Program Data

```
Example Program Data for nag_spline_1d_ex04
0.0000E+00 6.0000E+00          : a, b
14                             : p
 1.0  3.0  3.0  3.0  4.0  4.0  : End of lambda
10.0 12.0 13.0 15.0 22.0
26.0 24.0 18.0 14.0 12.0       : End of kappa
7                              : n
0.0 1.5                        : alpha, beta
```

3 Program Results

```
 Example Program Results for nag_spline_1d_ex04

      x             spline      1st deriv   2nd deriv   3rd deriv

  0.0000E+00 LEFT   0.1000E+02  0.6000E+01 -0.1000E+02  0.1067E+02
  0.0000E+00 RIGHT  0.1000E+02  0.6000E+01 -0.1000E+02  0.1067E+02

  0.1000E+01 LEFT   0.1278E+02  0.1333E+01  0.6667E+00  0.1067E+02
  0.1000E+01 RIGHT  0.1278E+02  0.1333E+01  0.6667E+00  0.3917E+01

  0.2000E+01 LEFT   0.1510E+02  0.3958E+01  0.4583E+01  0.3917E+01
  0.2000E+01 RIGHT  0.1510E+02  0.3958E+01  0.4583E+01  0.3917E+01

  0.3000E+01 LEFT   0.2200E+02  0.1050E+02  0.8500E+01  0.3917E+01
  0.3000E+01 RIGHT  0.2200E+02  0.1200E+02 -0.3600E+02  0.3600E+02

  0.4000E+01 LEFT   0.2200E+02 -0.6000E+01  0.0000E+00  0.3600E+02
  0.4000E+01 RIGHT  0.2200E+02 -0.6000E+01  0.0000E+00  0.1500E+01

  0.5000E+01 LEFT   0.1625E+02 -0.5250E+01  0.1500E+01  0.1500E+01
  0.5000E+01 RIGHT  0.1625E+02 -0.5250E+01  0.1500E+01  0.1500E+01

  0.6000E+01 LEFT   0.1200E+02 -0.3000E+01  0.3000E+01  0.1500E+01
  0.6000E+01 RIGHT  0.1200E+02 -0.3000E+01  0.3000E+01  0.1500E+01

 Spline integral =   0.1836E+02
```

Further Details

1 Alternative Interpolation Procedures

`nag_spline_1d_interp` computes an interpolating cubic spline, to a set of m data points, using a particular choice for the set of knots which has proved generally satisfactory in practice. If you wish to choose a different set of knots, the least-squares approximating procedure `nag_spline_1d_lsq_fit` will also produce an interpolating spline if it is supplied with exactly $m + 4$ knots and the data points are distinct.

The cubic spline does not always avoid unwanted fluctuations, especially when the data show a steep slope close to a region of small slope, or when the data inadequately represent the underlying curve. In such cases you should consider using the procedure `nag_pch_monot_interp` from the module `nag_pch_interp` (8.1) as an alternative.

2 Choice of Knots for `nag_spline_1d_lsq_fit`

`nag_spline_1d_lsq_fit` fits to arbitrary data points, with arbitrary weights, a least-squares cubic spline approximant with given interior knots. The choice of these knots so as to give an acceptable fit must largely be a matter of trial and error, though with a little experience a satisfactory choice can often be made after one or two trials. It is usually best to start with a small number of knots (too many will result in unwanted fluctuations in the fit, or even in there being no unique solution) and, examining the fit graphically at each stage, to add a few knots at a time at places where the fit is particularly poor. Moving the existing knots towards these places will also often improve the fit. In regions where the behaviour of the curve underlying the data is changing rapidly, closer knots will be needed than elsewhere. Otherwise, positioning is not usually very critical and equally-spaced knots are often satisfactory.

A useful feature of the procedure is that it can be used in applications which require the continuity to be less than the normal continuity of the cubic spline. For example, the approximant may be required to have a discontinuous slope at some point in the range. This can be achieved by placing three coincident knots at the given point. Similarly a discontinuity in the second derivative at a point can be achieved by placing two knots there. Analogy with these discontinuous cases can provide guidance in more usual cases: for example, just as three coincident knots can produce a discontinuity in slope, so three close knots can produce a rapid change in slope. The closer the knots are, the more rapid the change can be.

An example set of data is given in Figure 1. It is a rather tricky set, because of the scarcity of data on the right, but it will serve to illustrate some of the above points and to show some of the dangers to be avoided. Three interior knots (indicated by the vertical lines at the top of the diagram) are chosen as a start. We see that the resulting curve is not steep enough in the middle and fluctuates at both ends, severely on the right. The spline is unable to cope with the shape and more knots are needed.

In Figure 2, three knots have been added in the centre, where the data shows a rapid change in behaviour, and one further out towards each end, where the fit is poor. The fit is still poor, so a further knot is added in this region and, in Figure 3, disaster ensues in rather spectacular fashion.

The reason is that, at the right-hand end, the fits in Figure 1 and 2 have been interpreted as poor simply because of the fluctuations about the curve underlying the data (or what it is naturally assumed to be). But the fitting process knows only about the data and nothing else about the underlying curve, so it is important to consider only closeness to the data when deciding goodness of fit.

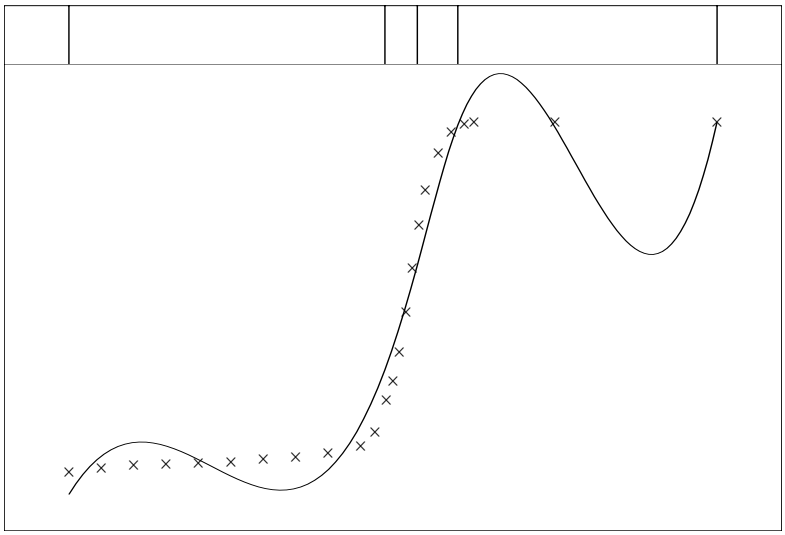

Figure 1: Data and initial knot set.

Thus, in Figure 1, the curve fits the last two data points quite well compared with the fit elsewhere, so no knot should have been added in this region. In Figure 2, the curve goes exactly through the last two points, so a further knot is certainly not needed here.

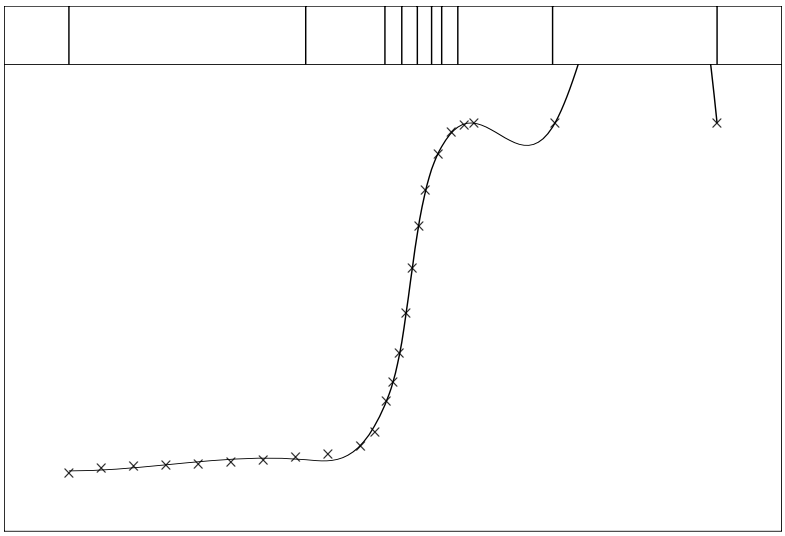

Figure 2: Five additional knots.

Figure 4 shows what can be achieved without the extra knot on each of the flat regions. Remembering that within each knot interval the spline is a cubic polynomial, there is really no need to have more than one knot interval covering each flat region.

What we have, in fact, in Figures 2 and 3 is a case of too many knots (so too many coefficients in the spline equation) for the number of data points. As a consequence the fit is too close to the data, tending to have unwanted fluctuations between the data points.

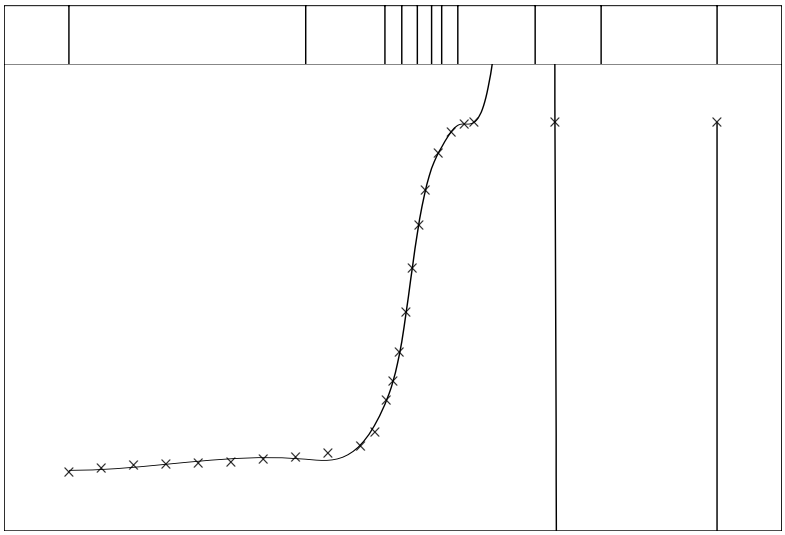

Figure 3: Disastrous addition of a further knot.

This problem is only local, in the sense that, in localities where there are plenty of data points, there can be a lot of knots, as long as there are few knots where there are few points, especially near the ends of the interval. In the present example, with so few data points on the right, just the one extra knot in Figure 2 is too many! The signs are clearly present, with the last two points fitted exactly (at least to the graphical accuracy and actually much closer than that) and fluctuations *within* the last two knot-intervals (compare Figure 1, where only the final point is fitted exactly and one of the wobbles spans several data points).

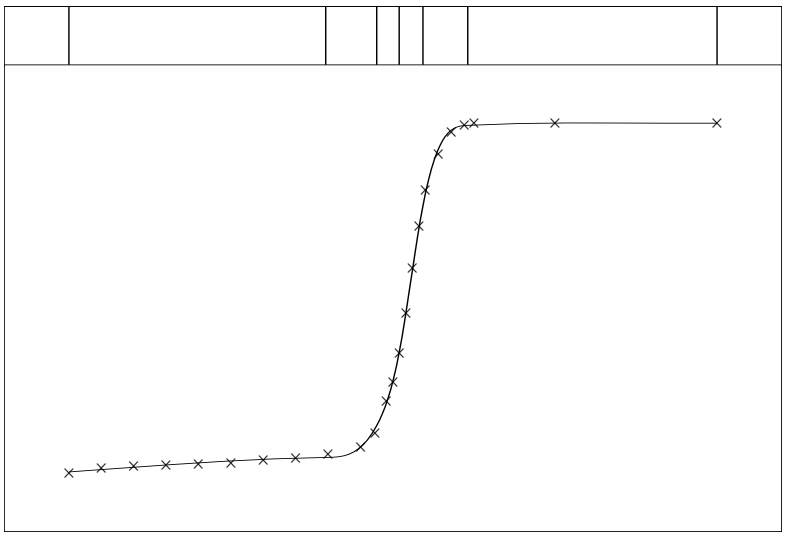

Figure 4: A satisfactory approximant.

The situation in Figure 3 is different. The fit, if computed exactly, *would* still pass through the last two data points, with even more violent fluctuations. However, the problem has become so ill conditioned that all accuracy has been lost. Indeed, if the last interior knot were moved a tiny amount to the right, there would be no unique solution and an error message would have been caused. *Near singularity* is, sadly, not picked up by the procedure, but can be spotted readily in a graph, as in Figure 3. B-spline coefficients becoming large, with alternating signs, is another indication; these may be inspected after a call to `nag_spline_1d_extract`. However, it is better to avoid such situations, firstly by providing, whenever possible, data adequately covering the range of interest, and secondly by placing knots only where there is a reasonable amount of data.

The example here could, in fact, have utilised from the start the observation made in the second paragraph of this section, that three close knots can produce a rapid change in slope. The example has two such rapid changes and so requires two sets of three close knots (in fact, the two sets can be so close that one knot can serve in both sets, so only five knots prove sufficient in Figure 4). It should be noted, however, that the rapid turn occurs within the range spanned by the three knots. This is the reason that the six knots in Figure 2 are not satisfactory as they do not quite span the two turns.

Some more examples to illustrate the choice of knots are given in Cox and Hayes [7].

References

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

[2] Cox M G (1974) A data-fitting package for the non-specialist user *Software for Numerical Mathematics* (ed D J Evans) Academic Press

[3] Cox M G (1975) An algorithm for spline interpolation *J. Inst. Math. Appl.* **15** 95–108

[4] Cox M G (1975) Numerical methods for the interpolation and approximation of data by spline functions *PhD Thesis* City University, London

[5] Cox M G (1977) A survey of numerical methods for data and function approximation *The State of the Art in Numerical Analysis* (ed D A H Jacobs) Academic Press 627–668

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

[7] Cox M G and Hayes J G (1973) Curve fitting: A guide and suite of algorithms for the non-specialist user *NPL Report NAC 26* National Physical Laboratory

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

[9] Dierckx P (1975) An algorithm for smoothing, differentiating and integration of experimental data using spline functions *J. Comput. Appl. Math.* **1** 165–184

[10] Dierckx P (1981) An improved algorithm for curve fitting with spline functions *Report TW54* Department of Computer Science, Katholieke Universiteit Leuven

[11] Dierckx P (1982) A fast algorithm for smoothing data on a rectangular grid while using spline functions *SIAM J. Numer. Anal.* **19** 1286–1304

[12] Dierckx P (1993) *Curve and Surface Fitting with Splines* Clarendon Press, Oxford

[13] Gentleman W M (1969) An error analysis of Goertzel's (Watt's) method for computing Fourier coefficients *Comput. J.* **12** 160–165

[14] Gentleman W M (1973) Least-squares computations by Givens transformations without square roots *J. Inst. Math. Applic.* **12** 329–336

[15] Hayes J G (1974) Numerical methods for curve and surface fitting *Bull. Inst. Math. Appl.* **10** 144–152

[16] Reinsch C H (1967) Smoothing by spline functions *Numer. Math.* **10** 177–183

[17] Schoenberg I J and Whitney A (1953) On Polya frequency functions III *Trans. Amer. Math. Soc.* **74** 246–259