1

Introduction

The principal objective of this book is to present the method of lines (MOL) numerical integration of partial differential equations (PDEs), with spline collocation (SC) approximation of the PDE boundary-value derivatives. This approach is therefore termed *spline collocation method of lines* (SCMOL).

The details of SCMOL computer implementation are presented in terms of a series of applications, first for one-dimensional (1D) PDEs, then for two-dimensional (2D) PDEs, and finally for a series of legacy PDEs to illustrate the broad applicability of SCMOL. The approach is not with formal mathematics, for example, theorems and proofs, but rather, by examples of SCMOL discussed in detail, including routines in R.[1]

In this introduction, some basic properties of splines are reviewed, including example applications of the utilities (functions) for splines in R. The R spline utilities are then applied to PDEs in the following chapters within the SCMOL setting.

Splines are polynomials that can be used for the functional approximation of a set of numerical pairs

Table 1.1 Data pairs for spline interpolation.

x_1	y_1
x_2	y_2
$\vdots$	$\vdots$
x_{n-1}	y_{n-1}
x_n	y_n

1 R is a quality open-source scientific programming system that can be easily downloaded from the Internet (http://www.R-project.org/). In particular, R has (i) vector–matrix operations that facilitate the programming of linear algebra, (ii) a library of quality ordinary differential equation (ODE) integrators, and (iii) graphical utilities for the presentation numerical ODE/PDE solutions. All of these features and utilities are demonstrated through the applications in this book.

Spline Collocation Methods for Partial Differential Equations: With Applications in R, First Edition.
William E. Schiesser.

Companion website: www.wiley.com/go/Spline_Collocation

A cubic spline is of the form

$$p_3(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \tag{1.1a}$$

The coefficients a_0, a_1, a_2, a_3 are evaluated to (i) return the original data of Table 1.1 and (ii) provide continuity in the computed first and second derivatives of $p_3(x)$ at $x_1, x_2, \ldots, x_{n-1}, x_n$. The sequence in x does not have to be uniformly spaced, which is a particularly useful feature in the SCMOL solution of PDEs, that is, the points can be placed as required to achieve good resolution in x.

$p_3(x)$ can be differentiated to give the first and second derivatives

$$\frac{dp_3(x)}{dx} = a_1 + 2a_2 x + 3a_3 x^2 \tag{1.1b}$$

$$\frac{d^2 p_3(x)}{dx^2} = 2a_2 + 6a_3 x \tag{1.1c}$$

$$\frac{d^3 p_3(x)}{dx^3} = 6a_3 \tag{1.1d}$$

The derivatives of Eqs. (1.1a)–(1.1d) can then be used to approximate first, second, and third derivatives in PDEs.

All of the operations reflected in Eqs. (1.1) are implemented in the R function `splinefun`,[2] which returns the coefficients a_0, a_1, a_2, a_3 for a set of n data as listed in Table 1.1. a_0, a_1, a_2, a_3 are computed by the solution of a tridiagonal algebraic system of $n - 1$ equations. This set of $n + 1$ coefficients therefore requires the specification of two additional conditions.[3]

A variety of additional conditions can be specified. For example, if the two second derivatives at the end points of the data set are set to zero, so-called *natural cubic splines* result. The details of the splines resulting from various sets of two additional conditions as implemented in `splinefun` are given in Appendix A1 and in [1].

The use of `splinefun` is illustrated with a series of examples that follows.

1.1 Uniform Grids

Application of `splinefun` to the function $y = \sin(\pi x)$ is illustrated with the following code:

```
#
# Previous workspaces are cleared
```

2 Details of `splinefun`, including the programming options and example applications, are available from the online documentation accessed by `help(splinefun)` entered at the R prompt. Excerpts from this documentation are given in Appendix A1.

3 This gives a set of $n + 1$ equations in the $n + 1$ unknown coefficients.

```
  rm(list=ls(all=TRUE))
#
# Define uniform grid
  xl=0;xu=1;n=11;
  x=seq(from=xl,to=xu,by=(xu-xl)/(n-1));
#
# Define function to be approximated
  u=sin(pi*x);
#
# Set up spline table
  utable=splinefun(x,u);
#
# Compute spline approximation
  us=utable(x);
#
# Display comparison of function and its spline
# approximation
  cat(sprintf("\n    x          u        us
              diff"));
  for(i in 1:n){
    cat(sprintf("\n%5.2f%10.5f%10.5f%12.7f",
      x[i],u[i],us[i],us[i]-u[i]));
  }
```

Listing 1.1 Spline approximation of $y = \sin(\pi x)$.

We can note the following details about this listing:

- Previous workspaces are cleared to avoid the unintended use of out-of-date files.

```
#
# Previous workspaces are cleared
  rm(list=ls(all=TRUE))
```

- A uniform grid is defined with 11 points, $0 \leq x \leq 1$ so that $x = 0, 0.1, \ldots, 1$.

```
#
# Define uniform grid
  xl=0;xu=1;n=11;
  x=seq(from=xl,to=xu,by=(xu-xl)/(n-1));
```

  The `seq` utility is used for this purpose.
- The function $y = \sin(\pi x)$ is defined on the grid in x.

```
#
```

```
# Define function to be approximated
  u=sin(pi*x);
```

This definition of u[4] illustrates the vectorization available in R, that is, u is an n-vector since x is an n-vector. Also, the function sin can operate on a vector to produce a vector.

- splinefun is used to define a table of spline coefficients, for example, a_0, a_1, a_2, a_3 in Eq. (1.1a).

```
#
# Set up spline table
  utable=splinefun(x,u);
```

splinefun is part of the basic R and does not have to be accessed from an external library. Here the default spline mmf is used in which an exact cubic polynomial is fitted to the four points at each end of the data of Table 1.1 [1], p. 73.

- The table utable is used to compute spline approximations to u at the values of x in x.

```
#
# Compute spline approximation
  us=utable(x);
```

Other values of x within the interval xl to xu could be used for interpolation between the grid points.

- The values of u, their spline approximations us, and the difference between the two are displayed.

```
#
# Display comparison of function and its spline
# approximation
  cat(sprintf("\n    x        u       us
              diff"));
  for(i in 1:n){
    cat(sprintf("\n%5.2f%10.5f%10.5f%12.7f",
      x[i],u[i],us[i],us[i]-u[i]));
  }
```

The use of the combination cat(sprintf()) provides detailed formatting of the output.

Execution of the code in Listing 1.1 gives the following output:

4 The function $y = \sin(\pi x)$ is named u in this and subsequent code since it will ultimately be the dependent variable of a PDE. That is, in accordance with the usual convention in the literature, the dependent variables of PDEs are designated with u.

Table 1.2 Output from Listing 1.1.

x	u	us	diff
0.00	0.00000	0.00000	0.0000000
0.10	0.30902	0.30902	0.0000000
0.20	0.58779	0.58779	0.0000000
0.30	0.80902	0.80902	0.0000000
0.40	0.95106	0.95106	0.0000000
0.50	1.00000	1.00000	0.0000000
0.60	0.95106	0.95106	0.0000000
0.70	0.80902	0.80902	0.0000000
0.80	0.58779	0.58779	0.0000000
0.90	0.30902	0.30902	0.0000000
1.00	0.00000	0.00000	0.0000000

This output demonstrates that the spline in `splinefun` returns the original data (an important feature of splines in general).

Equations (1.1a)–(1.1d) can also give numerical approximations to the first to third derivatives, as demonstrated by the following routine:

```
#
# Previous workspaces are cleared
  rm(list=ls(all=TRUE))
#
# Define uniform grid
  xl=0;xu=1;n=11;
  x=seq(from=xl,to=xu,by=(xu-xl)/(n-1));
#
# Define function to be approximated, and its
# derivatives
     u=sin(pi*x);
    ux=pi*cos(pi*x);
   uxx=-pi^2*sin(pi*x);
  uxxx=-pi^3*cos(pi*x);
#
# Set up spline table for function, derivatives
  utable=splinefun(x,u);
#
# Compute spline approximation of function
# derivatives
      us=utable(x);
     usx=utable(x,deriv=1);
```

```
    usxx=utable(x,deriv=2);
   usxxx=utable(x,deriv=3);
#
# Display comparison of function and its spline
# approximation, and its derivatives
#
# u
  cat(sprintf("\n    x         u        us
              diff"));
  for(i in 1:n){
    cat(sprintf("\n%5.2f%10.5f%10.5f%12.7f",
      x[i],u[i],us[i],us[i]-u[i]));
  }
#
# ux
  cat(sprintf("\n    x        ux       usx
              diff"));
  for(i in 1:n){
    cat(sprintf("\n%5.2f%10.5f%10.5f%12.7f",
      x[i],ux[i],usx[i],usx[i]-ux[i]));
  }
#
# uxx
  cat(sprintf("\n    x       uxx      usxx
              diff"));
  for(i in 1:n){
    cat(sprintf("\n%5.2f%10.5f%10.5f%12.7f",
      x[i],uxx[i],usxx[i],usxx[i]-uxx[i]));
  }
#
# uxxx
  cat(sprintf("\n    x      uxxx     usxxx
              diff"));
  for(i in 1:n){
    cat(sprintf("\n%5.2f%10.5f%10.5f%12.7f",
      x[i],uxxx[i],usxxx[i],usxxx[i]-uxxx[i]));
  }
```

Listing 1.2 Spline approximation of $y = \sin(\pi x)$ and the first to third derivatives.

We can note the following details about Listing 1.2:

- Previous workspaces are cleared and the same uniform grid in x as in Listing 1.1 is defined.

- The function $y = \sin(\pi x)$ and its first three derivatives are computed. Again, `u, ux, uxx, uxxx` are n-vectors.

```
#
# Define function to be approximated, and its
# derivatives
     u=sin(pi*x);
    ux=pi*cos(pi*x);
   uxx=-pi^2*sin(pi*x);
  uxxx=-pi^3*cos(pi*x);
```

  The derivatives are used to evaluate the numerical approximations from `splinefun`.
- The table of spline coefficients is defined as in Listing 1.1.

```
#
# Set up spline table for function, derivatives
  utable=splinefun(x,u);
```

- The function `us` and its first three derivatives are computed from `utable`.

```
#
# Compute spline approximation of function
# derivatives
      us=utable(x);
     usx=utable(x,deriv=1);
    usxx=utable(x,deriv=2);
   usxxx=utable(x,deriv=3);
```

  `us, usx, usxx, usxxx` are n-vectors. The argument `deriv` specifies the order of the derivative to be computed.
- The exact values of $y = \sin(\pi x)$ and the numerical approximations are compared and displayed.

```
#
# Display comparison of function and its spline
# approximation, and its derivatives
#
# u
  cat(sprintf("\n    x          u         us
              diff"));
  for(i in 1:n){
    cat(sprintf("\n%5.2f%10.5f%10.5f%12.7f",
      x[i],u[i],us[i],us[i]-u[i]));
  }
#
```

```
# ux
  cat(sprintf("\n    x         ux        usx
              diff"));
  for(i in 1:n){
    cat(sprintf("\n%5.2f%10.5f%10.5f%12.7f",
      x[i],ux[i],usx[i],usx[i]-ux[i]));
  }
#
# uxx
  cat(sprintf("\n    x        uxx       usxx
              diff"));
  for(i in 1:n){
    cat(sprintf("\n%5.2f%10.5f%10.5f%12.7f",
      x[i],uxx[i],usxx[i],usxx[i]-uxx[i]));
  }
#
# uxxx
  cat(sprintf("\n    x       uxxx      usxxx
              diff"));
  for(i in 1:n){
    cat(sprintf("\n%5.2f%10.5f%10.5f%12.7f",
      x[i],uxxx[i],usxxx[i],usxxx[i]-uxxx[i]));
  }
```

The output from Listing 1.2 is in Table 1.3.

Table 1.3 Output from Listing 1.2.

x	u	us	diff
0.00	0.00000	0.00000	0.0000000
0.10	0.30902	0.30902	0.0000000
0.20	0.58779	0.58779	0.0000000
0.30	0.80902	0.80902	0.0000000
0.40	0.95106	0.95106	0.0000000
0.50	1.00000	1.00000	0.0000000
0.60	0.95106	0.95106	0.0000000
0.70	0.80902	0.80902	0.0000000
0.80	0.58779	0.58779	0.0000000
0.90	0.30902	0.30902	0.0000000
1.00	0.00000	0.00000	0.0000000

Table 1.3 (Continued)

x	ux	usx	diff
0.00	3.14159	3.14930	0.0077113
0.10	2.98783	2.98556	-0.0022759
0.20	2.54160	2.54203	0.0004266
0.30	1.84658	1.84633	-0.0002520
0.40	0.97081	0.97079	-0.0000154
0.50	0.00000	0.00000	0.0000000
0.60	-0.97081	-0.97079	0.0000154
0.70	-1.84658	-1.84633	0.0002520
0.80	-2.54160	-2.54203	-0.0004266
0.90	-2.98783	-2.98556	0.0022759
1.00	-3.14159	-3.14930	-0.0077113

x	uxx	usxx	diff
0.00	-0.00000	-0.27309	-0.2730880
0.10	-3.04988	-3.00187	0.0480096
0.20	-5.80121	-5.86869	-0.0674824
0.30	-7.98468	-8.04528	-0.0606045
0.40	-9.38655	-9.46551	-0.0789615
0.50	-9.86960	-9.95029	-0.0806842
0.60	-9.38655	-9.46551	-0.0789615
0.70	-7.98468	-8.04528	-0.0606045
0.80	-5.80121	-5.86869	-0.0674824
0.90	-3.04988	-3.00187	0.0480096
1.00	-0.00000	-0.27309	-0.2730880

x	uxxx	usxxx	diff
0.00	-31.00628	-27.28778	3.7184973
0.10	-29.48872	-27.28778	2.2009421
0.20	-25.08460	-21.76592	3.3186866
0.30	-18.22503	-21.76592	-3.5408860
0.40	-9.58147	-4.84776	4.7337109
0.50	-0.00000	-4.84776	-4.8477555
0.60	9.58147	14.20231	4.6208422
0.70	18.22503	14.20231	-4.0227235
0.80	25.08460	28.66824	3.5836399
0.90	29.48872	28.66824	-0.8204768
1.00	31.00628	27.28778	-3.7184973

We can note the following details about this output:

- The output for the function is repeated as in Table 1.2.
- As expected, the computed derivatives are not exact, and the errors increase with the order of the derivative. The increasing errors are expected when considering Eqs. (1.1a)–(1.1d) (the approximations vary from a second-order quadratic of Eq. (1.1a) to a constant of Eq. (1.1d)).
  The errors tend to be largest at the end points in x, for example, ± 0.0077113 for `ux`, -0.2730880 for `uxx`. The exception is for the errors in `uxxx`, but since the approximation is the constant $6a_3$ of Eq. (1.1d), the error is large, even within the interval. In other words, the use of the third derivative in the PDE applications to follow should be avoided.

One method to reduce the errors in Table 1.3 would be to use a smaller grid spacing (more points in the grid). This is accomplished by changing `n=11` in Listing 1.2 to, for example, `n=21` (everything else in the code remains unchanged). The numerical output follows.

Table 1.4 Output from Listing 1.2, `n=21`.

x	u	us	diff
0.00	0.00000	0.00000	0.0000000
0.05	0.15643	0.15643	0.0000000
0.10	0.30902	0.30902	0.0000000
0.15	0.45399	0.45399	0.0000000
0.20	0.58779	0.58779	0.0000000
0.25	0.70711	0.70711	0.0000000
0.30	0.80902	0.80902	0.0000000
0.35	0.89101	0.89101	0.0000000
0.40	0.95106	0.95106	0.0000000
0.45	0.98769	0.98769	0.0000000
0.50	1.00000	1.00000	0.0000000
0.55	0.98769	0.98769	0.0000000
0.60	0.95106	0.95106	0.0000000
0.65	0.89101	0.89101	0.0000000
0.70	0.80902	0.80902	0.0000000
0.75	0.70711	0.70711	0.0000000
0.80	0.58779	0.58779	0.0000000
0.85	0.45399	0.45399	0.0000000
0.90	0.30902	0.30902	0.0000000

Table 1.4 (Continued)

```
0.95   0.15643   0.15643   0.0000000
1.00   0.00000   0.00000   0.0000000

   x        ux        usx         diff
0.00   3.14159   3.14209   0.0004935
0.05   3.10291   3.10277  -0.0001456
0.10   2.98783   2.98786   0.0000261
0.15   2.79918   2.79916  -0.0000192
0.20   2.54160   2.54160  -0.0000060
0.25   2.22144   2.22143  -0.0000082
0.30   1.84658   1.84658  -0.0000061
0.35   1.42625   1.42625  -0.0000049
0.40   0.97081   0.97080  -0.0000033
0.45   0.49145   0.49145  -0.0000017
0.50   0.00000  -0.00000  -0.0000000
0.55  -0.49145  -0.49145   0.0000017
0.60  -0.97081  -0.97080   0.0000033
0.65  -1.42625  -1.42625   0.0000049
0.70  -1.84658  -1.84658   0.0000061
0.75  -2.22144  -2.22143   0.0000082
0.80  -2.54160  -2.54160   0.0000060
0.85  -2.79918  -2.79916   0.0000192
0.90  -2.98783  -2.98786  -0.0000261
0.95  -3.10291  -3.10277   0.0001456
1.00  -3.14159  -3.14209  -0.0004935

   x       uxx       usxx         diff
0.00  -0.00000  -0.03493  -0.0349290
0.05  -1.54395  -1.53776   0.0061820
0.10  -3.04988  -3.05866  -0.0087840
0.15  -4.48071  -4.48926  -0.0085487
0.20  -5.80121  -5.81333  -0.0121181
0.25  -6.97886  -6.99318  -0.0143132
0.30  -7.98468  -8.00112  -0.0164442
0.35  -8.79388  -8.81197  -0.0180931
0.40  -9.38655  -9.40587  -0.0193171
0.45  -9.74809  -9.76815  -0.0200599
```

(Continued)

Table 1.4 (Continued)

0.50	-9.86960	-9.88991	-0.0203103
0.55	-9.74809	-9.76815	-0.0200599
0.60	-9.38655	-9.40587	-0.0193171
0.65	-8.79388	-8.81197	-0.0180931
0.70	-7.98468	-8.00112	-0.0164442
0.75	-6.97886	-6.99318	-0.0143132
0.80	-5.80121	-5.81333	-0.0121181
0.85	-4.48071	-4.48926	-0.0085487
0.90	-3.04988	-3.05866	-0.0087840
0.95	-1.54395	-1.53776	0.0061820
1.00	-0.00000	-0.03493	-0.0349290

x	xuxx	usxxx	diff
0.00	-31.00628	-30.05671	0.9495706
0.05	-30.62454	-30.05671	0.5678318
0.10	-29.48872	-28.61192	0.8768047
0.15	-27.62679	-28.61192	-0.9851220
0.20	-25.08460	-23.59703	1.4875760
0.25	-21.92475	-23.59703	-1.6722802
0.30	-18.22503	-16.21706	2.0079710
0.35	-14.07656	-16.21706	-2.1405061
0.40	-9.58147	-7.24569	2.3357781
0.45	-4.85045	-7.24569	-2.3952380
0.50	-0.00000	2.43523	2.4352332
0.55	4.85045	2.43523	-2.4152171
0.60	9.58147	11.87787	2.2964074
0.65	14.07656	11.87787	-2.1986812
0.70	18.22503	20.15889	1.9338574
0.75	21.92475	20.15889	-1.7658589
0.80	25.08460	26.48141	1.3968090
0.85	27.62679	26.48141	-1.1453810
0.90	29.48872	30.41790	0.9291820
0.95	30.62454	30.41790	-0.2066345
1.00	31.00628	30.05671	-0.9495706

We can conclude generally that the errors in Table 1.4 have been reduced with the change `n=11` to `n=21`. This suggests the question of how the errors vary with `n`. This question can be addressed empirically (from the observed numerical output for increasing `n`).

For `n=11,21,31,41,51`, selected errors are summarized in Table 1.5.

Table 1.5 Summary of errors for `n=11,21,31,41,51`.

derivative	x	n	error
usx	0	11	0.0077113
		21	0.0004935
		31	0.0000979
		41	0.0000310
		51	0.0000127
	0.5	11	-0.0000000
	0.5	21	-0.0000000
	0.5	31	0.0000000
	0.5	41	0.0000000
	0.5	51	0.0000000
usxx	0	11	-0.2730880
		21	-0.0349290
		31	-0.0103934
		41	-0.0043912
		51	-0.0022499
	0.5	11	-0.0806842
	0.5	21	-0.0203103
	0.5	31	-0.0090227
	0.5	41	-0.0050744
	0.5	51	-0.0032474

We can note the following details about Table 1.5:

- Quantitative relationships between the error and the number of points `n` are not apparent. For example, an order relationship such as error $= O(\Delta x^p)$, where Δx is the grid spacing, does not apply for `ux, x=0.5` (the error does not vary with grid spacing).
- Generally, some form of convergence appears to occur in the sense that the errors decrease with increasing `n`.
- The accuracy decreases with successive differentiation, for example, the errors for `usxx` are greater than for `usx`.

- The errors at the boundaries, for example, `x = 0`, are larger than at the interior points, for example, `x = 0.5`.

These results, although specific to the function $y = \sin(\pi x)$, suggest that some experimentation with the grid interval (value of `n`) can be used to infer accuracy of the spline approximation, even when exact values are not available (the usual case in PDE applications). For example, for `x=0.5, n=11, usxx=-9.95029` (Table 1.3) and for `x=0.5, n=21, usxx=-9.88991` (Table 1.4) suggests an accuracy of `-9.95029 -(-9.88991)= -0.06038`, which can be inferred merely by changing n and comparing numerical results.[5] Exact values of the function and its derivatives and the associated exact errors are not required for an error analysis (and are generally unavailable anyway). Rather, the errors can be *estimated* from approximate results.

To summarize, the preceding results indicate the calculation of derivatives of tabulated data pairs using `splinefun` is straightforward. This is further demonstrated next with the test function $y = e^{ax}$.

The routine for this case is the same as in Listing 1.2 except that

```
#
# Define function to be approximated, and its
# derivatives
     a=1;
     u=exp(a*x);
    ux=a*exp(a*x);
   uxx=a^2*exp(a*x);
  uxxx=a^3*exp(a*x);
```

is used in place of

```
#
# Define function to be approximated, and its
# derivatives
     u=sin(pi*x);
    ux=pi*cos(pi*x);
   uxx=-pi^2*sin(pi*x);
  uxxx=-pi^3*cos(pi*x);
```

The rate of change of $y = e^{ax}$ is defined by the parameter a, which can be varied to test the spline approximations as discussed next.

The output is in Table 1.6.

Since the function and its derivatives are the same (for $a = 1$), the reduction in the accuracy of successive derivatives is clear from Table 1.6. Also, the error

5 This form of error analysis is termed *h refinement* since the grid spacing in the numerical analysis literature is frequently designated as *h*.

Table 1.6 Output for the function $y = e^{ax}$, $a = 1$.

x	u	us	diff
0.00	1.00000	1.00000	0.0000000
0.10	1.10517	1.10517	0.0000000
0.20	1.22140	1.22140	0.0000000
0.30	1.34986	1.34986	0.0000000
0.40	1.49182	1.49182	0.0000000
0.50	1.64872	1.64872	0.0000000
0.60	1.82212	1.82212	0.0000000
0.70	2.01375	2.01375	0.0000000
0.80	2.22554	2.22554	0.0000000
0.90	2.45960	2.45960	0.0000000
1.00	2.71828	2.71828	0.0000000

x	ux	usx	diff
0.00	1.00000	1.00026	0.0002555
0.10	1.10517	1.10510	-0.0000692
0.20	1.22140	1.22142	0.0000177
0.30	1.34986	1.34985	-0.0000056
0.40	1.49182	1.49182	0.0000003
0.50	1.64872	1.64872	-0.0000005
0.60	1.82212	1.82212	-0.0000038
0.70	2.01375	2.01376	0.0000095
0.80	2.22554	2.22550	-0.0000408
0.90	2.45960	2.45975	0.0001463
1.00	2.71828	2.71773	-0.0005526

x	uxx	usxx	diff
0.00	1.00000	0.99030	-0.0097019
0.10	1.10517	1.10663	0.0014559
0.20	1.22140	1.21975	-0.0016548
0.30	1.34986	1.34891	-0.0009520
0.40	1.49182	1.49053	-0.0012956
0.50	1.64872	1.64739	-0.0013349
0.60	1.82212	1.82050	-0.0016196
0.70	2.01375	2.01244	-0.0013094
0.80	2.22554	2.22232	-0.0032249
0.90	2.45960	2.46267	0.0030664
1.00	2.71828	2.69693	-0.0213550

(Continued)

Table 1.6 (Continued)

x	uxxx	usxxx	diff
0.00	1.00000	1.16329	0.1632873
0.10	1.10517	1.16329	0.0581164
0.20	1.22140	1.29159	0.0701854
0.30	1.34986	1.29159	-0.0582707
0.40	1.49182	1.56857	0.0767480
0.50	1.64872	1.56857	-0.0801486
0.60	1.82212	1.91944	0.0973222
0.70	2.01375	1.91944	-0.0943117
0.80	2.22554	2.40353	0.1779934
0.90	2.45960	2.40353	-0.0560688
1.00	2.71828	2.34257	-0.3757088

in the computed derivatives is greater at the boundaries $x = 0, 1$ than at the interior points, as before with $\sin(\pi x)$. Since the function changes more rapidly at $x = 1$ than at $x = 0$, the errors are greater at $x = 1$.

The last point of greater variation at $x = 1$ can be demonstrated by using a larger value of a. For $a = 2$, the numerical output is

Table 1.7 Output for the function $y = e^{ax}$, $a = 2$.

x	u	us	diff
0.00	1.00000	1.00000	0.0000000
0.10	1.22140	1.22140	0.0000000
0.20	1.49182	1.49182	0.0000000
0.30	1.82212	1.82212	0.0000000
0.40	2.22554	2.22554	0.0000000
0.50	2.71828	2.71828	0.0000000
0.60	3.32012	3.32012	0.0000000
0.70	4.05520	4.05520	0.0000000
0.80	4.95303	4.95303	0.0000000
0.90	6.04965	6.04965	0.0000000
1.00	7.38906	7.38906	0.0000000

Table 1.7 (Continued)

x	ux	usx	diff
0.00	2.00000	2.00460	0.0045955
0.10	2.44281	2.44155	-0.0012576
0.20	2.98365	2.98395	0.0003043
0.30	3.64424	3.64412	-0.0001189
0.40	4.45108	4.45106	-0.0000235
0.50	5.43656	5.43654	-0.0000249
0.60	6.64023	6.64007	-0.0001672
0.70	8.11040	8.11074	0.0003389
0.80	9.90606	9.90444	-0.0016219
0.90	12.09929	12.10491	0.0056194
1.00	14.77811	14.75661	-0.0215021

x	uxx	usxx	diff
0.00	4.00000	3.82687	-0.1731250
0.10	4.88561	4.91217	0.0265622
0.20	5.96730	5.93594	-0.0313577
0.30	7.28848	7.26736	-0.0211146
0.40	8.90216	8.87143	-0.0307319
0.50	10.87313	10.83818	-0.0349520
0.60	13.28047	13.23238	-0.0480843
0.70	16.22080	16.18106	-0.0397388
0.80	19.81213	19.69302	-0.1191094
0.90	24.19859	24.31641	0.1178172
1.00	29.55622	28.71751	-0.8387162

x	uxxx	usxxx	diff
0.00	8.00000	10.85298	2.8529819
0.10	9.77122	10.85298	1.0817599
0.20	11.93460	13.31420	1.3795979
0.30	14.57695	13.31420	-1.2627549
0.40	17.80433	19.66744	1.8631082
0.50	21.74625	19.66744	-2.0788190
0.60	26.56094	29.48678	2.9258419
0.70	32.44160	29.48678	-2.9548224
0.80	39.62426	46.23387	6.6096075
0.90	48.39718	46.23387	-2.1633128
1.00	59.11245	44.01101	-15.1014368

The larger variation in e^{ax} and the resulting larger errors are clear in Table 1.7 when compared with Table 1.6. Again, these errors could be reduced by using a larger value of n. However, a reduction in the errors could also possibly be accomplished by concentrating the grid points near $x = 1$. That is, we could take advantage of the basic feature of splines that the grid spacing can be variable.

1.2 Variable Grids

To investigate this alternative, we consider first an example of how a variable grid can be produced. This is illustrated by the following code:

```
n=11;x=rep(0,n);
x[1]=0;xin=1/(n-1);
cat(sprintf("\n x[1] = %6.4f  xin = %6.4f",
            x[1],xin));
for(i in 2:n){
  x[i]=x[i-1]+xin/i;
  cat(sprintf("\n i = %2d  x = %6.4f  dx = %6.4f",
              i,x[i],x[i]-x[i-1]));
}
```

Listing 1.3 Generation of a variable grid.

We can note the following details of Listing 1.3:

- A grid of `n=11` points is declared with the `rep` utility.

```
n=11;x=rep(0,n);
```

- The first (leftmost) point is defined. Then an increment is computed.

```
x[1]=0;xin=1/(n-1);
cat(sprintf("\n x[1] = %6.4f  xin = %6.4f",
            x[1],xin));
```

- A `for` is used to step through points `i = 2,3,...,n`.

```
for(i in 2:n){
   x[i]=x[i-1]+xin/i;
   cat(sprintf("\n i = %2d  x = %6.4f
               dx = %6.4f",i,x[i],x[i]-
               x[i-1]));
 }
```

A particular value of the grid point, `x[i]`, is computed from the previous point, `u[i-1]`, plus an increment, `xin/i`. This increment decreases with increasing `i`, thereby giving a grid with a decreasing spacing.

The output from this code follows (Table 1.8).

Table 1.8 Output from Listing 1.3.

```
x[1] = 0.0000  xin = 0.1000

i =  2  x = 0.0500  dx = 0.0500
i =  3  x = 0.0833  dx = 0.0333
i =  4  x = 0.1083  dx = 0.0250
i =  5  x = 0.1283  dx = 0.0200
i =  6  x = 0.1450  dx = 0.0167
i =  7  x = 0.1593  dx = 0.0143
i =  8  x = 0.1718  dx = 0.0125
i =  9  x = 0.1829  dx = 0.0111
i = 10  x = 0.1929  dx = 0.0100
i = 11  x = 0.2020  dx = 0.0091
```

We can note the following details about this output:

- The grid spacing varies from `0.1000` at `x = 0` to `0.0091` at x = 0.2020.
- The interval in x is not correct since it should be $0 \leq x \leq 1$. This can be corrected by using a normalizing factor as

```
x[1]=0;xin=(1/0.2020)/(n-1);
                .
                .
                .
x[i]=x[i-1]+xin/i;
```

in Listing 1.3. The resulting output is in Table 1.9.
The grid spacing is again variable (by more than a factor of 1/10, `0.4950` to `0.0450`) and the interval in x is $0 \leq x \leq 1$.
This discussion is presented in some detail to illustrate how a variable grid might be generated. The approach can be generalized as

```
x[1]=0;xin=c/(n-1);
                .
                .
                .
x[i]=x[i-1]+xin/(i^p);
```

where `c` is a normalizing constant (e.g., `1/0.2020`) and `p` is a power of the grid index `i`. For `p < 1`, the grid spacing varies more slowly with x than for

Table 1.9 Output from Listing 1.3 with $0 \leq x \leq 1$.

```
x[1] = 0.0000  xin = 0.4950

i =  2  x = 0.2475  dx = 0.2475
i =  3  x = 0.4125  dx = 0.1650
i =  4  x = 0.5363  dx = 0.1238
i =  5  x = 0.6353  dx = 0.0990
i =  6  x = 0.7178  dx = 0.0825
i =  7  x = 0.7885  dx = 0.0707
i =  8  x = 0.8504  dx = 0.0619
i =  9  x = 0.9054  dx = 0.0550
i = 10  x = 0.9549  dx = 0.0495
i = 11  x = 0.9999  dx = 0.0450
```

`p = 1`, and for `p > 1`, it varies more rapidly. An important detail is that the grid spacing varies smoothly with x (as in Table 1.9), since an abrupt change in the spacing can cause problems with the spline approximation.

The variable grid in Table 1.9 is now used as an input to `splinefun`. This requires only a change in the grid definition code in Listing 1.2 (with the use of e^{ax}).

```
#
# Previous workspaces are cleared
  rm(list=ls(all=TRUE))
#
# Define grid
  xl=0;xu=1;n=11;ng=2;
#
# Uniform grid
  if(ng==1){
    x=seq(from=xl,to=xu,by=(xu-xl)/(n-1));
    for(i in 2:n){
      cat(sprintf("\n i = %2d  x = %6.4f  dx = %6.4f",
                  i,x[i],x[i]-x[i-1]));
    }
  }
#
# Variable grid
  if(ng==2){
```

```
    x=rep(0,n);
    x[1]=0;xin=(1/0.2020)/(n-1);
    cat(sprintf("\n x[1] = %6.4f  xin = %6.4f",
                x[1],xin));
    for(i in 2:n){
      x[i]=x[i-1]+xin/i;
      cat(sprintf("\n i = %2d  x = %6.4f  dx = %6.4f",
                  i,x[i],x[i]-x[i-1]));
    }
  }
#
# Define function to be approximated, and its
# derivatives
     a=2;
     u=exp(a*x);
    ux=a*exp(a*x);
   uxx=a^2*exp(a*x);
  uxxx=a^3*exp(a*x);
#
# Set up spline table for function, derivatives
    utable=splinefun(x,u);
#
# Compute spline approximation of function
# derivatives
      us=utable(x);
     usx=utable(x,deriv=1);
    usxx=utable(x,deriv=2);
   usxxx=utable(x,deriv=3);
#
# Display comparison of function and its spline
# approximation, and its derivatives
#
# u
  cat(sprintf("\n    x         u        us
              diff"));
  for(i in 1:n){
    cat(sprintf("\n%7.4f%10.5f%10.5f%12.7f",
      x[i],u[i],us[i],us[i]-u[i]));
  }
#
# ux
  cat(sprintf("\n    x        ux       usx
              diff"));
```

```
  for(i in 1:n){
    cat(sprintf("\n%7.4f%10.5f%10.5f%12.7f",
      x[i],ux[i],usx[i],usx[i]-ux[i]));
  }
#
# uxx
  cat(sprintf("\n    x        uxx       usxx
              diff"));
  for(i in 1:n){
    cat(sprintf("\n%7.4f%10.5f%10.5f%12.7f",
      x[i],uxx[i],usxx[i],usxx[i]-uxx[i]));
  }
#
# uxxx
  cat(sprintf("\n    x       uxxx      usxxx
              diff"));
  for(i in 1:n){
    cat(sprintf("\n%7.4f%10.5f%10.5f%12.7f",
      x[i],uxxx[i],usxxx[i],usxxx[i]-uxxx[i]));
  }
```

Listing 1.4 Spline approximation of $y = e^{ax}$ and the first to third derivatives, uniform and variable grids.

For `ng=1` a uniform grid is used (the output is in Table 1.7) and for `ng=2` a variable grid is used. The output for `ng=2` follows.

Table 1.10 Output for the function $y = e^{ax}$, $a = 2$, $n = 11$, $ng = 2$.

x	u	us	diff
0.0000	1.00000	1.00000	0.0000000
0.2475	1.64058	1.64058	0.0000000
0.4125	2.28207	2.28207	0.0000000
0.5363	2.92299	2.92299	0.0000000
0.6353	3.56309	3.56309	0.0000000
0.7178	4.20235	4.20235	0.0000000
0.7885	4.84083	4.84083	0.0000000
0.8504	5.47859	5.47859	0.0000000
0.9054	6.11570	6.11570	0.0000000
0.9549	6.75221	6.75221	0.0000000
0.9999	7.38816	7.38816	0.0000000

Table 1.10 (Continued)

x	ux	usx	diff
0.0000	2.00000	2.05688	0.0568850
0.2475	3.28116	3.26995	-0.0112076
0.4125	4.56414	4.56707	0.0029304
0.5363	5.84598	5.84545	-0.0005361
0.6353	7.12617	7.12639	0.0002166
0.7178	8.40470	8.40470	0.0000029
0.7885	9.68166	9.68168	0.0000179
0.8504	10.95719	10.95726	0.0000707
0.9054	12.23141	12.23122	-0.0001861
0.9549	13.50442	13.50513	0.0007133
0.9999	14.77632	14.77382	-0.0025024

x	uxx	usxx	diff
0.0000	4.00000	3.07122	-0.9287777
0.2475	6.56232	6.73035	0.1680358
0.4125	9.12828	8.99071	-0.1375689
0.5363	11.69196	11.66786	-0.0241047
0.6353	14.25234	14.20721	-0.0451373
0.7178	16.80939	16.77906	-0.0303371
0.7885	19.36332	19.33390	-0.0294229
0.8504	21.91438	21.89290	-0.0214782
0.9054	24.46281	24.42823	-0.0345759
0.9549	27.00883	27.03774	0.0289033
0.9999	29.55264	29.34267	-0.2099619

x	uxxx	usxxx	diff
0.0000	8.00000	14.78289	6.7828905
0.2475	13.12464	14.78289	1.6582549
0.4125	18.25655	21.63137	3.3748219
0.5363	23.38392	21.63137	-1.7525503
0.6353	28.50469	31.17082	2.6661338
0.7178	33.61879	31.17082	-2.4479661
0.7885	38.72664	41.35348	2.6268371
0.8504	43.82876	41.35348	-2.4752784
0.9054	48.92562	52.71191	3.7862934
0.9549	54.01766	52.71191	-1.3057497
0.9999	59.10527	51.21573	-7.8895430

A comparison of Tables 1.7 and 1.10 indicates the errors in the spline approximations are reduced for x near the right end $x = 1$, where e^{ax} changes more rapidly but are increased at the left end $x = 0$ due to the larger grid spacing. Thus, some experimentation and evaluation is required when using a variable grid. For example, smaller variations in the grid spacing at both ends could be used to reduce end effects.

The preceding discussion of uniform and variable grid spline differentiation can now be used to develop SCMOL for PDEs. This is done subsequently through example applications.

1.3 Stagewise Differentiation

An alternative approach to computing numerical derivatives is by successive differentiation, termed *stagewise differentiation*. This is illustrated by the following code applied to e^{ax}:

```
#
# Previous workspaces are cleared
  rm(list=ls(all=TRUE))
#
# Define uniform grid
  xl=0;xu=1;n=11;
  x=seq(from=xl,to=xu,by=(xu-xl)/(n-1));
#
# Define function to be approximated, and its
# derivatives
     a=1;
     u=exp(a*x);
   uxx=a^2*exp(a*x);
#
# Set up spline tables for u, ux; compute uxx
# by stagewise differentiation
  utable=splinefun(x,u);
  usx=utable(x,deriv=1);
  uxtable=splinefun(x,usx);
  usxx=uxtable(x,deriv=1);
#
# Display comparison of exact and spline derivatives
#
# uxx
  cat(sprintf("\n    x       uxx      usxx
              diff"));
```

```
for(i in 1:n){
  cat(sprintf("\n%6.4f%10.5f%10.5f%12.7f",
    x[i],uxx[i],usxx[i],usxx[i]-uxx[i]));
}
```

Listing 1.5 Stagewise differentiation of e^{ax}.

We can note the following details about Listing 1.5:

- Previous workspaces are removed and a uniform grid in x with `11` points is defined.

```
#
# Previous workspaces are cleared
  rm(list=ls(all=TRUE))
#
# Define uniform grid
  xl=0;xu=1;n=11;
  x=seq(from=xl,to=xu,by=(xu-xl)/(n-1));
```

- e^{ax} and its second derivative are computed and placed in two n-vectors, `u`, `uxx`.

```
#
# Define function to be approximated, and its
# derivatives
     a=1;
     u=exp(a*x);
   uxx=a^2*exp(a*x);
```

- `splinefun` operates on `u` to produce a spline table, `utable`, which is then used to calculate the first derivative, `usx`. `splinefun` then operates on `usx` to produce a spline table, `uxtable`, which is then used to calculate the second derivative, `usxx`. `usx`.

```
#
# Set up spline tables for u, ux; compute uxx
# by stagewise differentiation
  utable=splinefun(x,u);
  usx=utable(x,deriv=1);
  uxtable=splinefun(x,usx);
  usxx=uxtable(x,deriv=1);
```

Two successive calls to `splinefun` demonstrate the stagewise differentiation (note the use of `deriv=1` for each stage). In principle, the successive differentiation can be continued to calculate higher-order derivatives, but in

practice, the additional error at each stage of the differentiation will eventually give inaccurate derivatives.

- The exact and numerical second derivatives, and their difference, are displayed.

```
#
# Display comparison of exact and spline derivatives
#
# uxx
  cat(sprintf("\n    x       uxx      usxx
              diff"));
  for(i in 1:n){
    cat(sprintf("\n%6.4f%10.5f%10.5f%12.7f",
      x[i],uxx[i],usxx[i],usxx[i]-uxx[i]));
  }
```

The output from Listing 1.5 is in Table 1.11, along with the second derivative by direct differentiation from Table 1.6.

Table 1.11 Output for the function $y = e^{ax}$, $a = 1$, $n = 11$, `usxx` by direct and stagewise differentiation.

```
Stagewise differentiation

     x       uxx      usxx        diff
  0.00   1.00000   0.99301  -0.0069892
  0.10   1.10517   1.10498  -0.0001877
  0.20   1.22140   1.22201   0.0006029
  0.30   1.34986   1.34954  -0.0003202
  0.40   1.49182   1.49198   0.0001514
  0.50   1.64872   1.64858  -0.0001371
  0.60   1.82212   1.82239   0.0002701
  0.70   2.01375   2.01310  -0.0006504
  0.80   2.22554   2.22675   0.0012137
  0.90   2.45960   2.45950  -0.0001065
  1.00   2.71828   2.70213  -0.0161501

Direct differentiation (from Table 1.6)

     x       uxx      usxx        diff
  0.00   1.00000   0.99030  -0.0097019
```

Table 1.11 (Continued)

0.10	1.10517	1.10663	0.0014559
0.20	1.22140	1.21975	-0.0016548
0.30	1.34986	1.34891	-0.0009520
0.40	1.49182	1.49053	-0.0012956
0.50	1.64872	1.64739	-0.0013349
0.60	1.82212	1.82050	-0.0016196
0.70	2.01375	2.01244	-0.0013094
0.80	2.22554	2.22232	-0.0032249
0.90	2.45960	2.46267	0.0030664
1.00	2.71828	2.69693	-0.0213550

The comparison in Table 1.11 indicates that stagewise differentiation gives smaller errors than direct differentiation (this conclusion is for e^{ax} only and does not constitute a general result or proof). This validation is important since stagewise differentiation can be used to implement several types of PDE boundary conditions (BCs), as explained in examples to follow.

The preceding discussion of spline differentiation will now be used as the basis for the SCMOL integration of PDEs.

Appendix A1 – Online Documentation for `splinefun`

The technical details and options of `splinefun` are provided in the following online documentation produced at the R prompt by entering `help(splinefun)`.

```
splinefun {stats} R Documentation

Interpolating Splines

Description

Perform cubic (or Hermite) spline interpolation of
given data points, returning either a list of points
obtained by the interpolation or a function
performing the interpolation.
```

```
Usage

splinefun(x, y = NULL,
  method = c("fmm", "periodic", "natural",
   "monoH.FC"),
  ties = mean)

spline(x, y = NULL, n = 3*length(x), method = "fmm",
  xmin = min(x), xmax = max(x), xout, ties = mean)

splinefunH(x, y, m)
Arguments

x,y
vectors giving the coordinates of the points to be
inter-polated.   Alternatively a single plotting
structure can be specified: see xy.coords.

m
(for splinefunH()): vector of slopes m[i] at the
points (x[i],y[i]); these together determine the
Hermite ?spline? which is piecewise cubic, (only) once
differentiable continuously.

method
specifies the type of spline to be used. Possible val-
ues are "fmm","natural", "periodic" and "monoH.FC".

n
if xout is left unspecified, interpolation takes place
at n equally spaced points spanning the interval
[xmin, xmax].

xmin, xmax
left-hand and right-hand endpoint of the interpolation
interval (when xout is unspecified).

xout
an optional set of values specifying where interpola-
tion is to take place.
```

```
ties

Handling of tied x values. Either a function with a
single vector argument returning a single number
result or the string "ordered".

Details

The inputs can contain missing values that are
deleted, so at least one complete (x, y) pair is
required. If method = "fmm", the spline used is that
of Forsythe, Malcolm and Moler (an exact cubic is fit-
ted through the four points at each end of the data,
and this is used to determine the end conditions).
Natural splines are used when method = "natural", and
periodic splines when method = "periodic".

The new (R 2.8.0) method "monoH.FC" computes a mono-
tone Hermite spline according to the method of Fritsch
an Carlson. It does so by determining slopes such that
the Hermite spline, determined by (x[i],y[i],m[i]), is
monotone (increasing or decreasing) iff the data are.

These interpolation splines can also be used for
extrapolation, that is prediction at points outside
the range of x.  Extrapolation makes little sense for
method = "fmm"; for natural splines it is linear using
the slope of the interpolating curve at the nearest
data point.

Value

spline returns a list containing components x and y
which give the ordinates where interpolation took
place and the interpolated values.

splinefun returns a function with formal arguments x
and deriv, the latter defaulting to zero. This func-
tion can be used to evaluate the interpolating cubic
spline (deriv=0), or its derivatives (deriv=1,2,3) at
```

```
the points x, where the spline function interpolates
the data points originally specified. This is often
more useful than spline.

References
Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988)
The New S Language. Wadsworth & Brooks/Cole.

Forsythe, G. E., Malcolm, M. A. and Moler, C. B.
(1977) Computer Methods for Mathematical Computations.
Prentice-Hall

Fritsch, F. N. and Carlson, R. E. (1980) Monotone
piecewise cubic interpolation, SIAM Journal on Numeri-
cal Analysis 17, 238–246.
```

Reference

1 Forsythe, G.E., M.A. Malcolm, and C.B. Moler (1977), *Computer Methods for Mathematical Computations*, Prentice-Hall, Englewood Cliffs, NJ, pp. 70–79.