

# Appendix A

## Embedded Runge Kutta Pairs

The following 17 embedded RK pairs can be used for practice programming (Pairs 1, 9, and 11 are already programmed in [Chapters 1](#) and [2](#) and the programs can therefore be used as templates). The performance of the error estimators can also be studied by applying the algorithms to problems with known analytical solutions as was done in [Chapters 1 to 4](#). Of course, the 17 algorithms can also be applied to general  $nxn$  nonstiff ODE problems, with extensions to PDEs by the *method of lines*. Stiff ODEs can be studied by the methods discussed in [Appendix C](#).

The equations of each algorithm are listed in the order in which they would be programmed; i.e., they are listed in an executable sequence. Three pairs appear in routines discussed in [Chapter 2](#):

Pair	Routine
1	<i>euler2a</i> (fixed step) <i>euler2b</i> (variable step)
9	<i>rkf45a</i> (fixed step) <i>rkf45b</i> (variable step)
11	<i>rkc4a</i> (fixed step) <i>rkc4b</i> (variable step)

Pair 1 Calculation	(1,2) Chap 1 Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i + k_1, t_i + h)h$
Step $O(h^1)$	$y_{i+1}^{(1)} = y_i + k_1$
Est error	$e_{i+1}^{(1)} = \frac{1}{2}(k_2 - k_1)$
Step $O(h^2)$	$y_{i+1}^{(2)} = y_{i+1}^{(1)} + e_{i+1}^{(1)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 2 Calculation	(2,3) Chap 1 Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i + \frac{2}{3}k_1, t_i + \frac{2}{3}h)h$
Stage 3	$k_3 = f(y_i + \frac{2}{3}k_2, t_i + \frac{2}{3}h)h$
Step $O(h^2)$	$y_{i+1}^{(2)} = y_i + \frac{1}{4}k_1 + \frac{3}{4}k_2$
Step $O(h^3)$	$y_{i+1}^{(3)} = y_i + \frac{2}{8}k_1 + \frac{3}{8}k_2 + \frac{3}{8}k_3$
Est error	$e_{i+1}^{(2)} = y_{i+1}^{(3)} - y_{i+1}^{(2)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 3 Calculation	(2,3) Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i + \frac{k_1}{2}, t_i + \frac{h}{2})h$
Stage 3	$k_3 = f(y_i - k_1 + 2k_2, t_i + h)h$
Step $O(h^2)$	$y_{i+1}^{(2)} = y_i + \frac{1}{2}(k_1 + k_3)$
Est error	$e_{i+1}^{(2)} = \frac{1}{3}(-k_1 + 2k_2 - k_3)$
Step $O(h^3)$	$y_{i+1}^{(3)} = y_{i+1}^{(2)} + e_{i+1}^{(2)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 4 Calculation	(3,4) Equation
Stage 1	$k_1 = f(y_i, t_i)\frac{h}{3}$
Stage 2	$k_2 = f(y_i + k_1, t_i + \frac{h}{3})\frac{h}{3}$
Stage 3	$k_3 = f(y_i + \frac{1}{2}(k_1 + k_2), t_i + \frac{h}{3})\frac{h}{3}$
Stage 4	$k_4 = f(y_i + \frac{3}{8}k_1 + \frac{9}{8}k_3, t_i + \frac{h}{2})\frac{h}{3}$
Stage 5	$k_5 = f(y_i + \frac{3}{2}k_1 - \frac{9}{2}k_3 + 6k_4, t_i + h)\frac{h}{3}$
Step $O(h^3)$	$y_{i+1}^{(3)} = y_i + \frac{1}{2}(k_1 + 4k_4 + k_5)$
Est error	$e_{i+1}^{(3)} = \frac{1}{5}(-k_1 + \frac{9}{2}k_3 - 4k_4 + \frac{1}{2}k_5)$
Step $O(h^4)$	$y_{i+1}^{(4)} = y_{i+1}^{(3)} + e_{i+1}^{(3)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 5	(3,4)
Calculation	Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i - 0.4k_1, t_i - 0.4h)h$
Stage 3	$k_3 = f(y_i + 0.6684895833k_1 - 0.2434895833k_2, t_i + 0.425h)h$
Stage 4	$k_4 = f(y_i - 2.323685857k_1 + 1.125483559k_2 + 2.198202298k_3, t_i + h)h$
Step $O(h^3)$	$y_{i+1}^{(3)} = y_i + 0.03968253968k_2 + 0.7729468599k_3 + 0.18737060041k_4$
Est error	$e_{i+1}^{(3)} = -y_{i+1}^{(3)} + y_i + 0.03431372549k_1 + 0.02705627706k_2 + 0.7440130202k_3 + 0.1946169772k_4$
Step $O(h^4)$	$y_{i+1}^{(4)} = y_{i+1}^{(3)} + e_{i+1}^{(3)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 6	(4,5)
Calculation	Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i + 0.0005k_1, t_i + 0.0005h)h$
Stage 3	$k_3 = f(y_i - 80.89939470k_1 + 81.18439470k_2, t_i + 0.285h)h$
Stage 4	$k_4 = f(y_i + 2113.327899k_1 - 2117.778035k_2 + 5.442136522k_3, t_i + 0.992h)h$
Stage 5	$k_5 = f(y_i + 2249.757677k_1 - 2254.489040k_2 + 5.739991965k_3 - 0.008629230728k_4, t_i + h)h$
Step $O(h^4)$	$y_{i+1}^{(4)} = y_i - 131.2823524k_1 + 131.4998223k_2 + 0.4837620276k_3 + 0.2987680554k_4$
Est error	$e_{i+1}^{(4)} = -y_{i+1}^{(4)} + y_i + 65.80784286k_1 - 65.94767173k_2 + 0.7959885276k_3 + 4.715404915k_4 - 4.371564570k_5$
Step $O(h^5)$	$y_{i+1}^{(5)} = y_{i+1}^{(4)} + e_{i+1}^{(4)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 7	(3,4)
Calculation	Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i + \frac{1}{3}k_1, t_i + \frac{1}{3}h)h$
Stage 3	$k_3 = f(y_i + \frac{1}{6}(k_1 + k_2), t_i + \frac{1}{3}h)h$
Stage 4	$k_4 = f(y_i + \frac{1}{8}(k_1 + 3k_3), t_i + \frac{1}{2}h)h$
Stage 5	$k_5 = f(y_i + \frac{1}{2}k_1 - \frac{3}{2}k_3 + 2k_4, t_i + h)h$
Step $O(h^3)$	$y_{i+1}^{(3)} = y_i + \frac{1}{2}k_1 - \frac{3}{2}k_3 + 2k_4$
Est error	$e_{i+1}^{(3)} = -\frac{1}{3}k_1 + \frac{3}{2}k_3 - \frac{4}{3}k_4 + \frac{1}{6}k_5$
Step $O(h^4)$	$y_{i+1}^{(4)} = y_{i+1}^{(3)} + e_{i+1}^{(3)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 8 Calculation	(4,5) Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h$
Stage 3	$k_3 = f(y_i + \frac{1}{4}(k_1 + k_2), t_i + \frac{1}{2}h)h$
Stage 4	$k_4 = f(y_i - k_2 + 2k_3, t_i + h)h$
Stage 5	$k_5 = f(y_i + \frac{1}{27}(7k_1 + 10k_2 + k_4), t_i + \frac{2}{3}h)h$
Stage 6	$k_6 = f(y_i + \frac{1}{625}(28k_1 - 125k_2 + 546k_3 + 54k_4 - 378k_5), t_i + \frac{1}{5}h)h$
Step $O(h^4)$	$y_{i+1}^{(4)} = y_i + \frac{1}{6}(k_1 + 4k_3 + 4k_4)$
Est error	$e_{i+1}^{(4)} = \frac{1}{336}(-42k_1 - 224k_3 - 21k_4 + 162k_5 + 125k_6)$
Step $O(h^5)$	$y_{i+1}^{(5)} = y_{i+1}^{(4)} + e_{i+1}^{(4)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 9 Calculation	(4,5) Chap 1 Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i + \frac{1}{4}k_1, t_i + \frac{1}{4}h)h$
Stage 3	$k_3 = f(y_i + \frac{1}{32}(3k_1 + 9k_2), t_i + \frac{3}{8}h)h$
Stage 4	$k_4 = f(y_i + \frac{1}{2197}(1932k_1 - 7200k_2 + 7296k_3), t_i + \frac{12}{13}h)h$
Stage 5	$k_5 = f(y_i + \frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4, t_i + h)h$
Stage 6	$k_6 = f(y_i - \frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5, t_i + \frac{1}{2}h)h$
Step $O(h^4)$	$y_{i+1}^{(4)} = y_i + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5$
Step $O(h^5)$	$y_{i+1}^{(5)} = y_i + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6$
Est error	$e_{i+1}^{(4)} = y_{i+1}^{(5)} - y_{i+1}^{(4)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 10 Calculation	(4,5) Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i + \frac{1}{5}k_1, t_i + \frac{1}{5}h)h$
Stage 3	$k_3 = f(y_i + \frac{3}{40}k_1 + \frac{9}{40}k_2, t_i + \frac{3}{10}h)h$
Stage 4	$k_4 = f(y_i + \frac{3}{10}k_1 - \frac{9}{10}k_2 + \frac{6}{5}k_3, t_i + \frac{3}{5}h)h$
Stage 5	$k_5 = f(y_i - \frac{11}{54}k_1 + \frac{5}{2}k_2 - \frac{70}{27}k_3 + \frac{35}{27}k_4, t_i + h)h$
Stage 6	$k_6 = f(y_i + \frac{1631}{55296}k_1 + \frac{175}{512}k_2 + \frac{575}{13824}k_3 + \frac{44275}{110592}k_4$ $+ \frac{253}{4096}k_5, t_i + \frac{7}{8}h)h$
Step $O(h^4)$	$y_{i+1}^{(4)} = y_i + \frac{2825}{27648}k_1 + \frac{18575}{48384}k_3 + \frac{13525}{55296}k_4 + \frac{277}{14336}k_5 + \frac{1}{4}k_6$
Step $O(h^5)$	$y_{i+1}^{(5)} = y_i + \frac{37}{378}k_1 + \frac{250}{621}k_3 + \frac{125}{594}k_4 + \frac{512}{1771}k_6$
Est error	$e_{i+1}^{(4)} = y_{i+1}^{(5)} - y_{i+1}^{(4)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 11 Calculation	(2,4) Chap 1 Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h$
Stage 3	$k_3 = f(y_i + \frac{1}{2}k_2, t_i + \frac{1}{2}h)h$
Stage 4	$k_4 = f(y_i + k_3, t_i + h)h$
Step $O(h^2)$	$y_{i+1}^{(2)} = y_i + k_2$
Est error	$e_{i+1}^{(2)} = \frac{1}{6}(k_1 - 4k_2 + 2k_3 + k_4)$
Step $O(h^4)$	$y_{i+1}^{(4)} = y_{i+1}^{(2)} + e_{i+1}^{(2)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 12 Calculation	(2,4) Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i + \frac{1}{3}k_1, t_i + \frac{1}{3}h)h$
Stage 3	$k_3 = f(y_i - \frac{1}{3}k_1 + k_2, t_i + \frac{2}{3}h)h$
Stage 4	$k_4 = f(y_i + k_1 - k_2 + k_3, t_i + h)h$
Step $O(h^2)$	$y_{i+1}^{(2)} = y_i - \frac{1}{2}k_1 + \frac{3}{2}k_2$
Est error	$e_{i+1}^{(2)} = \frac{1}{8}(5k_1 - 9k_2 + 3k_3 + k_4)$
Step $O(h^4)$	$y_{i+1}^{(4)} = y_{i+1}^{(2)} + e_{i+1}^{(2)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 13 Calculation	(2,4) Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h$
Stage 3	$k_3 = f(y_i - (\frac{1}{2} - \frac{1}{\sqrt{2}})k_1 + (1 - \frac{1}{\sqrt{2}})k_2, t_i + \frac{1}{2}h)h$
Stage 4	$k_4 = f(y_i - \frac{1}{\sqrt{2}}k_2 + (1 + \frac{1}{\sqrt{2}})k_3, t_i + h)h$
Step $O(h^2)$	$y_{i+1}^{(2)} = y_i + k_2$
Est error	$e_{i+1}^{(2)} = \frac{1}{6}(k_1 - 2(2 + \frac{1}{\sqrt{2}})k_2 + 2(1 + \frac{1}{\sqrt{2}})k_3 + k_4)$
Step $O(h^4)$	$y_{i+1}^{(4)} = y_{i+1}^{(2)} + e_{i+1}^{(2)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 14 Calculation	(2,4) Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h$
Stage 3	$k_3 = f(y_i - \frac{1}{2}k_1 + k_2, t_i + \frac{1}{2}h)h$
Stage 4	$k_4 = f(y_i + \frac{1}{2}k_2 + \frac{1}{2}k_3, t_i + h)h$
Step $O(h^2)$	$y_{i+1}^{(2)} = y_i + k_2$
Est error	$e_{i+1}^{(2)} = \frac{1}{6}(k_1 - 3k_2 + k_3 + k_4)$
Step $O(h^4)$	$y_{i+1}^{(4)} = y_{i+1}^{(2)} + e_{i+1}^{(2)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 15	(2,4)
Calculation	Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i + \frac{2}{5}k_1, t_i + \frac{2}{5}h)h$
Stage 3	$k_3 = f(y_i + 0.29697760924775360k_1 + 0.15875964497103583k_2, t_i + 0.45573725421878943h)h$
Stage 4	$k_4 = f(y_i + 0.21810038822592047k_1 - 3.0509651486929308k_2 + 3.8328647604670103k_3, t_i + h)h$
Step $O(h^2)$	$y_{i+1}^{(2)} = y_i - \frac{1}{4}k_1 + 1.25k_2$
Est error	$e_{i+1}^{(2)} = 0.42476028226269037k_1 - 1.8014806628787329k_2 + 1.2055355993965235k_3 + 0.17118478121951903k_4$
Step $O(h^4)$	$y_{i+1}^{(4)} = y_{i+1}^{(2)} + e_{i+1}^{(2)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 16	(2,5)
Calculation	Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i + \frac{1}{2}k_1, t_i + \frac{1}{2}h)h$
Stage 3	$k_3 = f(y_i + \frac{1}{4}(k_1 + k_2), t_i + \frac{1}{2}h)h$
Stage 4	$k_4 = f(y_i - k_2 + 2k_3, t_i + h)h$
Stage 5	$k_5 = f(y_i + \frac{1}{27}(7k_1 + 10k_2 + k_4), t_i + \frac{2}{3}h)h$
Stage 6	$k_6 = f(y_i + \frac{1}{625}(28k_1 - 125k_2 + 546k_3 + 54k_4 - 378k_5), t_i + \frac{1}{5}h)h$
Step $O(h^2)$	$y_{i+1}^{(2)} = y_i + \frac{1}{2}k_2$
Est error	$e_{i+1}^{(2)} = \frac{1}{336}(14k_1 - 336k_2 + 35k_4 + 162k_5 + 125k_6)$
Step $O(h^5)$	$y_{i+1}^{(5)} = y_{i+1}^{(2)} + e_{i+1}^{(2)}$
Step $t$	$t_{i+1} = t_i + h$

Pair 17 Calculation	(2,5) Equation
Stage 1	$k_1 = f(y_i, t_i)h$
Stage 2	$k_2 = f(y_i + \frac{1}{3}k_1, t_i + \frac{1}{3}h)h$
Stage 3	$k_3 = f(y_i + \frac{1}{25}(4k_1 + 6k_2), t_i + \frac{2}{5}h)h$
Stage 4	$k_4 = f(y_i + \frac{1}{4}(k_1 - 12k_2 + 15k_3), t_i + h)h$
Stage 5	$k_5 = f(y_i + \frac{1}{81}(6k_1 + 90k_2 - 50k_3 + 8k_4), t_i + \frac{2}{3}h)h$
Stage 6	$k_6 = f(y_i + \frac{1}{75}(6k_1 + 36k_2 + 10k_3 + 8k_4), t_i + \frac{4}{5}h)h$
Step $O(h^2)$	$y_{i+1}^{(2)} = y_i - \frac{1}{2}k_1 + \frac{3}{2}k_2$
Est error	$e_{i+1}^{(2)} = \frac{1}{192}(119k_1 - 288k_2 + 125k_3 - 81k_5 + 125k_6)$
Step $O(h^5)$	$y_{i+1}^{(5)} = y_{i+1}^{(2)} + e_{i+1}^{(2)}$
Step $t$	$t_{i+1} = t_i + h$

Note that these 17 integrators have error estimates based on varying numbers of terms in the associated Taylor series. The number of terms in each error estimate, which just equals the difference in the orders of the pairs, is summarized below. For example, a (2, 4) pair has  $4 - 2 = 2$  terms in the error estimate.

Pairs	Terms in the Error Estimate
1,2,3,4,5,6,7,8,9,10	1
11,12,13,14,15	2
16,17	3

The algorithms with 3 term error estimates (Pairs 16 and 17) are recommended for ODE problems with particularly stringent accuracy requirements.

In addition to the references for Pairs 1, 2, 9, and 11 given in [Chapter 1](#), references for the other pairs (except 10), are given in Reference 1. This reference also provides Fortran routines for the ODE integrators summarized above (except 2 and 10). Pair 10 is taken from Reference 2.

## References

1. Silebi, C.A., and W.E. Schiesser, *Dynamic Modeling of Transport Process Systems*, Academic Press, San Diego, CA, 1992.
2. Press, W.H., S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in Fortran*, 2nd ed., Cambridge University Press, Cambridge, U.K., 1992.



# Appendix B

---

## *Integrals from ODEs*

---

The ODE algorithms discussed in [Chapter 1](#) can be used to compute one-dimensional integrals. Specifically, the special case ODE (with the derivative function  $f(t)$  a function of the independent variable  $t$  and not the dependent variable  $y$ )

$$\frac{dy}{dt} = f(t), y(t_0) = y_0 \quad (\text{B.1})$$

has the solution

$$y(t) = y_0 + \int_{t_0}^t f(\tau) d\tau \quad (\text{B.2})$$

Thus, we can integrate Equation B.1 to produce a numerical value of the integral of Equation B.2. To illustrate this procedure, consider the ODE

$$\frac{dy}{dt} = \frac{2}{\sqrt{\pi}} e^{-t^2}, y(0) = 0 \quad (\text{B.3})$$

which has the solution

$$y(t) = \frac{2}{\sqrt{\pi}} \int_0^t e^{-\tau^2} d\tau \quad (\text{B.4})$$

$y(t)$  of Equation B.4 is the well-known *error function*,  $\text{erf}(t)$ , which has broad application in science and engineering, and is tabulated extensively.

Functions *intpar*, *inital*, *derv*, and *fprint*, which follow directly from the preceding discussion (for the solution of Equation B.3), are listed below:

```
function [neqn,nout,nsteps,t0,tf,abserr,relerr]=intpar
%
% Function intpar sets the parameters to control the
% integration of the ODE with erf as a solution
%
% Number of first order ODEs
    neqn=1;
```

```

%
% Number of output points
nout=6;
%
% Maximum number of steps in the interval t0 to tf
nsteps=100;
%
% Initial, final values of independent variable
t0=0.0;
tf=0.1;
%
% Error tolerances
abserr=1.0e-05;
relerr=1.0e-05;

function [y]=inital(neqn,t)
%
% Function inital sets the initial condition vector
% for the ODE with erf as the solution
%
% Initial condition
y(1)=0.0;

function [yt]=derv(neqn,t,y)
%
% Function derv computes the derivative vector
% of the ODE with erf as the solution
%
% Declare global variables
global nsteps;
%
% ODE
yt(1)=exp(-t^2);

function [out]=fprint(ncase,neqn,t,y)
%
% Function fprint displays the numerical and
% tabulated solutions to the ODE with erf as
% a solution
%
% Define global variables
global nsteps;

```

```

%
% Return current value of independent variable
% (MATLAB requires at least one return argument)
out=t;
%
% Print a heading for the solution at t = 0
if(t<=0.0)
%
%   Label for ODE integrator
%
%   Fixed step modified Euler
    if(ncase==1)
        fprintf('\n\n euler2a integrator\n\n');
%
%   Variable step modified Euler
    elseif(ncase==2)
        fprintf('\n\n euler2b integrator\n\n');
%
%   Fixed step classical fourth order RK
    elseif(ncase==3)
        fprintf('\n\n rkc4a integrator\n\n');
%
%   Variable step classical fourth order RK
    elseif(ncase==4)
        fprintf('\n\n rkc4b integrator\n\n');
%
%   Fixed step RK Fehlberg 45
    elseif(ncase==5)
        fprintf('\n\n rkf45a integrator\n\n');
%
%   Variable step RK Fehlberg 45
    elseif(ncase==6)
        fprintf('\n\n rkf45b integrator\n\n');
    end
%
% Heading
    fprintf(' ncase = %2d   nsteps = %3d\n\n',ncase,nsteps);
    fprintf('      t          y(num)          y(tab)          diff\n');
%
% End of t = 0 heading
end
%
% Tabulated solution
if(t<0.001) ytab=0.0; end
if((t>0.0999)&(t<0.1001)) ytab=0.112463; end

```

```

    if((t>0.1999)&(t<0.2001)) ytab=0.222703; end
    if((t>0.2999)&(t<0.3001)) ytab=0.328627; end
    if((t>0.3999)&(t<0.4001)) ytab=0.428392; end
    if((t>0.4999)&(t<0.5001)) ytab=0.520500; end
%
% Numerical and tabulated solutions and their difference
% as output
    y1=(2.0/sqrt(pi))*y(1);
    diff=y1-ytab;
%
% Display the numerical and exact solutions, and their
% difference
    fprintf('%5.2f %11.6f %11.6f %11.6f\n',t,y1,ytab,diff);

```

### Program B.1

*intpar, initial, derv, and fprint* for the solution of Equation B.3

The main program is again Program 3.1.1. The output from the preceding program is listed below

euler2a integrator

ncase = 1    nsteps = 100

t	y(num)	y(tab)	diff
0.00	0.000000	0.000000	0.000000
0.10	0.112463	0.112463	0.000000
0.20	0.222703	0.222703	0.000000
0.30	0.328627	0.328627	0.000000
0.40	0.428392	0.428392	0.000000
0.50	0.520500	0.520500	0.000000

euler2b integrator

ncase = 2    nsteps = 100

t	y(num)	y(tab)	diff
0.00	0.000000	0.000000	0.000000
0.10	0.112461	0.112463	-0.000002
0.20	0.222700	0.222703	-0.000003
0.30	0.328623	0.328627	-0.000004
0.40	0.428388	0.428392	-0.000004
0.50	0.520496	0.520500	-0.000004

rkc4a integrator

ncase = 3    nsteps = 100

t	y(num)	y(tab)	diff
0.00	0.000000	0.000000	0.000000
0.10	0.112463	0.112463	0.000000
0.20	0.222703	0.222703	0.000000
0.30	0.328627	0.328627	0.000000
0.40	0.428392	0.428392	0.000000
0.50	0.520500	0.520500	0.000000

rkc4b integrator

ncase = 4    nsteps = 100

t	y(num)	y(tab)	diff
0.00	0.000000	0.000000	0.000000
0.10	0.112463	0.112463	0.000000
0.20	0.222703	0.222703	0.000000
0.30	0.328627	0.328627	0.000000
0.40	0.428392	0.428392	0.000000
0.50	0.520500	0.520500	0.000000

rkf45a integrator

ncase = 5    nsteps = 100

t	y(num)	y(tab)	diff
0.00	0.000000	0.000000	0.000000
0.10	0.112463	0.112463	0.000000
0.20	0.222703	0.222703	0.000000
0.30	0.328627	0.328627	0.000000
0.40	0.428392	0.428392	0.000000
0.50	0.520500	0.520500	0.000000

rkf45b integrator

ncase = 6    nsteps = 100

t	y(num)	y(tab)	diff
0.00	0.000000	0.000000	0.000000
0.10	0.112463	0.112463	0.000000

0.20	0.222703	0.222703	0.000000
0.30	0.328627	0.328627	0.000000
0.40	0.428392	0.428392	0.000000
0.50	0.520500	0.520500	0.000000

The error tolerances set in *intpar* are satisfied by all six integrators. Of course, this problem tests only the stepping in  $t$ , and not  $y$ , because of the special form of Equation B.1.

# Appendix C

---

## *Stiff ODE Integration*

---

The explicit integrators discussed in [Chapters 1](#) and [2](#), and applied to ODE and PDE systems in [Chapters 2](#) to [5](#), can be used for a broad spectrum of applications. However, as discussed in [Section 1.7](#), these explicit (nonstiff) integrators will require lengthy calculations for *stiff systems*. Again, this difficulty with stiff systems results from having to take small integration steps to maintain stability of the numerical integration (because of the stability constraints of explicit algorithms) while having to cover a large interval in the independent variable to compute a complete solution. An example is the 2x2 system of [Chapter 3](#) for which the maximum integration step to maintain stability is set by the largest eigenvalue, while the timescale of the solution is set by the smallest eigenvalue. If the eigenvalues are widely separated, the problem is stiff, and an implicit integrator is required to compute a solution efficiently, as we will demonstrate.

We now illustrate some of the properties of stiff systems using the 2x2 system. Specifically, we will consider:

- The BDF formulas of [Section 1.7](#) applied to the 2x2 ODE system
- A MATLAB program for the *first-order BDF method (implicit Euler method)*
- The 2x2 ODE system integrated by the MATLAB stiff integrators *ode23s* and *ode15s*

---

### C.1 The BDF Formulas Applied to the 2x2 ODE System

The BDF method is first applied to the general (possibly nonlinear) 2x2 ODE system

$$\begin{aligned}dy_1/dt &= f_1(y_1, y_2, t) \\ dy_2/dt &= f_2(y_1, y_2, t)\end{aligned}\tag{C.1}$$

The BDF equations are

$$\alpha_0 y_{i+1,1} + \alpha_1 y_{i,1} + \cdots + \alpha_v y_{i-v+1,1} = hf_1(y_{i+1,1}, y_{i+1,2}, t_{i+1})$$

$$\alpha_0 y_{i+1,2} + \alpha_1 y_{i,2} + \cdots + \alpha_v y_{i-v+1,2} = hf_2(y_{i+1,1}, y_{i+1,2}, t_{i+1})$$

or

$$\begin{aligned} hf_1(y_{i+1,1}, y_{i+1,2}, t_{i+1}) - [\alpha_0 y_{i+1,1} + \alpha_1 y_{i,1} + \cdots + \alpha_v y_{i-v+1,1}] \\ = g_1(y_{i+1,1}, y_{i+1,2}, t_{i+1}) = 0 \\ hf_2(y_{i+1,1}, y_{i+1,2}, t_{i+1}) - [\alpha_0 y_{i+1,2} + \alpha_1 y_{i,2} + \cdots + \alpha_v y_{i-v+1,2}] \\ = g_2(y_{i+1,1}, y_{i+1,2}, t_{i+1}) = 0 \end{aligned} \quad (C.2)$$

Equations C.2 are a 2x2 nonlinear system (nonlinear if the original ODE system, Equations C.1, is nonlinear), for the two unknowns  $y_{i+1,1}$ ,  $y_{i+1,2}$  (the ODE solution at the advanced point  $i+1$ ). We need to apply a nonlinear solver to Equations C.2 such as Newton's method to compute the two unknowns,  $y_{i+1,1}$ ,  $y_{i+1,2}$ .

$$\mathbf{J}\delta\mathbf{y} = -\mathbf{g}(\mathbf{y}) \quad (C.3)$$

where for the  $n \times n$  problem

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \delta\mathbf{y} = \begin{bmatrix} \delta y_1 \\ \delta y_2 \\ \vdots \\ \delta y_n \end{bmatrix} \quad (C.4)(C.5)$$

$$\mathbf{g} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_n \end{bmatrix} = \begin{bmatrix} g_1(y_1, y_2, \dots, y_n, t) \\ g_2(y_1, y_2, \dots, y_n, t) \\ \vdots \\ g_n(y_1, y_2, \dots, y_n, t) \end{bmatrix} \quad (C.6)$$

$$\mathbf{J} = \begin{bmatrix} \frac{\partial g_1}{\partial y_1} & \frac{\partial g_1}{\partial y_2} & \cdots & \frac{\partial g_1}{\partial y_n} \\ \frac{\partial g_2}{\partial y_1} & \frac{\partial g_2}{\partial y_2} & \cdots & \frac{\partial g_2}{\partial y_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_n}{\partial y_1} & \frac{\partial g_n}{\partial y_2} & \cdots & \frac{\partial g_n}{\partial y_n} \end{bmatrix} = \begin{bmatrix} J_{11} & J_{12} & \cdots & J_{1n} \\ J_{21} & J_{22} & \cdots & J_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ J_{n1} & J_{n2} & \cdots & J_{nn} \end{bmatrix} \quad (C.7)$$



$\mathbf{J}$  is the  $n \times n$  *Jacobian matrix*, consisting of all first-order partial derivatives of the functions,  $[g_1 \ g_2 \ \dots \ g_i \ \dots \ g_n]^T$  with respect to the dependent variables  $[y_1 \ y_2 \ \dots \ y_j \ \dots \ y_n]^T$ , i.e.,

$$J_{ij} = \frac{\partial g_i}{\partial y_j} \quad (\text{C.8})$$

$\delta \mathbf{y}$  is the vector of *Newton corrections*, which should decrease below a specified tolerance or threshold as the Newton iteration proceeds.

Application of the preceding equations to the 2x2 ODE system of Equations 1.6 of [Chapter 1](#):

$$\begin{aligned} \frac{dy_1}{dt} &= -ay_1 + by_2 \\ \frac{dy_2}{dt} &= by_1 - ay_2 \end{aligned} \quad (\text{C.9})$$

gives

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, \quad \delta \mathbf{y} = \begin{bmatrix} \delta y_1 \\ \delta y_2 \end{bmatrix} \quad (\text{C.10})(\text{C.11})$$

$$\begin{aligned} \mathbf{g} &= \begin{bmatrix} g_1(y_1, y_2, t) \\ g_2(y_1, y_2, t) \end{bmatrix} \\ &= \begin{bmatrix} h[-ay_{i+1,1} + by_{i+1,2}] - [\alpha_0 y_{i+1,1} + \alpha_1 y_{i,1} + \dots + \alpha_v y_{i-v+1,1}] \\ h[+by_{i+1,1} - ay_{i+1,2}] - [\alpha_0 y_{i+1,2} + \alpha_1 y_{i,2} + \dots + \alpha_v y_{i-v+1,2}] \end{bmatrix} \end{aligned} \quad (\text{C.12})$$

$$\mathbf{J} = \begin{bmatrix} \frac{\partial g_1}{\partial y_1} & \frac{\partial g_1}{\partial y_2} \\ \frac{\partial g_2}{\partial y_1} & \frac{\partial g_2}{\partial y_2} \end{bmatrix} = \begin{bmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{bmatrix} = \begin{bmatrix} -ah - \alpha_0 & bh \\ bh & -ah - \alpha_0 \end{bmatrix} \quad (\text{C.13})$$

Thus, Equation C.3 becomes

$$\begin{aligned} &\begin{bmatrix} -ah - \alpha_0 & bh \\ bh & -ah - \alpha_0 \end{bmatrix} \begin{bmatrix} \delta y_1 \\ \delta y_2 \end{bmatrix} \\ &= - \begin{bmatrix} h[-ay_{i+1,1} + by_{i+1,2}] - [\alpha_0 y_{i+1,1} + \alpha_1 y_{i,1} + \dots + \alpha_v y_{i-v+1,1}] \\ h[+by_{i+1,1} - ay_{i+1,2}] - [\alpha_0 y_{i+1,2} + \alpha_1 y_{i,2} + \dots + \alpha_v y_{i-v+1,2}] \end{bmatrix} \end{aligned} \quad (\text{C.14})$$

For the first-order BDF (*first-order implicit RK* or *implicit Euler method*),  $v = 1$ ,  $\alpha_0 = 1$ ,  $\alpha_1 = -1$ ,  $\alpha_2, \dots, \alpha_v = 0$ . Thus, Equations C.14 require only the preceding values  $y_{i,1}$ ,  $y_{i,2}$  (generally RK methods are *self-starting* while higher-order BDF methods require a *history of the solution* at the points  $i-1$ ,  $i-2$ ,  $\dots$ ,  $i-v+1$ )

to take the step from  $i$  to  $i + 1$ ; but, of course, the higher-order BDF methods have better accuracy than the first-order BDF).

Equations C.14 are programmed in the following MATLAB program. The numerical integration continues until the condition

$$|\delta y_1| < \textit{eps}$$

$$|\delta y_2| < \textit{eps}$$

is satisfied where  $\textit{eps}$  is a tolerance set in the program.

---

## C.2 MATLAB Program for the Solution of the 2x2 ODE System

Equations C.14 with  $\nu = 1$  (first-order BDF or the implicit Euler method) are solved by Newton's method in the following program:

```
%
% 2 x 2 ODE system by first order BDF
%
% Open a file for output
fid=fopen('appc1.out','w');
%
% Number of ODEs
neqn=2;
%
% Convergence tolerance, maximum number of
% iterations to compute Newton corrections
eps=0.00001;
maxiter=20;
%
% Initial condition
y10=0.0;
y20=2.0;
y(1)=y10;
y(2)=y20;
t=0.0;
%
% Values just for initial output
dy(1)=0.0;
dy(2)=0.0;
niter=0;
%
% Integration step, steps/output, number of outputs
h=0.01;
```

```

nsteps=100;
nout=11;
%
% Problem parameters
a=500000.5;
b=499999.5;
%
% Jacobian matrix
J(1,1)=-a*h-1.0;
J(1,2)=b*h;
J(2,1)=b*h;
J(2,2)=-a*h-1.0;
%
% Print a heading
fprintf('      t      dy(1)      dy(2)
      y(1)      y(2)      erry(1)      erry(2)      iter\n');
%
% nout outputs
for i=1:nout
%
%   Initial output or after nstep integration steps
%   completed; display Newton corrections, numerical and
%   exact solutions for output
lambda1=-(a-b);
lambda2=-(a+b);
exp1=exp(lambda1*t);
exp2=exp(lambda2*t);
yle=exp1-exp2;
y2e=exp1+exp2;
erry1=yle-y(1);
erry2=y2e-y(2);
fprintf(fid, '%10.5f%10.5f%10.5f%10.5f%10.5f%10.5f
              %10.5f%5d\n', ... t, dy(1), dy(2), y(1), y(2),
              erry1, erry2, niter);
fprintf('      %10.5f%10.5f%10.5f%10.5f%10.5f%10.5f
              %10.5f%5d\n', ... t, dy(1), dy(2), y(1), y(2),
              erry1, erry2, niter);
%
% nsteps steps/output
for is=1:nsteps
%
%   Initialize iteration counter
niter=1;
stop=0;
%
```

```

%      Test for the end of the current step
while stop==0
%
%      Functions g1, g2
g=[h*(-a*y(1)+b*y(2))-y(1)+y10
   h*( b*y(1)-a*y(2))-y(2)+y20];
%
%      Solve for Newton corrections
%
%      Gaussian elimination
dy=-J\g;
%
%      Jacobian inverse
dy=-J^(-1)*g;
%
%      Update solution
for ic=1:neqn
    y(ic)=y(ic)+dy(ic);
end
stop=1;
%
%      Check if the corrections are within the tolerance eps
for ic=1:neqn
    if abs(dy(ic))>eps
%
%      Convergence not achieved; continue calculation
niter=niter+1;
stop=0;
break;
    end
end
%
%      If maximum iterations reached, accept current step
if (niter==maxiter) stop=1; end
%
%      Continue integration step
end
%
%      Integration step completed
y10=y(1);
y20=y(2);
t=t+h;
%
%      Continue nstep integration steps
end

```

```
%
% Continue integration for next output interval
end
```

### Program C.1

Solution of Equations C.14 by Newton's method

We can note the following points about Program C.1:

- An output file is defined, then the number of nonlinear equations to be solved, in this case 2 for Equations C.14:

```
%
% 2 x 2 ODE system by first order BDF
%
% Open a file for output
fid=fopen('appc1.out','w');
%
% Number of ODEs
neqn=2;
```

- The tolerance for the Newton corrections and the maximum number of Newton iterations are then defined numerically:

```
%
% Convergence tolerance, maximum number of
% iterations to compute Newton corrections
eps=0.00001;
maxiter=20;
```

- An initial estimate of the solution, required by Newton's method is defined, which is taken as the initial condition for Equations C.9 ( $y_1(0) = 0$ ,  $y_2(0) = 2$ ). Also, the independent variable,  $t$ , in Equations C.9 is initialized:

```
%
% Initial condition
y10=0.0;
y20=2.0;
y(1)=y10;
y(2)=y20;
t=0.0;
```

- The Newton corrections are zeroed and the counter for the Newton iterations is initialized:

```
%
% Values just for initial output
dy(1)=0.0;
dy(2)=0.0;
niter=0;
```

- The variables that control the BDF integration are set, specifically, the integration step in Equations C.14 (this will be for a fixed step BDF integration), the maximum number of integration steps for each output interval, and the number of outputs:

```
%
% Integration step, steps/output, number of outputs
h=0.01;
nsteps=100;
nout=11;
```

- The parameters in Equations C.9 are set:

```
%
% Problem parameters
a=500000.5;
b=499999.5;
```

These values correspond to the stiff case discussed in Section 1.7 for which the eigenvalues of Equations 1.6 (or C.9) are  $\lambda_1 = -(a - b) = -1$ ,  $\lambda_2 = -(a + b) = -1,000,000$ . Thus, there will be a total of  $100 \times (11 - 1) = 1000$  Newton steps, each of length 0.01 so that the final value of  $t$  is  $0.01 \times 1000 = 10$ .

- The elements of the  $n \times n = 2 \times 2$  Jacobian matrix of Equation C.13 are programmed:

```
%
% Jacobian matrix
J(1,1)=-a*h-1.0;
J(1,2)=b*h;
J(2,1)=b*h;
J(2,2)=-a*h-1.0;
```

Note that since Equations C.9 are linear (constant coefficient ODEs), the *Jacobian matrix is constant* and therefore has to be evaluated only once. More generally, if the ODEs are nonlinear, the Jacobian matrix would have to be updated at each Newton iteration (which is a major portion of the calculation in using Newton's method).

- A heading for the numerical solution is then printed:

```
%
% Print a heading
fprintf('          t          dy(1)      dy(2)
        y(1)      y(2)      erry(1)      erry(2)      iter\n');
```

(this output statement has been put in two lines to fit within the available printed space; they should be returned to single lines if Equation C.1 is executed).

- An outer loop is used to output the solution at the *nout* output points:

```
%
% nout outputs
for i=1:nout
%
%   Initial output or after nstep integration steps
%   completed; display Newton corrections, numerical
%   and exact solutions for output
    lambda1=-(a-b);
    lambda2=-(a+b);
    exp1=exp(lambda1*t);
    exp2=exp(lambda2*t);
    yle=exp1-exp2;
    y2e=exp1+exp2;
    erry1=y1e-y(1);
    erry2=y2e-y(2);
    fprintf(fid, '%10.5f%10.5f%10.5f%10.5f%10.5f%10.5f
                %10.5f%5d\n', ...t, dy(1), dy(2), y(1),
                y(2), erry1, erry2, niter);
    fprintf(' %10.5f%10.5f%10.5f%10.5f%10.5f%10.5f
            %10.5f%5d\n', ...t, dy(1), dy(2), y(1),
            y(2), erry1, erry2, niter);
```

The first task in this loop is to compute the exact solution to Equations 1.6 (or Equations C.9), i.e., Equations 1.17. The difference between the numerical and exact solution is then computed, followed by printing of the independent variable,  $t$ , the two Newton corrections, the numerical solution, the error in the numerical solution, and the number of iterations required to produce the numerical solution.

- An intermediate loop performs the calculations through *nsteps* integration steps of length  $h$ ; the iteration counter is initialized for each integration step and a variable is initialized, which will indicate when the Newton iterations are stopped:

```

%
%   nsteps steps/output
%   for is=1:nsteps
%
%       Initialize iteration counter
%       niter=1;
%       stop=0;

```

- An inner loop then performs the Newton iterations while  $nstop=0$ ; first the vector of functions to be zeroed is computed according to Equation C.12 (with  $v=1$ ):

```

%
%       Test for the end of the current step
%       while stop==0
%
%           Functions g1, g2
%           g=[h*(-a*y(1)+b*y(2))-y(1)+y10
%             h*( b*y(1)-a*y(2))-y(2)+y20];

```

- The linear Newton Equations C.3 are then solved for the Newton corrections:

```

%
%       Solve for Newton corrections
%
%       Gaussian elimination
%       dy=-J\g;
%
%       Jacobian inverse
%       dy=-J^(-1)*g;

```

Note that two methods of solution are programmed:

- Gaussian elimination, which is the preferred method of solution
- Inverse Jacobian matrix, i.e.,

$$\delta \mathbf{y} = -\mathbf{J}^{-1} \mathbf{g}(\mathbf{y}) \quad (\text{C.15})$$

Although Equation C.15 is formally (mathematically) correct, it is not used in practice because computing the inverse Jacobian matrix is inefficient; rather, some form of Gaussian elimination is generally used. Note, however, how easily either method is programmed in MATLAB; this is due to the facility of MATLAB to handle matrices (arrays) without subscripting, plus the definition of basic matrix operations, e.g., multiplication, inverse, Gaussian reduction. This step (for computing the Newton corrections) will fail if the Jacobian matrix is *singular* or *near singular* (i.e., *ill-conditioned*). Thus, the calculation of the *condition* of  $\mathbf{J}$  at



this point would be a good idea. MATLAB has a utility for calculating the condition of a matrix which can then be compared with the machine epsilon discussed in Section 1.7; specifically, if the *condition number exceeds the reciprocal of the machine epsilon, the linear algebraic system* (in this case, Equation C.3) *is numerically singular*.

- The Newton corrections are then applied to the current solution vector to produce (one hopes) an improved solution:

```
%
%      Update solution
%      for ic=1:neqn
%          y(ic)=y(ic)+dy(ic);
%      end
%      stop=1;
```

If the Newton corrections are small enough (to be tested next), the iterations are terminated by setting *stop*= 1.

- Each Newton correction is tested against the convergence tolerance:

```
%
%      Check if the corrections are within the
%      tolerance eps
%      for ic=1:neqn
%          if abs(dy(ic))>eps
%
%              Convergence not achieved; continue
%              calculation
%              niter=niter+1;
%              stop=0;
%              break;
%          end
%      end
```

If *any* Newton correction exceeds the tolerance, the iteration counter is incremented, the iterations are continued (*stop* = 0) and the testing is ended (*break* from the *for* loop)

- If the maximum number of iterations is reached, the iterations are stopped. Otherwise, if the iterations are to be continued (*stop* = 0), the next pass through the *while* loop (based on *stop* == 0) is initiated

```
%
%      If maximum iterations reached, accept current step
%      if (niter==maxiter) stop=1; end
%
%      Continue integration step
%      end
```

- Convergence has been achieved ( $stop = 1$ ) so the new solution is now used as the old (base) solution in the next step of the ODE integration; the independent variable is incremented for the next step along the solution:

```
%
%      Integration step completed
y10=y(1);
y20=y(2);
t=t+h;
```

- Finally, the  $nstep$  integration steps are completed, and the next output interval is covered until all  $nout$  output intervals are completed.

```
%
%      Continue nstep integration steps
end
%
% Continue integration for next output interval
end
```

Note that Program C.1 contains a general Newton solver that can be applied to an  $nxn$  system of nonlinear equations; all that really is required is to reprogram: (1) the Jacobian matrix (which generally will be inside the loop for the Newton iterations rather than outside) and (2) the vector of functions to be zeroed. Also, some tuning of the parameters will generally be required (e.g., the tolerance  $eps$  and the maximum number of iterations  $maxiter$ ).

The output from the preceding program is as follows:

t	dy(1)	dy(2)	y(1)	y(2)	erry(1)	erry(2)	iter
0.00000	0.00000	0.00000	0.00000	2.00000	0.00000	0.00000	0
1.00000	0.00000	0.00000	0.36971	0.36971	-0.00183	-0.00183	2
2.00000	0.00000	0.00000	0.13669	0.13669	-0.00135	-0.00135	2
3.00000	0.00000	0.00000	0.05053	0.05053	-0.00075	-0.00075	2
4.00000	0.00000	0.00000	0.01868	0.01868	-0.00037	-0.00037	2
5.00000	0.00000	0.00000	0.00691	0.00691	-0.00017	-0.00017	2
6.00000	0.00000	0.00000	0.00255	0.00255	-0.00007	-0.00007	2
7.00000	-0.00001	-0.00001	0.00094	0.00094	-0.00003	-0.00003	1
8.00000	0.00000	0.00000	0.00035	0.00035	-0.00001	-0.00001	1
9.00000	0.00000	0.00000	0.00013	0.00013	-0.00001	-0.00001	1
10.00000	0.00000	0.00000	0.00005	0.00005	0.00000	0.00000	1

We can note the following points about this example:

- Only 1000 implicit Euler steps were used. This contrasts with the  $5 \times 10^6$  steps estimated for the explicit Euler method in Section 1.7; thus there was a reduction of 1/5000 in the number of steps required by an explicit integrator, which clearly shows the advantage of using an implicit integrator for stiff ODEs. Again, as discussed in Section 1.7, if this

conclusion is not convincing, using  $a = 500,000,000.5$ ,  $b = 499,999,999.5$  would result in a reduction of  $1/5 \times 10^6$  steps!

- The two Newton corrections,  $\delta y_1$  and  $\delta y_2$ , met the tolerance  $eps = 0.00001$  with no more than two iterations, which is an indication of the *quadratic convergence of Newton's method*, i.e., when Newton's method works, it generally is very efficient.
- Accuracy (not stability) was limited by the step size  $h = 0.01$ ; this suggests a higher-order BDF method could be used to good advantage (to increase the accuracy, while maintaining stability). Specifically, the BDF methods are stable along the entire negative real axis, and therefore would be stable for the  $2 \times 2$  linear problem of Equations C.9 (since the two eigenvalues are real and negative, e.g.,  $\lambda_1 = -(a - b) = -1$ ,  $\lambda_2 = -(a + b) = -1,000,000$ ). An extension of Program C.1 for BDF methods of order 2 and 3 is available from the authors (W.E.S.).

To investigate this last point (the possible advantage of using higher-order BDF methods), we now consider the use of a stiff ODE integrator in MATLAB, which varies the order of the BDF method.

---

### C.3 MATLAB Program for the Solution of the 2x2 ODE System Using *ode23s* and *ode15s*

A MATLAB program that calls *ode23s* and *ode15s* for the solution of the 2x2 ODE system of Equations C.9 is listed below:

```
%
% 2 x 2 ODE system by variable order BDF
%
% Global variables
global a b ncall;
%
% Model parameters
a=500000.5;
b=499999.5;
%
% Select method
for mf=1:2
%
% Error tolerances
reltol=1.0e-02;
abstol=1.0e-02;
%
```

```

% Cases for changes in error tolerances
for ncase=1:2
    reltol=1.0e-02*reltol;
    abstol=1.0e-02*abstol;
%
% Variables for ODE integration
t0=0.0;
tf=10.0;
tout=[t0:1.0:tf]';
nout=11;
%
% Initialize number of calls to derivative
% subroutine
ncall=0;
%
% Initial condition
y10=0.0;
y20=2.0;
y0=[y10 y20]';
%
% Call ODE integrator
options=odeset('RelTol',reltol,'AbsTol',abstol);
if(mf==1) [t,y]=ode23s('ode2x2',tout,y0,options); end
if(mf==2) [t,y]=ode15s('ode2x2',tout,y0,options); end
%
% Display solution and error
fprintf('\n\n mf = %1d\n case = %1d\n\n reltol = %6.2e\n abstol = %6.2e\n\n',...
mf,ncase,reltol,abstol);
fprintf('      t      y1e      y1      erry1\n\n      y2e      y2      erry2\n');
for i=1:nout
    lambda1=-(a-b);
    lambda2=-(a+b);
    exp1=exp(lambda1*t(i));
    exp2=exp(lambda2*t(i));
    y1e=(y10+y20)/2.0*exp1-(y20-y10)/2.0*exp2;
    y2e=(y10+y20)/2.0*exp1+(y20-y10)/2.0*exp2;
    erry1=y1e-y(i,1);
    erry2=y2e-y(i,2);
    fprintf('%5.1f%9.4f%9.4f%15.10f\n\n      %9.4f%9.4f%15.10f\n\n',...
t(i),y1e,y(i,1),erry1,y2e,y(i,2),erry2);
end
%

```

```

% Next case
    fprintf('  ncall = %5d\n',ncall);
end
%
% Next method
end
%
% Plot last solution
plot(t,y);
xlabel('t')
ylabel('y1(t),y2(t)')
title(' Appendix C, 2 x 2 Linear System')
gtext('y1(t)');
gtext('y2(t)');
print appc.ps

```

## Program C.2

Solution of Equations C.9 by *ode23s* and *ode15s*

We can note the following points about Program C.2:

- The beginning of the program is similar to that of Program C.1; three variables are declared *global* so that they can be shared with other routines:

```

%
% 2 x 2 ODE system by variable order BDF
%
% Global variables
global a b ncall;
%
% Model parameters
a=500000.5;
b=499999.5;

```

- An outer loop executes two times; for the first ( $mf = 1$ ), *ode23s* is called, while for the second, *ode15s* is called:

```

%
% Select method
for mf=1:2
%
% Error tolerances
reltol=1.0e-02;
abstol=1.0e-02;

```

For each of the two passes through this *for* loop, the relative and absolute error tolerances for the two ODE integrators are set to  $10^{-2}$ .

- In a subordinate loop, the error tolerances are reduced by  $10^{-2}$  (so that for the two passes through this loop, the error tolerances are set to  $10^{-4}$  and  $10^{-6}$ ); in this way, the performance of the integrators can be assessed using the exact solution of the  $2 \times 2$  ODE problem, Equation 1.17:

```
%
% Cases for changes in error tolerances
for ncase=1:2
    reltol=1.0e-02*reltol;
    abstol=1.0e-02*abstol;
```

- The variables controlling the integration are set:

```
%
% Variables for ODE integration
t0=0.0;
tf=10.0;
tout=[t0:1.0:tf]';
nout=11;
```

Note that the solution is computed to a final time  $tf = 10$  with 11 outputs at an interval of 1.

- A counter for the number of calls to the derivative routine (discussed subsequently) is initialized; also, the initial conditions for the two ODEs (Equations C.9) are set (these are a  $2 \times 1$  column vector, which is defined as the transpose of a  $1 \times 2$  row vector):

```
%
% Initialize number of calls to derivative
% subroutine
ncall=0;
%
% Initial condition
y10=0.0;
y20=2.0;
y0=[y10 y20]';
```

- The MATLAB utility *odeset* is called to set the error tolerances for the subsequent calls to *ode23s* and *ode15s*:

```
%
% Call ODE integrator
options=odeset('RelTol',reltol,'AbsTol',abstol);
if(mf==1) [t,y]=ode23s('ode2x2',tout,y0,options); end
if(mf==2) [t,y]=ode15s('ode2x2',tout,y0,options); end
```

Note that the two integrators call the function *ode2x2* to define the derivatives (RHS functions) of Equations C.9. Also, the coding to call *ode23s* and *ode15s* is straightforward (which facilitates their use).



```

gtext('y1(t)');
gtext('y2(t)');
print appc.ps

```

Function *ode2x2* to define the RHS functions of the ODEs (Equations C.9) is listed below:

```

function yt=ode2x2(t,y)
%
% Global variables
global a b ncall;
%
% ODEs
yt(1)=-a*y(1)+b*y(2);
yt(2)= b*y(1)-a*y(2);
yt=yt';
%
% Increment number of calls to ode2x2
ncall=ncall+1;

```

### Program C.3

Function *ode2x2.m* called by Program C.2

We can note the following points about *ode2x2*:

- The global variables are available for use in calculating the derivatives; also, the counter for derivative evaluations is incremented by 1 each time *ode2x2* is called.
- The derivative row vector is transposed into a column vector, which is the required format for *ode23s* and *ode15s*.

The output from Programs C.2 and C.3 is listed in abbreviated form below (the output for  $t = 2, 3, \dots, 9$  is deleted to reduce the output to reasonable length):

```

mf = 1
case = 1
reltol = 1.00e-004
abstol = 1.00e-004

```

t	y1e	y1	erry1
	y2e	y2	erry2
0.0	0.0000	0.0000	0.0000000000
	2.0000	2.0000	0.0000000000
1.0	0.3679	0.3684	-0.0004764099
	0.3679	0.3684	-0.0004764099



```

      .
      .
      .
10.0  0.0000  0.0000 -0.0000030593
      0.0000  0.0000 -0.0000030594

```

```
ncall = 324
```

```
mf = 1
case = 2
reltol = 1.00e-006
abstol = 1.00e-006

```

t	y1e	y1	erry1
	y2e	y2	erry2
0.0	0.0000	0.0000	0.0000000000
	2.0000	2.0000	0.0000000000
1.0	0.3679	0.3678	0.0000421012
	0.3679	0.3678	0.0000420695
	.	.	.
	.	.	.
	.	.	.
10.0	0.0000	0.0000	-0.0000017745
	0.0000	0.0000	-0.0000017747

```
ncall = 1591
```

```
mf = 2
case = 1
reltol = 1.00e-004
abstol = 1.00e-004

```

t	y1e	y1	erry1
	y2e	y2	erry2
0.0	0.0000	0.0000	0.0000000000
	2.0000	2.0000	0.0000000000
1.0	0.3679	0.3680	-0.0001104769
	0.3679	0.3680	-0.0001104769
	.	.	.
	.	.	.
	.	.	.

```

10.0    0.0000    0.0000    0.0000057390
        0.0000    0.0000    0.0000057390

```

```
ncall =    139
```

```

mf = 2
case = 2
reltol = 1.00e-006
abstol = 1.00e-006

```

t	y1e	y1	erry1
	y2e	y2	erry2
0.0	0.0000	0.0000	0.0000000000
	2.0000	2.0000	0.0000000000
1.0	0.3679	0.3679	0.0000002033
	0.3679	0.3679	0.0000002033
	.		.
	.		.
	.		.
10.0	0.0000	0.0000	-0.0000002999
	0.0000	0.0000	-0.0000002999

```
ncall =    233
```

We can note the following points about this output:

- *ode23s* did not meet the error tolerances, even with 1591 calls to *ode2x2*, e.g.,  $1.00e-004$  vs.  $-0.0004764099$ :

```

mf = 1
case = 1
reltol = 1.00e-004
abstol = 1.00e-004

```

t	y1e	y1	erry1
	y2e	y2	erry2
1.0	0.3679	0.3684	-0.0004764099
	0.3679	0.3684	-0.0004764099

```
ncall =    324
```

```

mf = 1
case = 2

```

```
reltol = 1.00e-006
abstol = 1.00e-006
```

t	y1e	y1	erry1
	y2e	y2	erry2
1.0	0.3679	0.3678	0.0000421012
	0.3679	0.3678	0.0000420695

```
ncall = 1591
```

However, we should keep in mind that Program C.1 did not produce solutions that were any better than about 2+ figures, e.g.,  $-0.00183$ , with 1000 derivative evaluations (due to the use of a fixed step, first-order BDF):

t	dy(1)	dy(2)	y(1)	y(2)	erry(1)	erry(2)	iter
1.00000	0.00000	0.00000	0.36971	0.36971	-0.00183	-0.00183	2

- *ode15s* did come close to meeting, or exceeded, the error tolerances, e.g.,  $1.00e-004$  vs.  $-0.0001104769$ :

```
mf = 2
case = 1
reltol = 1.00e-004
abstol = 1.00e-004
```

t	y1e	y1	erry1
	y2e	y2	erry2
1.0	0.3679	0.3680	-0.0001104769
	0.3679	0.3680	-0.0001104769

```
ncall = 139
```

```
mf = 2
case = 2
reltol = 1.00e-006
abstol = 1.00e-006
```

t	y1e	y1	erry1
	y2e	y2	erry2
1.0	0.3679	0.3679	0.0000002033
	0.3679	0.3679	0.0000002033

```
ncall = 233
```

- The number of derivative evaluations by *ode15s* was substantially lower than for *ode23s*, which infers better performance of the higher-order methods in *ode15s*. For *ode23s*:

```
mf = 1
case = 1
reltol = 1.00e-004
abstol = 1.00e-004

ncall = 324
```

```
mf = 1
case = 2
reltol = 1.00e-006
abstol = 1.00e-006

ncall = 1591
```

and for *ode15s*:

```
mf = 2
case = 1
reltol = 1.00e-004
abstol = 1.00e-004

ncall = 139
```

```
mf = 2
case = 2
reltol = 1.00e-006
abstol = 1.00e-006

ncall = 233
```

*ode23s* and *ode15s* vary the order of the BDF method (in accordance with the table of coefficients in Section 1.7) as well as the integration step in attempting to meet the specified error tolerance. Thus, they are *variable step–variable order* implementations of the BDF method; i.e., they perform *h refinement* and *p refinement* simultaneously.

To conclude this appendix, we have observed the effectiveness (superior efficiency) of implicit methods, such as BDF, for stiff ODE problems. This improved performance, however, involves *greater computational complexity* (generally the solution of linear or nonlinear algebraic or transcendental equations (Equations C.2), depending on whether the ODE system is linear

or nonlinear); therefore *implicit methods should be used only if the ODE system is stiff*. Also, we *should not conclude that a nonlinear ODE system is necessarily stiff*, and therefore an implicit integrator is required.

This final discussion suggests a fundamental question: “How do we know if an ODE system is stiff and therefore an implicit integrator should be used?” In the case of linear ODE systems, we can look at the spread in the eigenvalues, as we did for Equations C.9. However, in the case of nonlinear ODEs, eigenvalues are not defined (and therefore cannot be studied for possible stiffness). For this more general case (of nonlinear ODEs), we suggest the following criterion for determining if an implicit integrator should be used:

$$\text{maximum stable step} \ll \text{problem timescale}$$

In other words, observe if the ODE problem timescale is much greater than the largest integration step ( $h$ ) that can be taken while still maintaining a stable solution, i.e., this suggests that stability is the limiting condition on  $h$ , and therefore the ODE system is effectively stiff so that an implicit integrator should be used.

To illustrate the application of this criterion, for the 2x2 ODE system of Equations C.9 with  $a = 500000.5$ ,  $b = 499999.5$ , and  $\lambda_1 = -(a - b) = -1$ ,  $\lambda_2 = -(a + b) = -1,000,000$ , we found in Section 1.7 that (1) the maximum step for a stable explicit solution is  $2/1,000,000$ , and (2) the timescale for the ODE system is 10 (so that  $e^{-1(10)}$  has decayed to insignificance). Thus, application of the preceding criterion gives

$$\frac{2}{1,000,000} \ll 10$$

which implies that an implicit integrator should be used.

To confirm the preceding analysis, Programs c.2 and c.3 were executed with the two MATLAB explicit integrators, *ode23* and *ode45*. This was easily accomplished by changing the following lines:

```
%
% Call ODE integrator
options=odeset('RelTol',reltol,'AbsTol',abstol);
if(mf==1) [t,y]=ode23s('ode2x2',tout,y0,options); end
if(mf==2) [t,y]=ode15s('ode2x2',tout,y0,options); end

to

%
% Call ODE integrator
options=odeset('RelTol',reltol,'AbsTol',abstol);
if(mf==1) [t,y]=ode23('ode2x2',tout,y0,options); end
if(mf==2) [t,y]=ode45('ode2x2',tout,y0,options); end
```

The resulting change in the numbers of calls to the ODE routine *ode2x2* is summarized below:

Integrator	Tolerance	Calls to <i>ode2x2</i>
<i>ode23s</i>	$10^{-4}$	324
<i>ode23</i>	$10^{-4}$	11, 939, 197
<i>ode23s</i>	$10^{-6}$	1591
<i>ode23</i>	$10^{-6}$	11, 939, 440
<i>ode15s</i>	$10^{-4}$	139
<i>ode45</i>	$10^{-4}$	19, 283, 629
<i>ode15s</i>	$10^{-6}$	233
<i>ode45</i>	$10^{-6}$	19, 283, 713

Clearly, the MATLAB stiff integrators, and even the basic BDF integrator of Program C.1, are substantially more efficient than the MATLAB explicit (nonstiff) integrators (and the same conclusion would be true for the explicit integrators discussed in [Chapters 1 and 2](#)).

However, we now have two additional questions to answer in applying the preceding criterion for stiffness (involving the maximum integration step and problem timescale):

1. How do we determine the maximum integration step for an explicit integrator that still produces a stable solution? Answer: In general, by trial and error using a computer program with an explicit integrator. Or, if the computer run times for a stable explicit solution are large, or an excessive number of derivative evaluations is required to maintain a stable solution, an implicit integrator may possibly be used to good advantage.
2. How do we determine the timescale for the ODE problem? Answer: Either from some knowledge of the characteristics of the problem such as physical reasoning, or again, by trial and error to observe when a complete solution appears to have been computed.

In other words, some trial and error with an explicit integrator is generally required. If the computational effort required to compute a complete solution appears to be excessive, switch to an implicit integrator.

Admittedly, this procedure is rather vague (a general, easily applied mathematical test is not available, especially for nonlinear problems), and some trial and error with explicit integrators first, followed possibly by a switch to implicit integrators, may be required (this is the procedure we generally follow for a new ODE problem). The preceding discussion (in this appendix and [Chapters 1 to 5](#)) indicates that a spectrum of ODE/PDE problems can be handled with explicit integrators (e.g., the 1x1 and 2x2 ODE problems, and the linear and nonlinear PDE problems), and that only under the condition of stiffness (or stability constraints) is an implicit integrator required. Thus, some judgment based on direct computational experience is required.

# Appendix D

---

## *Alternative Forms of ODEs*

---

The ODE systems considered previously (in [Chapters 1 to 5](#) and Appendices A to C) all defined the derivatives (RHS of the ODEs) explicitly; that is, only one derivative appeared in each ODE; Equations 1.6 for the 2x2 linear, constant coefficient ODE system are an example (renumbered here as Equations D.1)

$$\begin{aligned}\frac{dy_1}{dt} &= a_{11}y_1 + a_{12}y_2 & y_1(0) &= y_{10} \\ \frac{dy_2}{dt} &= a_{21}y_1 + a_{22}y_2 & y_2(0) &= y_{20}\end{aligned}\tag{D.1}$$

Equations D.1 are an example of *explicit ODEs* (not to be confused with explicit ODE integration algorithms); this designation comes from the characteristic that *each ODE defines one derivative explicitly*.

However, we could consider ODEs in which each ODE contains more than one derivative, e.g.,

$$\begin{aligned}c_{11}\frac{dy_1}{dt} + c_{12}\frac{dy_2}{dt} &= -ay_1 + by_2 \\ c_{21}\frac{dy_1}{dt} + c_{22}\frac{dy_2}{dt} &= by_1 - ay_2 \\ y_1(0) &= y_{10}, \quad y_2(0) = y_{20}\end{aligned}\tag{D.2}$$

or in matrix form

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \cdot \begin{bmatrix} \frac{dy_1}{dt} \\ \frac{dy_2}{dt} \end{bmatrix} = \begin{bmatrix} -a & b \\ b & -a \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$
$$\begin{bmatrix} y_1(0) \\ y_2(0) \end{bmatrix} = \begin{bmatrix} y_{10} \\ y_{20} \end{bmatrix}$$

For the special case  $c_{11} = 1, c_{12} = 0, c_{21} = 0, c_{22} = 1$ , is the *explicit ODE system* (not to be confused with an explicit integrator):

$$\begin{aligned}\frac{dy_1}{dt} &= -ay_1 + by_2 \\ \frac{dy_2}{dt} &= by_1 - ay_2 \\ y_1(0) &= y_{10}, \quad y_2(0) = y_{20}\end{aligned}\tag{D.3}$$

which can be written in matrix form (for the general  $nxn$  case) as

$$\mathbf{I} \frac{d\mathbf{y}}{dt} = \mathbf{A}\mathbf{y}\tag{D.4}$$

where

$\mathbf{I}$  = identity matrix

$\mathbf{A}$  = ODE coefficient matrix

$\mathbf{y}$  = dependent variable vector

Equations D.2 are an example of a *linearly coupled implicit ODE system*, with the coupling matrix  $\mathbf{M}$

$$\mathbf{M} \frac{d\mathbf{y}}{dt} = \mathbf{A}\mathbf{y}\tag{D.5}$$

The term *linearly coupled* comes from the linear coupling (or linear combinations) of the derivatives on the LHS of Equations D.2.

This form of coupled ODEs is common in applications, and therefore library integrators are available to handle such systems. For example, the MATLAB ODE integrators can accept a coupling matrix that is not the identity matrix; *LSODI*,<sup>1,4</sup> *DASSL*,<sup>2,4,5</sup> *RADAU5*,<sup>3,5</sup> and *MEBDFDAE*<sup>5</sup> can also accommodate such coupled ODE systems.

One approach to the solution of Equations D.2 would be to consider them as linear algebraic equations in the derivatives  $dy_1/dt$  and  $dy_2/dt$ . Then these equations can be solved in the usual ways for linear algebraic equations to arrive at the derivatives explicitly. For example, eliminating  $dy_1/dt$  from Equations D.2 gives

$$\frac{dy_2}{dt} = \frac{c_{21}ay_1 - c_{21}by_2 + c_{11}by_1 - c_{11}ay_2}{c_{11}c_{22} - c_{21}c_{12}}\tag{D.6}$$

Higher-order ( $nxn$ ) linearly coupled ODE systems can be uncoupled by Gaussian elimination or any other established method for simultaneous algebraic equations. Of course, this presupposes that the coupling matrix  $\mathbf{M}$  is not singular or ill-conditioned (e.g., that  $c_{11}c_{22} - c_{21}c_{12} \neq 0$  in Equation D.6). If this is the case, which is common in applications, more sophisticated methods must be applied to perform the numerical integration of the equations, as subsequently discussed briefly.



For example, if in Equations D.2  $c_{11} = c_{12} = 1$ ,  $c_{21} = c_{22} = 0$ , the coupling matrix

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \quad (\text{D.7})$$

is singular. Note that the second ODE is actually an *algebraic equation*, i.e.,

$$\begin{aligned} \frac{dy_1}{dt} + \frac{dy_2}{dt} &= -ay_1 + by_2 \\ 0 \frac{dy_1}{dt} + 0 \frac{dy_2}{dt} &= by_1 - ay_2 \end{aligned} \quad (\text{D.8})$$

Thus, Equations D.8 are actually an example of a *differential algebraic* or *DAE* system.

Equation D.5 can be generalized to

$$\mathbf{M}(\mathbf{y}) \frac{d\mathbf{y}}{dt} = \mathbf{A}\mathbf{y} \quad (\text{D.9})$$

where now the coupling matrix  $\mathbf{M}(\mathbf{y})$  is a function of  $\mathbf{y}$ ; thus, Equation D.9 defines a *nonlinearly coupled implicit ODE system*.

Finally, if the ODE system is of the form

$$\mathbf{f}\left(\mathbf{y}, \frac{d\mathbf{y}}{dt}, t\right) = \mathbf{0} \quad (\text{D.10})$$

Equation D.10 is a *fully implicit ODE system*; it is also frequently designated as a *DAE system* since some of the equations defined by  $\mathbf{f}$  can be algebraic.

All of the preceding ODEs are a special case of Equation D.10 (depending on the form of  $\mathbf{f}$  in Equation D.10). Library integrators are available, in principle, for all of the preceding forms of ODE systems. In particular, *DASSL*<sup>2,4</sup> and *RADAU5*<sup>3</sup> will accommodate Equations D.10 for certain cases of coupling between the ODEs and algebraic equations. The solution of Equation D.10 for completely general forms of  $\mathbf{f}$  is still an open and active area of research.

In summary, we list these alternate forms of ODEs (beyond the explicit ODEs illustrated by Equations D.4 to indicate that (1) all of these forms occur in applications in science and engineering, and (2) library solvers are available for most of these forms (but without a guarantee for the successful calculation of an accurate solution, especially for Equation D.10, depending on the form of  $\mathbf{f}$ ).

A detailed discussion of these alternate ODE forms and the available solvers is beyond the scope of this book. However, the following mathematical software libraries are a good starting point and source of solvers: *netlib*,<sup>4</sup> *gams*,<sup>4</sup> and *mebdfdae*.<sup>5</sup>

---

## References

1. Hindmarsh, A.C., ODEPACK, a systematized collection of ODE solvers, in *Scientific Computing*, R.S. Stepleman et al., Eds., North-Holland, Amsterdam, 1983, 55–64.
2. Brenan, K.E., S.L. Campbell, and L.R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, SIAM, Philadelphia, 1996.
3. Hairer, E., and G. Wanner, *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, Springer-Verlag, Berlin, 1991.
4. Library mathematical software is available from  
<http://www.netlib.org/index.html>; <http://gams.nist.gov>.
5. Library ODE/DAE integrators are available from  
<http://hilbert.dm.uniba.it/~testset/software.htm>

# Appendix E

---

## *Spatial $p$ Refinement*

---

In Section 4.1 and Section 5.1 we considered the numerical integration of a PDE (Equation 4.1) in which the finite difference approximation of the second-order spatial derivative  $\partial u^2/\partial x^2$  was programed in *derv*. Since first- and second-order derivatives in space, with their associated boundary conditions, are so commonplace in applications, the calculation of these derivatives can be facilitated by using library routines. We consider here a few library routines for this purpose.

For example, a *derv* is listed below that can be used in place of the *derv* in Section 4.1 (Program 4.1.3):

```
function [ut]=derv(n,t,u)
%
% Function derv computes the derivative vector
% of the linear PDE problem
%
% Declare global variables
global nsteps ndss;
%
% Problem parameters
xl=0.0;
xu=1.0;
%
% BC at x = 0
u(1)=0.0;
%
% BC at x = 1
u(n)=0.0;
%
```

```

% ux
if ndss == 2 [ux]=dss002(xl,xu,n,u); end
if ndss == 4 [ux]=dss004(xl,xu,n,u); end
%
% uxx
if ndss == 2 [uxx]=dss002(xl,xu,n,u); end
if ndss == 4 [uxx]=dss004(xl,xu,n,u); end
if ndss == 42
    nl=1;
    nu=1;
    ux=zeros(n,1);
    [uxx]=dss042(xl,xu,n,u,ux,nl,nu);
end
if ndss == 44
    nl=1;
    nu=1;
    ux=zeros(n,1);
    [uxx]=dss044(xl,xu,n,u,ux,nl,nu);
end
%
% pdelin
for i=1:n
    ut(i)=uxx(i);
end

```

### Program E.1

*deriv* for the solution of Equations 4.1 to 4.4

We can note the following points about *deriv*:

- After setting the boundary values in  $x$ ,  $x_l = 0.0$ ,  $x_u = 1.0$ , boundary conditions (Equations 4.3 and 4.4) are programmed:

```

%
% BC at x = 0
u(1)=0.0;
%
% BC at x = 1
u(n)=0.0;

```

Note that here we have zeroed the dependent variables,  $u(1)$  and  $u(n)$ , rather than the time derivatives,  $ut(1)$  and  $ut(n)$  as in Program 4.1.3. This difference is due to the way the PDE, Equation 4.1, is programmed at the end of *deriv*, as explained below.

- The first derivative  $\partial u / \partial x$  is computed by one of two spatial differentiation (DSS) routines, *dss002* or *dss004*:

```
%
% ux
if ndss == 2 [ux]=dss002(xl,xu,n,u); end
if ndss == 4 [ux]=dss004(xl,xu,n,u); end
```

$dss002$  implements three point ( $O(\Delta x^2)$ ) finite difference approximations for  $\partial u/\partial x$  whereas  $dss004$  implements five point ( $O(\Delta x^4)$ ) approximations for this derivative. The vector of dependent variables to be differentiated,  $u$ , is an input to  $dss002$  and  $dss004$ , and  $\partial u/\partial x$  is returned in the vector  $ux$ . The choice of the differentiator is through  $ndss$ , which is set in *intpar* ( $ndss$  is added as another parameter to *intpar* of Program 4.1.1, with  $ndss = 2$  to call  $dss002$  and  $ndss = 4$  to call  $dss004$ ).

- The second derivative,  $\partial^2 u/\partial x^2$ , is then computed by differentiating the first derivative  $\partial u/\partial x$  (i.e., by *stagewise differentiation*):

```
%
% uxx
if ndss == 2 [uxx]=dss002(xl,xu,n,ux); end
if ndss == 4 [uxx]=dss004(xl,xu,n,ux); end
```

The second derivative is returned in vector  $uxx$ .

- For  $ndss = 42$  or  $ndss = 44$ , two differentiation routines,  $dss042$  and  $dss044$ , are called that calculate the second derivative,  $\partial^2 u/\partial x^2$ , directly:

```
if ndss == 42
    nl=1;
    nu=1;
    ux=zeros(n,1);
    [uxx]=dss042(xl,xu,n,u,ux,nl,nu);
end
if ndss == 44
    nl=1;
    nu=1;
    ux=zeros(n,1);
    [uxx]=dss044(xl,xu,n,u,ux,nl,nu);
end
```

$nl = 1$  and  $nu = 1$  specify that Dirichlet boundary conditions are used (according to Equations 4.3 and 4.4); if  $nl = 2$  and/or  $nu = 2$ , Neumann boundary conditions are used. For the latter, the first derivative,  $ux$ , is required and it is therefore an input to  $dss042$  and  $dss044$ . In the present case (with Dirichlet boundary conditions), the first derivative is not required and it is therefore zeroed.

- The finite difference approximations in *dss042* and *dss044* are  $O(\Delta x^2)$  and  $O(\Delta x^4)$ , respectively. For example, for *dss042*, the calculation of the second derivative  $\partial^2 u / \partial x^2$  is done with the following code (taken from *dss042*):

```
%...
%...  Grid spacing
      dx=(xu-xl)/(n-1);
%...
%...  Calculate uxx at the left boundary, without ux
      if nl==1
          uxx(1)=( (    2.) *u(  1)...
                    +(   -5.) *u(  2)...
                    +(    4.) *u(  3)...
                    +(   -1.) *u(  4))/(dx^2);
%...
%...  Calculate uxx at the left boundary, including ux
      elseif nl==2
          uxx(1)=( (   -7.) *u(  1)...
                    +(    8.) *u(  2)...
                    +(   -1.) *u(  3))/(2.*dx^2)...
                    +(   -6.) *ux(  1)/(2.*dx);
      end
%...
%...  Calculate uxx at the right boundary, without ux
      if nu==1
          uxx(n)=( (    2.) *u(n  )...
                    +(   -5.) *u(n-1)...
                    +(    4.) *u(n-2)...
                    +(   -1.) *u(n-3))/(dx^2);
%...
%...  Calculate uxx at the right boundary, including ux
      elseif nu==2
          uxx(n)=( (   -7.) *u(n  )...
                    +(    8.) *u(n-1)...
                    +(   -1.) *u(n-2))/(2.*dx^2)...
                    +(    6.) *ux(n  )/(2.*dx);
      end
%...
%...  Calculate uxx at the interior grid points
      for i=2:n-1
          uxx(i)=(u(i+1)-2.*u(i)+u(i-1))/dx^2;
      end
```

This code involves several finite difference approximations at the boundaries for Dirichlet and Neumann boundary conditions. To keep this

discussion to reasonable length, the details are not discussed (they are available in Reference 1). However, the calculation of the second derivative at the interior points by

```
%...
%... Calculate uxx at the interior grid points
      for i=2:n-1
          uxx(i)=(u(i+1)-2.*u(i)+u(i-1))/dx^2;
      end
```

directly parallels the coding in *deriv* of Program 4.1.3:

```
%
% Interior points
dx=(xu-xl)/(n-1);
dxs=dx*dx;
for i=2:n-1
    ut(i)=(u(i+1)-2.0*u(i)+u(i-1))/dxs;
end
```

- The corresponding code in *dss044* for the second derivative at the interior points is

```
%...
%...      i = 3, 4, ..., n-2
      for i=3:n-2
          uxx(i)=r12dxs*...
              (    -1.0*u(i-2)...
                +16.0*u(i-1)...
                -30.0*u(i)  )...
                +16.0*u(i+1)...
                -1.0*u(i+2));
```

Note that five points (or values of  $u$ ) are used to calculate the second derivative, while in *dss042*, only three points are used. This explains the greater accuracy for *dss044* ( $O(\Delta x^4)$  for *dss044* and  $O(\Delta x^2)$  for *dss042*). Again, the details of the finite difference approximations in *dss042* and *dss044* are given in Reference 1.

- Finally, the PDE, Equation 4.1 is programmed:

```
%
% pdelin
      for i=1:n
          ut(i)=uxx(i);
      end
```

The close correspondence between this coding and the PDE is clear; this is possible through the use of the vector  $uxx$  computed by the preceding differentiation ( $DSS$ ) routines.

Functions *inital* of Program 4.1.2 and *fprint* of Program 4.1.4 are essentially the same (the value of  $ndss$  is printed in *fprint*).

The output from this combination of functions, plus a main program that is also the same as Program 3.1.1, is discussed below. Since this output is quite voluminous, only a summary with key points is given. For  $ndss = 2$ ,  $n = 21$ , the output is as follows:

euler2a integrator

ncase = 1    neqn = 21    nsteps = 250    ndss = 2

t	u(num)	u(exact)	diff
0.00	1.000000	1.000000	0.0000e+000
0.20	0.141180	0.138911	2.2689e-003
0.40	0.019932	0.019296	6.3550e-004
0.60	0.002814	0.002680	1.3350e-004
0.80	0.000397	0.000372	2.4930e-005
1.00	0.000056	0.000052	4.3644e-006

euler2b integrator

ncase = 2    neqn = 21    nsteps = 250    ndss = 2

t	u(num)	u(exact)	diff
0.00	1.000000	1.000000	0.0000e+000
0.20	0.141232	0.138911	2.3208e-003
0.40	0.019940	0.019296	6.4378e-004
0.60	0.002817	0.002680	1.3634e-004
0.80	0.000398	0.000372	2.5813e-005
1.00	0.000056	0.000052	4.5242e-006

rkc4a integrator

ncase = 3    neqn = 21    nsteps = 250    ndss = 2

t	u(num)	u(exact)	diff
0.00	1.000000	1.000000	0.0000e+000
0.20	0.141177	0.138911	2.2661e-003
0.40	0.019931	0.019296	6.3470e-004



0.60	0.002814	0.002680	1.3333e-004
0.80	0.000397	0.000372	2.4898e-005
1.00	0.000056	0.000052	4.3588e-006

rkc4b integrator

ncase = 4    neqn = 21    nsteps = 250    ndss = 2

t	u(num)	u(exact)	diff
0.00	1.000000	1.000000	0.0000e+000
0.20	0.141177	0.138911	2.2661e-003
0.40	0.019931	0.019296	6.3471e-004
0.60	0.002814	0.002680	1.3333e-004
0.80	0.000397	0.000372	2.4898e-005
1.00	0.000056	0.000052	4.3589e-006

rkf45a integrator

ncase = 5    neqn = 21    nsteps = 250    ndss = 2

t	u(num)	u(exact)	diff
0.00	1.000000	1.000000	0.0000e+000
0.20	0.141177	0.138911	2.2661e-003
0.40	0.019931	0.019296	6.3470e-004
0.60	0.002814	0.002680	1.3333e-004
0.80	0.000397	0.000372	2.4898e-005
1.00	0.000056	0.000052	4.3588e-006

rkf45b integrator

ncase = 6    neqn = 21    nsteps = 250    ndss = 2

t	u(num)	u(exact)	diff
0.00	1.000000	1.000000	0.0000e+000
0.20	0.141177	0.138911	2.2661e-003
0.40	0.019931	0.019296	6.3470e-004
0.60	0.002814	0.002680	1.3333e-004
0.80	0.000397	0.000372	2.4898e-005
1.00	0.000056	0.000052	4.3587e-006

For this case ( $n = 21$ ), the errors are relatively large, e.g., for *rkf45a* at  $t = 0.20$  the error at  $x = 0.5$  is  $2.2661e-003$ :

rkf45a integrator

ncase = 5    neqn = 21    nsteps = 250    ndss = 2

t	u(num)	u(exact)	diff
0.00	1.000000	1.000000	0.0000e+000
0.20	0.141177	0.138911	2.2661e-003

Thus, we would expect that as more points are added (*h refinement*) and a higher-order spatial differentiator is used (*dss044* in place of *dss042*), which is a form of *p refinement* in space, the accuracy of the solution would improve. This improvement is demonstrated in the following summary of results from *rkf45a* at  $t = 0.2$ :

ncase = 5    neqn = 21    nsteps = 250    ndss = 2  
0.20    0.141177    0.138911    2.2661e-003

ncase = 5    neqn = 31    nsteps = 250    ndss = 2  
0.20    0.139828    0.138911    9.1682e-004

ncase = 5    neqn = 41    nsteps = 250    ndss = 2  
0.20    0.139476    0.138911    5.6448e-004

ncase = 5    neqn = 21    nsteps = 500    ndss = 4  
0.20    0.138921    0.138911    9.6684e-006

ncase = 5    neqn = 31    nsteps = 500    ndss = 4  
0.20    0.138913    0.138911    2.1371e-006

ncase = 5    neqn = 41    nsteps = 500    ndss = 4  
0.20    0.138912    0.138911    6.5075e-007

ncase = 5    neqn = 21    nsteps = 500    ndss = 42  
0.20    0.139476    0.138911    5.6448e-004

ncase = 5    neqn = 31    nsteps = 500    ndss = 42  
0.20    0.139162    0.138911    2.5071e-004

ncase = 5    neqn = 41    nsteps = 1000    ndss = 42  
0.20    0.139052    0.138911    1.4099e-004

ncase = 5    neqn = 21    nsteps = 500    ndss = 44  
0.20    0.138913    0.138911    1.6930e-006

```
ncase = 5    neqn = 31    nsteps = 500    ndss = 44
0.20      0.138911      0.138911      3.5667e-007
```

```
ncase = 5    neqn = 41    nsteps = 1000    ndss = 44
0.20      0.138911      0.138911      1.1461e-007
```

We can note the following points about this output:

- The number of steps in  $t$  to maintain stability changed through the values 250, 500, 1000. Generally this number had to be increased with increasing numbers of grid points,  $neqn$  (as  $h$  refinement was used), and increasing order of the finite difference approximations (as  $p$  refinement was used).
- Generally, the *order conditions* were maintained when changing the number of grid points,  $neqn$ . For example, with  $ndss = 44$  ( $dss044$  was called),  $O(\Delta x^4)$ , and comparing the  $neqn = 21$  and  $neqn = 41$  solutions,

$$1.6930 \times 10^{-6} (1/2)^4 = 1.0581 \times 10^{-7} \simeq 1.1461 \times 10^{-7}$$

These order conditions will not be maintained exactly because of the additional error introduced by the integration in  $t$ , i.e., errors occur because of the approximate spatial and time discretizations. Note that  $nsteps = 500$  and  $1000$  in the preceding solutions so that the change in the number of steps in  $t$  complicates the comparisons in  $x$ .

- Also, the order conditions were maintained when changing the order of the approximations, e.g.,  $dss042$  (with  $O(\Delta x^2)$ ) to  $dss044$  (with  $O(\Delta x^4)$ ). For example, for  $neqn = 21$ ,

$$5.6448 \times 10^{-4} \frac{(1/20)^4}{(1/20)^2} = 1.4112 \times 10^{-6} \simeq 1.6930 \times 10^{-6}$$

This result clearly indicates the advantage of using higher-order approximations, e.g.,  $O(\Delta x^4)$  rather than  $O(\Delta x^2)$ .

Finally, in addition to the routines for  $\partial u / \partial x$  and  $\partial^2 u / \partial x^2$ , i.e., three point ( $O(\Delta x^2)$ ) and five point ( $O(\Delta x^4)$ ) approximations in the six languages, routines with seven point ( $O(\Delta x^6)$ ), nine point ( $O(\Delta x^8)$ ) and 11 point ( $O(\Delta x^{10})$ ) approximations in Fortran and MATLAB are available from the authors; the translation of these routines to the other four languages is straightforward.

---

## References

1. Schiesser, W.E. *The Numerical Method of Lines Integration of Partial Differential Equations*, Academic Press, San Diego, CA, 1991.

# Appendix F

---

## *Testing ODE/PDE Codes*

---

The development of a new ODE/PDE application typically requires some trial-and-error analysis until the application is running correctly. The errors that can occur during this development process are generally of two types:

1. *Compiler errors* resulting from incorrect syntax for the particular compiler (language)
2. *Execution errors* after successful compilation of the source code

We cannot give any specific help with compiler errors since they must be corrected by repeated attempts at compiling the source code until the compiler accepts the source code with no reported errors. The success in the elimination of compiler errors is directly tied to the programmer's experience with the language.

Execution errors can generally occur for two reasons:

1. The ODE/PDE mathematical model has fundamental flaws that eventually cause arithmetic problems. For example, if the model involves division by zero, or if the model equations are unstable so that eventually the calculations cause an overflow (the calculations produce numbers that exceed the largest number the computer can handle), an execution error will eventually be reported.
2. Errors were made in the programming of the model equations; this could be something as simple as a sign error.

Execution errors are generally the more difficult to correct (compiler errors will be rather explicit and can generally be corrected by reading the compiler error messages and making corrections in the source code). Thus, we present here a method for detecting and correcting execution errors when developing an ODE/PDE application.

Here, then, are the steps that should be most relevant to finding execution errors:

- We assume that some form of numerical integration of an ODE system is a fundamental part of the calculations and the associated source code (PDEs will generally be approximated as a system of ODEs by the method of lines (MOL) as discussed in [Chapters 4 and 5](#)).
- Next, we assume that the execution errors are most likely the result of errors in programming the RHSs of the ODEs. Thus, we concentrate on the programming of the ODEs to look for execution errors.
- In all the programming of initial value ODEs we have considered previously, a vector of dependent variables is to be computed (starting with the vector defined as initial conditions). Thus, the principal output of the calculations is the vector of dependent variables as a function of the independent variable (which is the solution to a system of ODEs), and we therefore examine the vector of dependent variables in detail *to look for results that are obviously in error, e.g., some “bad numbers.”*
- However, the vector of dependent variables is generated by numerically integrating the associated vector of derivatives. So we also should examine in detail the vector of derivatives computed from the RHSs of the ODEs. In fact, it is the vector of derivatives that determines the vector of dependent variables (i.e., that determines the solution). In this sense, the derivatives are as interesting as the dependent variables since they define the solution (through the ODEs).
- To organize these ideas, we can output the dependent variable vector at the beginning of the derivative code or routine (since this vector defines the vector of derivatives through the ODEs).
- We can then output the vector of derivatives at the end of the derivative calculations (the end of the code for the ODEs) to look for possible errors in the programming of the ODEs.

To illustrate this procedure for finding the source of execution errors, we consider again the  $2 \times 2$  linear ODE system of Equations 1.6. Function *inital* now includes initialization of a counter for the number of times the derivative routine is called:

```
function [u0]=inital(n,t)
%
% Function inital sets the initial condition vector
% for the 2 x 2 ODE problem
%
% Define global variables
global ncall;
%
```

```
% Initialize counter for calls to the derivative
% routine (derv)
ncall=0;
%
% Initial condition vector
u0(1)=0;
u0(2)=2;
```

### **Program F.1**

*inital* with a counter for the derivative evaluations

Note that *ncall* is declared *global* so that it can be passed to other routines, in this case *derv*.

```
function [ut]=derv(n,t,u)
%
% Function derv computes the derivative vector
% of the 2 x 2 ODE problem
%
% Define global variables
global ncall;
%
% Heading during first call to derv
if ncall == 0
    fprintf('\n\n          Output from derv\n\n');
    fprintf('          t          u(1)          u(2)\n');
    fprintf('          ut(1)          ut(2)\n\n');
end
%
% Display dependent variable vector
fprintf('%10.4f%10.4f%10.4f\n',t,u(1),u(2));
%
% Problem parameters
a=5.5;
b=4.5;
%
% Derivative vector
ut(1)=-a*u(1)+b*u(2);
ut(2)= b*u(1)-a*u(2);
%
% Display derivative vector
fprintf('%20.4f%10.4f\n\n',ut(1),ut(2));
%
% Increment counter for calls to derv
ncall=ncall+1;
```

```
%
% Terminate execution after five calls to derv
if ncall==5 dbstop in derv; end
```

## Program F.2

*derv* with output of the dependent variable vector and derivative vector

We can note the following points about *derv*:

- The first time *derv* is executed (with *ncall* = 0), a heading for the output from *derv* is displayed:

```
%
% Heading during first call to derv
if ncall == 0
    fprintf('\n\n          Output from derv\n\n');
    fprintf('          t          u(1)          u(2)\n');
    fprintf('          ut(1)         ut(2)\n\n');
end
```

- Since *derv* has the central function of using the dependent variable vector, in this case  $[u(1) \ u(2)]^T$ , to compute the derivative vector  $[ut(1) \ ut(2)]^T$ , two output (*fprintf*) statements are used:

- The dependent variable vector is displayed at the beginning of *derv* so that the state of the dependent variables coming into *derv* can be judged. If they do not look reasonable, then there is probably an error in the numerical integration; e.g., perhaps some of the dependent variables are moving in the wrong direction (due possibly to an error in the programming of the derivatives that follows in *derv*), or are becoming large due to an instability in the numerical integration (for example, if an explicit ODE integrator is being used and the step *h* is too large).

```
%
% Display dependent variable vector
fprintf('%10.4f%10.4f%10.4f\n',t,u(1),u(2));
```

- The derivative vector is then programmed (the central function of *derv*) using the dependent variables, i.e., the RHSs of the ODEs are programmed. Once all of the derivatives are computed, they are displayed by an output statement (generally close to the end of the derivative routine)

```
%
% Derivative vector
ut(1)=-a*u(1)+b*u(2);
ut(2)= b*u(1)-a*u(2);
```

```
%
% Display derivative vector
fprintf('%20.4f%10.4f\n\n',ut(1),ut(2));
```

The purpose of this output statement for the derivatives is to check if the derivatives appear reasonable (not of the wrong sign, e.g., a dependent variable that should be decreasing should have a negative derivative or too large because units in the model equations are not correct or the numerical integration is becoming unstable).

- Once the input dependent variable vector and output derivative vector are checked, the number of times the output appears should be limited (because the derivative routine typically can be called hundreds or thousands of times during the computation of a complete numerical solution to an ODE problem). In this case, execution of the program is terminated when *ncall* reaches a value of 5 (five calls to *derv*):

```
%
% Increment counter for calls to derv
ncall=ncall+1;
%
% Terminate execution after five calls to derv
if ncall==5 dbstop in derv; end
```

Note that the MATLAB command *dbstop* is used when *ncall* = 5 to terminate execution.

The output from *derv* is listed below:

Output from derv		
t	u(1)	u(2)
	ut(1)	ut(2)
0.0000	0.0000	2.0000
	9.0000	-11.0000
0.0100	0.0900	1.8900
	8.0100	-9.9900
0.0100	0.0850	1.8950
	8.0600	-10.0400
0.0200	0.1656	1.7946
	7.1649	-9.1251
0.0200	0.1612	1.7992
	7.2101	-9.1704



Not too surprisingly, the output looks reasonable (this is a small, simple ODE problem). As expected,  $u_1(t)$  has a positive derivative (it increases from the initial condition  $u_1(0) = 0$ ) and  $u_2(t)$  has a negative derivative (it decreases from the initial condition  $u_2(0) = 2$ ). If these signs in the derivatives were not observed, we would know something is wrong, probably in the preceding programming of the derivatives.

Also, note that the initial conditions can be checked (in the  $t = 0$  output). This is an important check (the solution must start out at the right values). Although this check is obvious in this small problem (there are only two initial conditions,  $u_1(0) = 0$ ,  $u_2(0) = 2$ ), for larger problems, e.g., hundreds or thousands of ODEs, overlooking even one initial condition or using an incorrect value will most likely guarantee that the numerical solution will be incorrect.

The same is true for the derivative calculations in *deriv*. If we are integrating  $n$  first-order ODEs, we need to program  $n$  derivatives. If we overlook even one derivative, which is easy to do in a large ODE system, the solution will be incorrect. Thus, at the end of *deriv* we must have  $n$  good numerical values for the derivative vector.

Also, intermediate calculations before the final calculation of the derivatives are quite common. For example, we might have to solve a set of nonlinear equations, using the ODE dependent variables as inputs. Once these intermediate variables, such as from the solution of a set of nonlinear equations, are computed in *deriv*, they can be used in the calculation of the derivatives. Of course, errors can occur in these intermediate calculations, and they can be checked by using additional output statements.

Generally, a complete output of all of the variables used in the calculation of the derivative vector can be included in *deriv* to ensure that the calculations are done correctly (according to the model equations). This is particularly true for PDE systems in which spatial (boundary value) derivatives are computed in the method of lines. These spatial derivatives can be displayed from *deriv* to check their calculation.

Eventually, after detailed checking of the output from *deriv*, the derivative vector (that is the input to the ODE integrator) will be correct; the output statements in *deriv* and the derivative counter can then be removed to compute a complete solution (or these statements can be “commented out” in case they have to be subsequently reactivated for more checking).

Although the preceding method of checking the initial derivative calculations is usually effective in finding and correcting errors, it will not be effective for the case when the initial calculations appear to be correct, but later, the numerical solution develops an obvious problem; e.g., the dependent variables become excessively large, or the compiler reports a *NAN* (not a number). This may be due to integrator instability, or possibly to an error that grows slowly, but eventually causes a calculational failure. The difficulty in finding the cause of such problems stems from the uncertainty in knowing when it occurs (in order to produce some output from *deriv*, for example), and why. That is why

some knowledge of how numerical ODEs integrators function is important, such as issues of stability, and error monitoring and step size control. For the latter, a common source of failures is to specify unreasonable error tolerances, as discussed in [Chapter 1](#). But in any case, there is no substitute for thorough testing and scrutiny of the computed output.

Finally, two other points can be considered to assist in the checking of computer codes for ODE/PDE problems:

First, we can consider the question of the units of the timescale, e.g., in an application, when we compute the dependent variables of an ODE system as a function of the independent variable  $t$ , is the timescale (units of  $t$ ) in microseconds, milliseconds, seconds, hours, days, years, etc.? The answer generally is that the units of the timescale are the same as the units of the derivative vector. In other words, the derivatives will have the units of reciprocal time, and whatever those units are will be the units of the timescale.

Additionally, the time units must be the same for all of the computed derivatives. Thus, we cannot calculate the derivative for one dependent variable with the units of seconds, and another derivative with the units of hours. The simultaneous integration of the two derivatives will lead to an incorrect solution. To avoid this problem, the time units of all of the derivatives should be checked to ensure that they are the same.

The same reasoning can be applied to the units of the dependent variables. For example, if an energy balance produces a time derivative with the units of K/min, integration of this derivative will produce a solution with the temperature in K and the timescale in min.

Second, as a word of advice based on experience, we suggest that the development of a new ODE/PDE application should not necessarily be initiated with all of the details of the mathematical model included. This might seem like an inexplicable approach since we eventually want to solve the model equations with the full and complete details we think are necessary to describe the physical system. However, for a relatively complicated model, putting all the mathematical details into the code at the beginning very often guarantees that the code will not execute correctly. If this occurs, the question then that is often difficult or impossible to answer is "What caused the code to fail?" In other words, there are so many details that could be the source of the calculational failure, the identification of which particular detail(s) are the cause of the failure is difficult (although the testing with intermediate output as described previously can help in identifying the cause of the failure).

As an alternate approach, we suggest that the code be built up a little at a time. For example, very simple ODEs can be coded first which can be numerically integrated and the solution checked. The ODEs might even be coded with constant derivatives (and therefore their solutions should be linear in the independent variable); if the derivatives of some of the dependent variables are set to zero, those dependent variables should remain constant during the testing. Then an ODE can be added while the other dependent variables are computed from derivatives that are constant. The code is then

built by adding more ODEs. As this process continues, if a failure occurs, it must be due to the ODEs or mathematical relationships just added, and these can be examined in detail, or they can be reset to the previous condition for which the code executed and some additional testing can be performed to determine why the last step failed. In this way, mathematical details are added until the entire model is coded. The source of any failure along the way can be identified and corrected, particularly by using detailed output from the derivative routine as discussed previously.

In other words, the development of an ODE/PDE code is an experimental, trial-and-error, evolutionary process. Thus, as much as we might like to proceed directly to the solution of the complete mathematical model with all of the details that we think are relevant and should be included, starting out with a simpler ODE/PDE system that has a solution that can be checked, then extending the system through a series of steps, each of which can be checked before going on to the next step, we think is the best way to arrive at a final working code for the problem system of interest.