Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

RKC: an Explicit Solver for Parabolic PDEs

B.P. Sommeijer, L.F. Shampine, J.G. Verwer

Modelling, Analysis and Simulation (MAS)

MAS-R9715 June 30, 1997

Report MAS-R9715
ISSN 1386-3703

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

RKC: an Explicit Solver for Parabolic PDEs

B.P. Sommeijer
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

L.F. Shampine
Mathematics Department
Southern Methodist University
Dallas, TX 75275-0156, USA

J.G. Verwer
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

The FORTRAN program RKC is intended for the time integration of parabolic partial differential equations discretized by the method of lines. It is based on a family of Runge-Kutta-Chebyshev formulas with a stability bound that is quadratic in the number of stages. Remarkable properties of the family make it possible for the program to select at each step the most efficient stable formula as well as the most efficient step size. Moreover, they make it possible to evaluate the explicit formulas in just a few vectors of storage. These characteristics of the program make it especially attractive for problems in several spatial variables. RKC is compared to the BDF solver VODPK on two test problems in three spatial variables.

1991 Mathematics Subject Classification: 65L06,65M20,65Y20
1991 Computing Reviews Classification System: G.1.7,G.1.8
Keywords and Phrases: Parabolic partial differential equations, numerical software, time integration, Runge-Kutta-Chebyshev solver.
Note: Work carried out under project MAS1.4 - 'Exploratory research: Analysis of PDEs and their Discretization'.

1. Introduction

RKC is a variable step size, variable formula code that uses explicit Runge-Kutta formulas to solve efficiently a class of large systems of mildly stiff ordinary differential equations (ODEs). The systems arising when a parabolic partial differential equation (PDE) is approximated by semi-discretization exemplify the problems for which RKC is intended. To be more specific, let the initial value problem for the ODEs have the form

$$\frac{dU(t)}{dt} = F(t, U(t)), \quad 0 < t \leq T, \quad U(0) = U_0, \tag{1.1}$$

so that the Jacobian matrix is $F'(t, U) = \partial F(t, U)/\partial U$. RKC is intended for problems with Jacobians that are close to normal and that have all their eigenvalues near the negative real axis. These properties are certainly true when $F'(t, U)$ is symmetric and non-positive definite, which is frequently the case when discretizing elliptic operators.

RKC exploits some remarkable properties of a family of explicit Runge-Kutta formulas of the Chebyshev type proposed by van der Houwen and Sommeijer [13]. There is a member of s stages for all $s \geq 2$, and there are analytical expressions for its coefficients. All the formulas have stability regions

that include narrow strips about the negative real axis. The length of the strip, the stability boundary $\beta(s)$, is approximated well by $0.653s^2$. This makes it possible for RKC to solve problems that are mildly stiff with explicit formulas. A very important property is that because of a recursion for Chebyshev polynomials, it is possible to evaluate a formula with just a few vectors of working storage, no matter the number of stages. Most remarkable is that for practical purposes the local errors of all members of the family are the same. This means that the code can estimate first the most efficient step size and then use an estimate of the spectral radius of the Jacobian to determine the most efficient formula for which this step size is stable. Another important property of the family is that it is easy to obtain a continuous extension of excellent quality that is "free". This is especially valuable for a Runge-Kutta formula that might involve a great many stages.

RKC has very modest storage requirements because it uses explicit formulas that can be evaluated by recursion. It requires at most 7 vectors of storage. This makes it attractive for the solution of PDEs in several space variables by semi-discretization. Another advantage of explicit formulas is that vectorization and/or parallelization presents no particular difficulties. The code is, for example, suitable for problems with solutions that are travelling waves because small steps are needed to resolve fronts accurately. Generally reaction-diffusion systems

$$\frac{\partial u}{\partial t} = \nabla \cdot (K \nabla u) + f(u,x,t), \quad u = u(x,t), \quad x \in \Re^d,$$

where f is a modestly stiff reaction term can be solved efficiently with RKC. When f gives rise to severe stiffness, RKC is not recommended. In such cases it can still be useful as part of an operator splitting scheme that treats the reaction part at grid points with a standard code for stiff problems. Likewise, in combination with operator splitting RKC can be useful for systems of transport problems of advection-diffusion-reaction type

$$\frac{\partial u}{\partial t} + \nabla \cdot (a\, u) = \nabla \cdot (K \nabla u) + f(u,x,t), \quad u = u(x,t), \quad x \in \Re^d.$$

Problems of this kind play an important role in the modeling of pollution of the atmosphere, ground water, and surface water, and are the subject of much current research.

Section 2 presents the family of formulas implemented in RKC. The following section discusses the properties of the family that are crucial to the success of the solver and how they are exploited in software. Among the issues discussed are the estimation and control of error, estimation of the spectral radius and control of stability, and a continuous extension. Section 4 presents results for two PDEs in three spatial variables taken from [12]. The last section explains how to obtain a copy of RKC and its auxiliary programs along with examples showing how to use them. The source codes are listed in appendices.

2. RKC's formulas

Historically the principal goal when constructing Runge-Kutta formulas was to achieve the highest order possible with a given number of stages s. Stabilized methods are different in that the principal goal is to construct formulas with regions of absolute stability that are as large as possible in a sense that depends on the intended application. The formulas of RKC are intended for problems like those arising when parabolic PDEs are approximated by semi-discretization. Correspondingly, the goal is to construct formulas that are stable on a strip containing a long segment of the negative real axis. The wider the strip, the greater the applicability of the method, but the most important characteristic of the formula is the length of the segment, the stability boundary $\beta(s)$. For the ODEs of semi-discretization, a low order formula is appropriate because only a modest accuracy is expected of the approximation to the PDE. When the PDE involves more than one spatial variable, the size of the system of ODEs grows rapidly as the mesh spacing is decreased. The relatively crude meshes that are used for this reason lead to relatively large discretization errors in space, hence limits the accuracy that would be meaningful in the time integration and so favors low order methods. It turns out that

the higher order methods require more stages to achieve the same stability, another factor favoring low order formulas. For these reasons all the formulas of RKC are of order two.

The formulas of RKC are given in [17]. To avoid confusion, we point out that they are slightly different from the formulas of [13]. A comprehensive linear stability and convergence analysis of the formulas is found in [20]. The formulas are also studied in the review article [21] along with a number of related methods.

Let U_n denote the approximation to $U(t)$ at $t = t_n$ and let $\tau = t_{n+1} - t_n$ be the step size in the current step from t_n to t_{n+1}. The formulas of RKC have the form

$$\begin{aligned} Y_0 &= U_n, \\ Y_1 &= Y_0 + \tilde{\mu}_1 \tau F_0, \\ Y_j &= (1 - \mu_j - \nu_j) Y_0 + \mu_j Y_{j-1} + \nu_j Y_{j-2} + \tilde{\mu}_j \tau F_{j-1} + \tilde{\gamma}_j \tau F_0, \quad j = 2, \ldots, s, \\ U_{n+1} &= Y_s. \end{aligned} \tag{2.1}$$

All the coefficients are available in analytical form for arbitrary $s \geq 2$. They are defined as follows. Let T_j be the Chebyshev polynomial of the first kind of degree j. Then

$$\epsilon = 2/13, \quad w_0 = 1 + \epsilon/s^2, \quad w_1 = \frac{T_s'(w_0)}{T_s''(w_0)}, \quad b_j = \frac{T_j''(w_0)}{(T_j'(w_0))^2} \quad (2 \leq j \leq s), \quad b_0 = b_2, \quad b_1 = b_2$$

and

$$\tilde{\mu}_1 = b_1 w_1, \;\; \mu_j = \frac{2 b_j w_0}{b_{j-1}}, \;\; \nu_j = \frac{-b_j}{b_{j-2}}, \;\; \tilde{\mu}_j = \frac{2 b_j w_1}{b_{j-1}}, \;\; \tilde{\gamma}_j = -(1 - b_{j-1} T_{j-1}(w_0)) \tilde{\mu}_j \;\; (2 \leq j \leq s).$$

In (2.1) the stage $F_j = F(t_n + c_j \tau, Y_j)$. The c_j are

$$c_j = \frac{T_s'(w_0)}{T_s''(w_0)} \frac{T_j''(w_0)}{T_j'(w_0)} \approx \frac{j^2 - 1}{s^2 - 1} \quad (2 \leq j \leq s - 1), \quad c_1 = \frac{c_2}{T_2'(w_0)} \approx \frac{c_2}{4}, \quad c_s = 1.$$

The approximations show that the arguments $t_n + c_j \tau$ all lie within the span of the step to $t_n + \tau$.

3. Software issues

RKC is the result of both software and algorithmic development of Sommeijer's code [18]. Broadly speaking, the implementation is like that of any modern code based on an explicit Runge-Kutta formula. In this section we describe briefly aspects of the code that are unusual or even unique. Any modern general-purpose code for initial value problems will estimate the local error at each step and adjust the step size both to control this error and to solve the problem efficiently. Popular Adams, BDF, and extrapolation codes also select the formula dynamically. The main difficulty in selecting the most efficient formula is in estimating the step size that could be used with a formula other than the one used to take the step. The family of formulas implemented in RKC has the remarkable property that for practical purposes, all the formulas have the same accuracy. The stability boundary of the formulas increases quadratically with the number of stages. By computing an estimate of the spectral radius, the code is able to determine the most efficient formula that is stable with a step size predicted to yield the desired accuracy. An important property of the family is that it is possible to evaluate a formula using just a few vectors of working storage, no matter how large the number of stages. Still another important property is that it is easy to obtain a continuous extension of excellent quality.

Error control For a smooth F in (1.1), a Taylor series expansion of the local solution at $t = t_n$ results in

$$U_{n+1} = U + \tau \dot{U} + 1/2 \, \tau^2 \ddot{U} + C_{31,s} \, \tau^3 \, F_j F_k^j F^k + C_{32,s} \, \tau^3 \, F_{jk} F^j F^k + O(\tau^4), \quad s \geq 2.$$

Naturally the coefficients $C_{31,s}$ and $C_{32,s}$ depend on the formula, i.e., on the number of stages s. However, it is found that both tend rapidly to a constant value as s increases. Indeed, they are both close to 1/10 for *all* $s \geq 2$. This says that the leading term of the local error expansion is approximately proportional to the third derivative of the solution. As a consequence, the global error is approximately independent of s. The convergence results of [20] make this precise for linear problems. Extensive testing with both linear and nonlinear problems has confirmed that for practical purposes, the local error is independent of the number of stages for $s \geq 2$.

Let $Le(t_{n+1})$ be the approximation to the leading term of the local error expansion resulting from replacing the true s-dependent constants by their limiting values:

$$Le(t_{n+1}) = 1/15\ \tau^3\, d^3\, U(t_n)/dt^3.$$

The simple form of this expression for the error makes it easy to obtain an asymptotically correct estimate:

$$Est_{n+1} = 1/15\ [12(U_n - U_{n+1}) + 6\tau(F(U_n) + F(U_{n+1}))].$$

At each step the estimated local error is controlled so that accuracy tolerances specified by the user are met. There is a scalar relative error tolerance *rtol*. The user must ask for some relative accuracy, but not too much for the precision available. Because the formulas are of order two, the code is not appropriate for stringent tolerances. The absolute error tolerances can be supplied in the form of a scalar *atol* that is applied to all the solution components or as a vector that is applied to corresponding components. A scalar absolute error tolerance is convenient and saves a useful amount of storage, but is appropriate only when all the solution components are on the same scale. These tolerances are used in the weighted RMS norm

$$\|Est_{n+1}\| = \|w^{-1}\, Est_{n+1}\|_2, \quad w = \sqrt{m}\ \mathrm{diag}\,(Tol_1, \ldots, Tol_m),$$

where

$$Tol_k = atol_k + rtol\,|U_{n+1,k}|,$$

m is the dimension of the ODE system and $U_{n+1,k}$ the $k-$th component of U_{n+1}. Hence the step is accepted if $\|Est_{n+1}\| \leq 1$ and otherwise rejected and redone. The error is controlled by an error per step criterion, so if all is going well, the arguments of [16] show that reducing the tolerances by a factor of 0.1 will reduce the error in the numerical solution by a factor of roughly 0.2.

Compared to other Runge-Kutta methods, a failed step in RKC can be expensive in absolute terms because of a large number of stages. Besides this obvious expense, in a common way of using RKC a rejected step causes the spectral radius to be recomputed. A standard device for reducing the number of rejected steps is to use a fraction of the step size predicted to be optimal; a relatively small fraction is used in RKC. Watts [22] uses information gathered at the preceding step to refine the conventional prediction of the optimal step size. Later Gustafsson et al. [6] derived nearly the same algorithm from the completely different viewpoint of control theory. Versions of the algorithm are seen in RKSUITE [1] and RADAU5 [8]. These very successful codes have demonstrated the value of the refined prediction for reducing the number of step failures, so RKC also implements a version of the algorithm. Specifically, the prediction for the new step size after a successful step is given by

$$\tau_{\mathrm{new}} = \min\,(10, \max\,(0.1, fac))\ \tau,$$

with the fraction fac defined by

$$fac = 0.8\left(\frac{\|Est_n\|^{1/(p+1)}}{\|Est_{n+1}\|^{1/(p+1)}}\,\frac{\tau_n}{\tau_{n-1}}\right)\,\frac{1}{\|Est_{n+1}\|^{1/(p+1)}}.$$

The conventional prediction is obtained by deleting the parenthesized term. It is used after a step rejection.

Initial step size For the convenience of the user, RKC determines automatically an initial step size. In modern algorithms for this purpose, the main difficulty [5] is finding a step size that is on scale. Once this is done, a tentative step size can be refined by means of trial steps. The situation in RKC is special in two ways. A tentative step size τ_0 that is on scale is furnished by the reciprocal of the spectral radius that is computed for stability control. Further, the very simple form of the local error allows the error that would be made in a step of size τ to be estimated with a difference quotient [19] at a cost of a single function evaluation:

$$\tau_0 = 1/\sigma(F'(t_0, U_0)), \quad Est = \tau_0(F(t_0 + \tau_0, U_0 + \tau_0 F(t_0, U_0)) - F(t_0, U_0)).$$

The initial step size τ_{start} is taken to be one tenth of the largest step size predicted to satisfy the error test:

$$\tau_{start} = 0.1\frac{\tau_0}{||Est||^{1/2}}.$$

Absolute stability At each step RKC first selects the "optimal" step size for controlling the local error and then selects a formula for which this step size is absolutely stable. Roughly speaking, the absolute stability regions of the formulas used are strips containing a segment of the negative real axis, c.f. [21], and the length of the segment $\beta(s)$ is approximated well by $0.653s^2$. Assuming that the eigenvalues of local Jacobians lie in such a strip, the spectral radius of the Jacobian is all that is needed to find the smallest number of stages that yields stability for the step size τ:

$$\tau\sigma(F'(t, U)) \leq 0.653s^2. \tag{3.1}$$

Problems with constant Jacobians are sufficiently common that users are asked to identify them; RKC computes the spectral radius only once in such cases.

Sometimes it is easy enough to determine analytically a reasonably close upper bound on the spectral radius, using, e.g., Geršgorin's circle theorem, so RKC allows for this possibility. Generally it is not expensive to evaluate such a bound, so the code invokes it at each successful step.

Commonly RKC estimates the spectral radius automatically using a nonlinear power method. This is convenient for the user, but it does cost another vector of working storage and some computation. The basic idea of the power method is simple, but there are a good many ways the method can degenerate, so considerable care is needed in its implementation. Our implementation takes advantage of the experience reported in [10, 14, 19, 15], and here we describe only points that differ from previous work. An important difference is that it is assumed that the eigenvalues are close to the negative real axis. A Rayleigh quotient is then much more likely to reflect the magnitudes of the largest eigenvalues than in the general case of eigenvalues that might have substantial imaginary part. It is an upper bound on the spectral radius that is needed rather than the spectral radius itself, so the estimate is increased some and it is then used conservatively in selecting the number of stages.

It is important to hold down the cost of computing the spectral radius. The slope of the solution at the beginning of a step (which is always available) is likely to be rich in the directions corresponding to dominant eigenvalues [14], so it is used to start the power method at the first step. We have found it very advantageous to retain the computed eigenvector from one estimation of the spectral radius for use as the starting guess for the next. With such a good guess it is typical that only a few iterations are needed. Still, the Jacobian should change slowly, so it should not be necessary to estimate the spectral radius at every step. The spectral radius is estimated on a step failure because this may indicate a change in the character of the problem. Otherwise, it is estimated every 25 successful steps since the last estimate. Of course, unnecessary estimates are avoided when there are repeated step failures.

Storage The form (2.1) for the formulas of RKC results from the three term recursion relation for Chebyshev polynomials. It could be rewritten in the standard form of an explicit Runge-Kutta formula

of s stages, but (2.1) is much better for computation. One reason will be taken up shortly, but the most important reason is that it is obvious this form of the formula can be evaluated using just a few vectors of working storage, no matter how large the number of stages. The precise amount of storage required by RKC depends on how the code is used, but it never uses more than five vectors of storage for the computation itself. This makes it possible for RKC to solve the very large systems of ODEs arising from semi-discretization of PDEs in several spatial variables.

Internal stability For conventional explicit Runge-Kutta methods, the accumulation of roundoff in the course of a step is unimportant, but that is not the case for methods with a large number of stages. Indeed, in the application of stabilized methods to parabolic PDEs, there can be a serious accumulation of rounding error, so serious that the number of stages must be limited [19, 20, 21]. The form (2.1) minimizes this internal instability, but there is still potential for a growth of roundoff at a modest rate proportional to s^2. For the problems that are the object of RKC and a reasonable working precision, such a growth presents no difficulties. However, for robustness the number of stages is limited in RKC to prevent an unacceptable growth of roundoff in the course of a step. According to [20], a safe assumption about this growth is that it is bounded by a relative perturbation of $10\,s^2\,uround$, where $uround$ is the unit roundoff. The design of RKC emphasizes relative error, so it is required that this perturbation be no greater than $rtol$. Should the code find that it needs to use a larger s for stability with the desired step size, the number of stages is limited and the absolute stability condition (3.1) is satisfied by reducing the step size.

Continuous extension Early codes based on explicit Runge-Kutta methods provide answers at specific points by shortening the step size. This is inefficient, especially when the method has many stages like those of RKC, so modern codes make use of a continuous extension to obtain cheaply answers anywhere in the span of a step. Cubic Hermite interpolation to the value and slope at the two ends of a step proves very satisfactory in the circumstances. It is easy to implement and provides a globally C^1 piecewise-polynomial solution. The interpolant is "free" because the slopes are computed for other purposes. It is shown in [4] that to leading order, the error of this interpolant is independent of the problem. Further, the error increases smoothly from the beginning of the step to a maximum at the end of the step. The error at the end of the step is the local error controlled by the code. Accordingly, to leading order the C^1 piecewise-polynomial solution is uniformly as accurate as the values at the mesh points. RKC is organized so that it can return after each step with the step size taken and all the information required for interpolation stored in a work array. The interpolant is evaluated at a point within the span of the step by calling an auxiliary subroutine RKCINT with the point and the work array as arguments.

4. Numerical examples

In this section we present numerical results for two examples considered by Moore and Dillon [12]. Both are parabolic PDEs in three space dimensions. Moore and Dillon use high order finite elements for the spatial discretization and integrate the ODEs with DASPK. DASPK is a variant of DASSL that uses Krylov methods to make practical the evaluation of the implicit BDFs for "large" systems of ODEs and DAEs [2]. Because our main purpose here is to illustrate the use of RKC, we have discretized the PDEs with central differences on a uniform grid. Although the techniques of [12] are very different, solving the same examples provides some perspective about the use of explicit methods for such problems. We include results computed with VODPK, a BDF code similar to DASPK. It is a modification of VODE [3] and is available from netlib: send vodpk.f from ode. It uses a preconditioned Krylov method GMRES for the solution of the linear systems with matrix $A = I - h\gamma F'$, where F' is the Jacobian. Since iterative methods such as GMRES require only matrix-vector products, A itself need not be stored, reducing greatly the memory needed in the solution of three-dimensional PDEs. VODPK asks the user to specify the preconditioner P. For simplicity, in our experiments we used diagonal preconditioning, i.e. $P = I - h\gamma\,\mathrm{diag}\,(F')$. With this choice, the convergence behavior of GMRES is reasonable and

the storage requirement of 19 vectors is acceptable. In contrast, RKC requires only 5 vectors for the first example and 6 for the second. Default values were used for all the parameters of VODPK All computations were performed in double precision (≈ 16 digits) on a SGI workstation with a 180 MHz MIPS R5000 processor.

Example 1 The first example is the linear heat conduction problem

$$u_t = \Delta u + f(x, y, z, t), \quad 0 < x, y, z < 1, \quad t > 0,$$

where f, $u(x, y, z, 0)$, and Dirichlet boundary conditions are specified so that the solution is $u(x, y, z, t) = \tanh(5(x + 2y + 1.5z - 0.5 - t))$. The problem is solved for $0 \le t \le 0.7$. A uniform grid with spacing $h = 0.025$ is used, corresponding to $39^3 = 59319$ equations.

An analytical bound for the spectral radius of the Jacobian can be found easily by applying Geršgorin's circle theorem to the discrete Laplacian. Because three-point central differences are used, all rows of the matrix corresponding to an interior grid point have the form $h^{-2}(\ldots 1 \ldots 1 \ldots 1 \; -6 \; 1 \ldots 1 \ldots 1 \ldots)$, where "..." represents zero entries. For these rows the circle theorem yields a bound of $12/h^2$. Rows corresponding to a boundary point have more zero entries because of the Dirichlet boundary conditions. Thus $12/h^2$ is an upper bound for the spectral radius and it turns out that the true radius is only marginally smaller. For $h = 0.025$, $\sigma \approx 19200$, so this problem is rather stiff for RKC.

Results reported here were computed with scalar tolerances $rtol = atol = tol$. For a range of tol, Table 1 presents the following quantities: the integration error at the end of the integration measured in the maximum norm, the total number of steps with the number of rejected ones parenthesized, the total number of F-evaluations, the average number of F-evaluations per step (both accepted and rejected), and the CPU time on the workstation in seconds. The error displayed in the table is the difference between the numerical solution and a reference solution of the ODEs computed with a stringent tolerance. It would have been easier to compare the numerical solution to the analytical solution of the PDE, but this would be misleading because it mixes the error of the spatial discretization of the PDEs with the error made in the time integration of the ODEs. For the same tolerances $rtol = atol = tol$, Table 2 presents results for VODPK.

We see that both RKC and VODPK successfully solve the problem for all the tolerances, but RKC is better at delivering an accuracy comparable to the tolerance. The behavior of VODPK is particularly unsatisfactory when tol is reduced from 10^{-5} to 10^{-6}. The efficiency of the solvers is compared in Figure 1 where the CPU time is plotted against the accuracy achieved. RKC is seen to compete well over the whole range of tolerances.

Table 1: Results for RKC for Example 1.

tol	error	# steps	# F-evals	average #	CPU
10^{-1}	$.89\,10^{-2}$	6 (1)	402	67.0	186
10^{-2}	$.17\,10^{-2}$	15 (4)	729	48.6	338
10^{-3}	$.37\,10^{-3}$	27 (2)	786	29.1	366
10^{-4}	$.39\,10^{-4}$	57 (0)	1087	19.1	507
10^{-5}	$.43\,10^{-5}$	129 (1)	1682	13.0	787
10^{-6}	$.65\,10^{-6}$	262 (0)	2445	9.3	1149

Example 2 This example is a combustion problem described by the PDEs

$$c_t = \Delta c - Dce^{-\delta/T}, \quad LT_t = \Delta T + \alpha Dce^{-\delta/T}, \quad 0 < x, y, z < 1, \quad t > 0,$$

Table 2: Results for VODPK for Example 1.

tol	error	# steps	# F-evals	CPU
10^{-1}	.99	7 (0)	46	35
10^{-2}	$.83\,10^{-1}$	16 (0)	160	122
10^{-3}	$.10\,10^{-1}$	34 (0)	237	185
10^{-4}	$.12\,10^{-2}$	70 (0)	474	371
10^{-5}	$.13\,10^{-4}$	112 (3)	984	770
10^{-6}	$.19\,10^{-4}$	168 (1)	1151	913

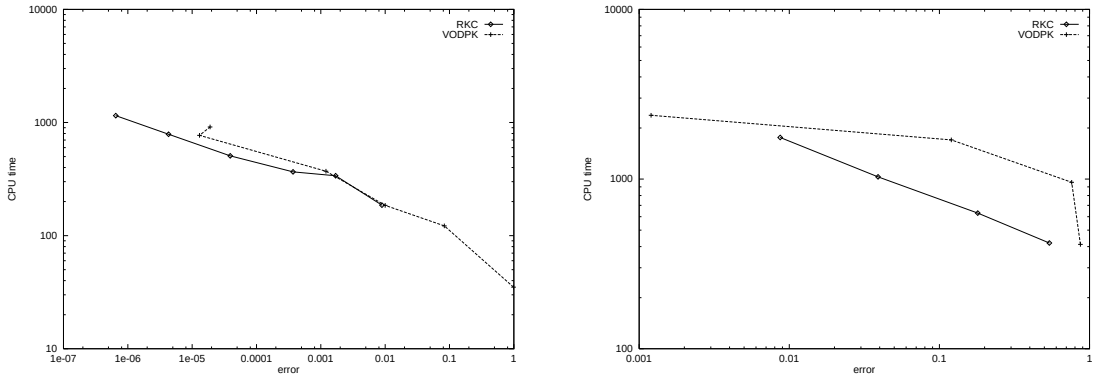

Figure 1: A log-log plot of CPU time versus error for Example 1 (left) and Example 2 (right) for RKC and VODPK.

along with the initial condition $c(x,y,z,0) = T(x,y,z,0) = 1$, homogeneous Neumann boundary conditions for $x = y = z = 0$ and the Dirichlet conditions $c(x,y,z,t) = T(x,y,z,t) = 1$ for $x = y = z = 1$. The parameters of the problem are $L = 0.9, \alpha = 1, \delta = 20$ and $D = Re^{\delta}/\alpha\delta$ with $R = 5$. The dependent variables c and T are the concentration and temperature of a chemical that is undergoing a one-step reaction. The temperature distribution develops a so-called "hot spot" at the origin. Ignition occurs at a finite time and T increases sharply to about $1 + \alpha$. A reaction front is formed that propagates towards the boundary planes $x = y = z = 1$ where it develops a boundary layer and finally ends in a steady state. Following [12] we solve the problem for $0 \leq t \leq 0.3$. By the end of this period the boundary layers have developed and the solution is approaching steady state. A uniform grid with spacing h was used and the Neumann boundary conditions were discretized by means of central differences with fictious points outside the region at a distance of $h/2$. The grid spacing $h = 1/(N + 0.5)$ where N is the number of grid points in each of the three spatial variables. In the computations reported here $N = 40$, leading to a total of $2 * 40^3 = 128000$ equations.

RKC is a natural candidate for the numerical integration of this flame propagation problem. For one thing, the travelling reaction front limits the step size of any integration scheme, be it implicit or explicit. For another, the problem becomes locally unstable in the course of the integration [21], so rather small steps are required to obtain an accurate solution in the transient phase, especially during ignition. Only during the start and near steady state is it possible to increase the step size to the point that an implicit method is competitive.

Tables 3 and 4 present results in the same way as for the first example. An extra column in Table 3 shows the number of F-evals needed by RKC for the estimation of the spectral radius. We see that the overhead for this automatic estimation is negligible. Both solvers integrate this difficult problem successfully with only a few step rejections. Neither code obtains accuracies comparable to the tolerance, though again RKC is notably better. With VODPK there is a striking change in accuracy

when reducing *tol* from 10^{-6} to 10^{-7}. The low accuracy achieved by both codes is to be expected from the local instability of the problem. Figure 1 shows that RKC competes well with VODPK for this problem, too. RKC adapts the formula, i.e., the number of stages s, to the problem and it may use s that are quite large compared to what is seen in general-purpose codes based on explicit Runge-Kutta formulas. The variation of s when solving this problem is displayed in Figure 2.

Table 3: Results for RKC for Example 2.

tol	error	# steps	# F-evals	average #	# F-evals σ	CPU
10^{-4}	.54	51 (1)	525	10.3	21	420
10^{-5}	.18	124 (0)	781	6.3	27	630
10^{-6}	$.39\,10^{-1}$	270 (0)	1270	4.7	39	1030
10^{-7}	$.87\,10^{-2}$	581 (0)	2147	3.7	65	1758

Table 4: Results for VODPK for Example 2.

tol	error	# steps	# F-evals	CPU
10^{-4}	.87	33 (2)	285	412
10^{-5}	.76	91 (8)	659	957
10^{-6}	.12	201 (9)	1141	1702
10^{-7}	$.12\,10^{-2}$	286 (10)	1548	2376

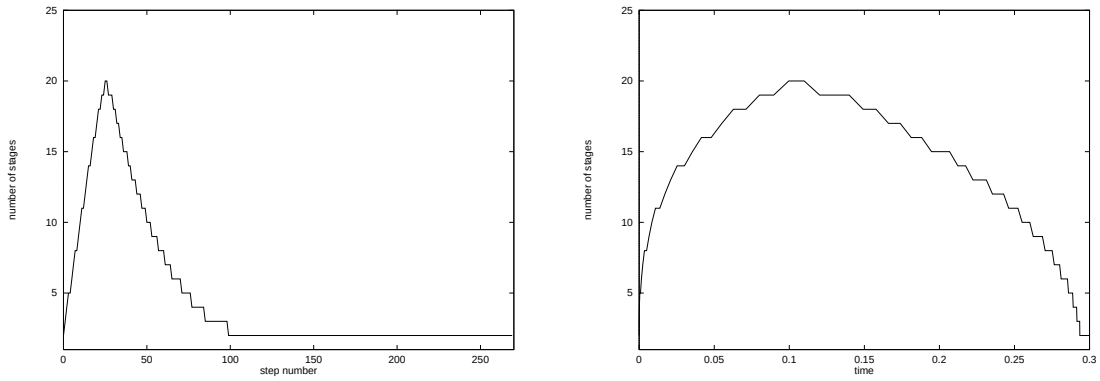

Figure 2: The number of stages s used by RKC when solving Example 2 with $tol = 10^{-6}$ plotted against step number (left) and against time (right).

5. Remarks

Other interesting stabilized explicit methods have been developed by Lebedev and co-workers, see, e.g. [9, 11] and [8, 21]. There are formulas of order up to four [11]. Although they are also based on Chebyshev polynomials and so possess optimal stability for real negative eigenvalues, the three-term recursion is not exploited. A code DUMKA based on these formulas is still in an experimental stage, but numerical results are promising, see Figure 10.14 in [8]).

Source code for RKC and some examples can be obtained by anonymous ftp from the address ftp://ftp.cwi.nl/pub/bsom/rkc. RKC can also be downloaded from netlib@ornl.gov (send rkc.f from ode). It replaces the program of [18].

References

1. R.W. Brankin, I. Gladwell, and L.F. Shampine, RKSUITE: a suite of Runge-Kutta codes for the initial value problem for ODEs, Softreport 91-1, Math. Dept., Southern Methodist Univ., Dallas (1991).
2. P.N. Brown, A.C. Hindmarsh, and L.R. Petzold, Using Krylov methods in the solution of large-scale differential-algebraic systems. *SIAM J. Sci. Comput.* 15 (1994) 1467 - 1488.
3. P.N. Brown, G.D. Byrne and A.C. Hindmarsh, VODE, a variable-coefficient ODE solver. *SIAM J. Sci. Stat. Comput.* 10 (1989) 1038 - 1051.
4. I. Gladwell, L.F. Shampine, L.S. Baca, and R.W. Brankin, Practical aspects of interpolation in Runge-Kutta codes, *SIAM J. Sci. Stat. Comput.* 8 (1987) 322-341.
5. I. Gladwell, L.F. Shampine, and R.W. Brankin, Automatic selection of the initial step size for an ODE solver, *J. Comput. Appl. Math.* 18 (1987) 175-192.
6. K. Gustafsson, M. Lundh, and G. Söderlind, A *PI* stepsize control for the numerical solution of ordinary differential equations. *BIT* 28, 270 - 287, 1988.
7. E. Hairer, S.P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I, Nonstiff Problems* (Springer-Verlag, 1987).
8. E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II, Stiff and Differential-Algebraic Problems*, 2nd ed., (Springer-Verlag, 1996).
9. V.I. Lebedev, *How to solve stiff systems of differential equations by explicit methods*, In: *Numerical methods and applications*, ed. by G.I. Marchuk, pp. 45–80 (CRC Press, 1994).
10. B. Lindberg, IMPEX: a program package for solution of systems of stiff differential equations, Rept. NA 72.50, Royal Inst. Technology, Stockholm, 1972.
11. A.A. Medovikov, High order explicit methods for stiff ordinary differential equations (Preprint Russian Academy of Sciences, 1996).
12. P.K. Moore and R.H. Dillon, A comparison of preconditioners in the solution of parabolic systems in three space dimensions using DASPK and a high order finite element method, *Appl. Numer. Math.* 20 (1996) 117 - 128.
13. P.J. van der Houwen and B.P. Sommeijer, On the internal stability of explicit m-stage Runge-Kutta methods for large values of m, *ZAMM* 60 (1980) 479 - 485.
14. L.F. Shampine, Lipschitz constants and robust ODE codes, pp. 427-449 in J.T. Oden, ed., *Computational Methods in Nonlinear Mechanics* (North Holland, New York, 1980).
15. L.F. Shampine, Diagnosing stiffness for Runge-Kutta methods, *SIAM J. Sci. Stat. Comput.* 12 (1991) 260-272.
16. L.F. Shampine, *Numerical Solution of Ordinary Differential Equations* (Chapman and Hall, New York, 1994).
17. B.P. Sommeijer and J.G. Verwer, A performance evaluation of a class of Runge-Kutta-Chebyshev methods for solving semi-discrete parabolic differential equations, Report NW91/80, Mathematisch Centrum, Amsterdam (1980).
18. B.P. Sommeijer, RKC, a nearly-stiff ODE solver. Available from netlib@ornl.gov, send rkc.f from ode (1991).
19. J.G. Verwer, An implementation of a class of stabilized explicit methods for the time integration of parabolic equations, *ACM Trans. Math. Software* 6 (1980) 188 - 205.
20. J.G. Verwer, W.H. Hundsdorfer, and B.P. Sommeijer, Convergence properties of the Runge-Kutta-Chebyshev method, *Numer. Math.* 57 (1990) 157 - 178.

21. J.G. Verwer, Explicit Runge-Kutta methods for parabolic partial differential equations, *Appl. Numer. Math.* 22 (1996) 359-379.

22. H.A. Watts, Step size control in ordinary differential equation solvers, *Trans. Soc. for Computer Simulation* 1 (1984) 15-25.

1. Source of RKC

```
      subroutine rkc(neqn,f,y,t,tend,rtol,atol,info,work,idid)
c-----------------------------------------------------------------------
c
c  ABSTRACT:  RKC integrates initial value problems for systems of first
c  order ordinary differential equations.  It is based on a family of
c  explicit Runge-Kutta-Chebyshev formulas of order two.  The stability
c  of members of the family increases quadratically in the number of
c  stages m. An estimate of the spectral radius is used at each step to
c  select the smallest m resulting in a stable integration. RKC is
c  appropriate for the solution to modest accuracy of mildly stiff problems
c  with eigenvalues of Jacobians that are close to the negative real axis.
c  For such problems it has the advantages of explicit one-step methods and
c  very low storage. If it should turn out that RKC is using m far beyond
c  100, the problem is not mildly stiff and alternative methods should be
c  considered.  Answers can be obtained cheaply anywhere in the interval
c  of integration by means of a continuous extension evaluated in the
c  subroutine RKCINT.
c
c  The initial value problems arising from semi-discretization of
c  diffusion-dominated parabolic partial differential equations and of
c  reaction-diffusion equations, especially in two and three spatial
c  variables, exemplify the problems for which RKC was designed.  Two
c  example programs, exa and exb, are provided that show how to use RKC.
c
c-----------------------------------------------------------------------
c  USAGE:  RKC integrates a system of NEQN first order ordinary differential
c  equations specified by a subroutine F from T to TEND.  The initial values
c  at T are input in Y(*).  On all returns from RKC, Y(*) is an approximate
c  solution at T.  In the computation of Y(*), the local error has been
c  controlled at each step to satisfy a relative error tolerance RTOL and
c  absolute error tolerances ATOL(*).  The array INFO(*) specifies the way
c  the problem is to be solved.  WORK(*) is a work array. IDID reports
c  success or the reason the computation has been terminated.
c
c  FIRST CALL TO RKC
c
c  You must provide storage in your calling program for the arrays in the
c  call list -- Y(NEQN), INFO(4), WORK(8+5*NEQN).  If INFO(2) = 0, you can
c  reduce the storage for the work array to WORK(8+4*NEQN).  ATOL may be
c  a scalar or an array.  If it is an array, you must provide storage for
c  ATOL(NEQN).  You must declare F in an external statement, supply the
c  subroutine F and the function SPCRAD, and initialize the following
c  quantities:
c
c    NEQN:  The number of differential equations. Integer.
c
c    T:     The initial point of the integration. Double precision.
c           Must be a variable.
c
```

```
c    TEND:  The end of the interval of integration.  Double precision.
c           TEND may be less than T.
c
c    Y(*):  The initial value of the solution.  Double precision array
c           of length NEQN.
c
c    F:     The name of a subroutine for evaluating the differential
c           equation.  It must have the form
c
c             subroutine f(neqn,t,y,dy)
c             integer          neqn
c             double precision t,y(neqn),dy(neqn)
c             dy(1)    = ...
c             ...
c             dy(neqn) = ...
c             return
c             end
c
c  RTOL,
C  ATOL(*):  At each step of the integration the local error is controlled
c           so that its RMS norm is no larger than tolerances RTOL, ATOL(*).
c           RTOL is a double precision scalar. ATOL(*) is either a double
c           precision scalar or a double precision array of length NEQN.
c           RKC is designed for the solution of problems to modest accuracy.
c           Because it is based on a method of order 2, it is relatively
c           expensive to achieve high accuracy.
c
c           RTOL is a relative error tolerance.  You must ask for some
c           relative accuracy, but you cannot ask for too much for the
c           precision available.  Accordingly, it is required that
c           0.1 >= RTOL >= 10*uround. (See below for the machine and
c           precision dependent quantity uround.)
c
c           ATOL is an absolute error tolerance that can be either a
c           scalar or an array.  When it is an array, the tolerances are
c           applied to corresponding components of the solution and when
c           it is a scalar, it is applied to all components.  A scalar
c           tolerance is reasonable only when all solution components are
c           scaled to be of comparable size.  A scalar tolerance saves a
c           useful amount of storage and is convenient.  Use INFO(*) to
c           tell RKC whether ATOL is a scalar or an array.
c
c           The absolute error tolerances ATOL(*) must satisfy ATOL(i) >= 0
c           for i = 1,...,NEQN.  ATOL(j)= 0 specifies a pure relative error
c           test on component j of the solution, so it is an error if this
c           component vanishes in the course of the integration.
c
c           If all is going well, reducing the tolerances by a factor of
c           0.1 will reduce the error in the computed solution by a factor
c           of roughly 0.2.
c
```

```
c  INFO(*)   Integer array of length 4 that specifies how the problem
c            is to be solved.
c
c  INFO(1):  RKC integrates the initial value problem from T to TEND.
c            This is done by computing approximate solutions at points
c            chosen automatically throughout [T, TEND].  Ordinarily RKC
c            returns at each step with an approximate solution. These
c            approximations show how y behaves throughout the interval.
c            The subroutine RKCINT can be used to obtain answers anywhere
c            in the span of a step very inexpensively. This makes it
c            possible to obtain answers at specific points in [T, TEND]
c            and to obtain many answers very cheaply when attempting to
c            locating where some function of the solution has a zero
c            (event location).  Sometimes you will be interested only in
c            a solution at TEND, so you can suppress the returns at each
c            step along the way if you wish.
c
c  INFO(1)  = 0 Return after each step on the way to TEND with a
c               solution Y(*) at the output value of T.
c
c           = 1 Compute a solution Y(*) at TEND only.
c
c  INFO(2):  RKC needs an estimate of the spectral radius of the Jacobian.
c            You must provide a function that must be called SPCRAD and
c            have the form
c
c              double precision function spcrad(neqn,t,y)
c              integer          neqn
c              double precision t,y(neqn)
c
c              spcrad = < expression depending on info(2) >
c
c              return
c              end
c
c            You can provide a dummy function and let RKC compute the
c            estimate. Sometimes it is convenient for you to compute in
c            SPCRAD a reasonably close upper bound on the spectral radius,
c            using, e.g., Gershgorin's theorem.  This may be faster and/or
c            more reliable than having RKC compute one.
c
c  INFO(2)  = 0  RKC is to compute the estimate internally.
c                Assign any value to SPCRAD.
c
c           = 1  SPCRAD returns an upper bound on the spectral
c                radius of the Jacobian of f at (t,y).
c
c  INFO(3):  If you know that the Jacobian is constant, you should say so.
c
c  INFO(3)  = 0  The Jacobian may not be constant.
c
```

```
c           = 1  The Jacobian is constant.
c
c  INFO(4):  You must tell RKC whether ATOL is a scalar or an array.
c
c  INFO(4)  = 0  ATOL is a double precision scalar.
c
c           = 1  ATOL is a double precision array of length NEQN.
c
c  WORK(*):  Work array.  Double precision array of length at least
c            8 + 4*NEQN if INFO(2) = 0 and otherwise, 8 + 5*NEQN.
c
c  IDID:     Set IDID = 0 to initialize the integration.
c
c
c
c  RETURNS FROM RKC
c
c  T:        The integration has advanced to T.
c
c  Y(*):     The solution at T.
c
c  IDID:     The value of IDID reports what happened.
c
c                          SUCCESS
c
c    IDID     = 1 T = TEND, so the integration is complete.
c
c             = 2 Took a step to the output value of T.  To continue on
c                 towards TEND, just call RKC again.   WARNING:  Do not
c                 alter any argument between calls.
c
c                 The last step, HLAST, is returned as WORK(1). RKCINT
c                 can be used to approximate the solution anywhere in
c                 [T-HLAST, T] very inexpensively using data in WORK(*).
c
c                 The work can be monitored by inspecting data in RKCDID.
c
c                          FAILURE
c
c             = 3 Improper error control: For some j, ATOL(j) = 0
c                 and Y(j) = 0.
c
c             = 4 Unable to achieve the desired accuracy with the
c                 precision available.  A severe lack of smoothness in
c                 the solution y(t) or the function f(t,y) is likely.
c
c             = 5 Invalid input parameters:  NEQN <= 0, RTOL > 0.1,
c                 RTOL < 10*UROUND, or ATOL(i) < 0 for some i.
c
c             = 6 The method used by RKC to estimate the spectral
c                 radius of the Jacobian failed to converge.
```

```
c
c  RKCDID is a labelled common block that communicates statistics
c         about the integration process:
c         common /rkcdid/   nfe,nsteps,naccpt,nrejct,nfesig,maxm
c
c         The integer counters are:
c
c        NFE      number of evaluations of F used
c                   to integrate the initial value problem
c        NSTEPS   number of integration steps
c        NACCPT   number of accepted steps
c        NREJCT   number of rejected steps
c        NFESIG   number of evaluations of F used
c                   to estimate the spectral radius
c        MAXM     maximum number of stages used
c
c        This data can be used to monitor the work and terminate a run
c        that proves to be unacceptably expensive.  Also, if MAXM should
c        be far beyond 100, the problem is too expensive for RKC and
c        alternative methods should be considered.
c
c-----------------------------------------------------------------------
c
c   CAUTION: MACHINE/PRECISION ISSUES
c
c     UROUND (the machine precision) is the smallest number such that
c     1 + UROUND > 1, where 1 is a floating point number in the working
c     precision. UROUND is set in a parameter statement in RKC. Its
c     value depends on both the precision and the machine used, so it
c     must be set appropriately.  UROUND is the only constant in RKC
c     that depends on the precision.
c
c     This version of RKC is written in double precision. It can be changed
c     to single precision by replacing DOUBLE PRECISION in the declarations
c     by REAL and changing the type of the floating point constants set in
c     PARAMETER statements from double precision to real.
c
c-----------------------------------------------------------------------
c
c  Authors: B.P. Sommeijer and J.G. Verwer
c           Centre for Mathematics and Computer Science (CWI)
c           Kruislaan 413
c           1098 SJ  Amsterdam
c           The Netherlands
c           e-mail: bsom@cwi.nl
c
c           L.F. Shampine
c           Mathematics Department
c           Southern Methodist University
c           Dallas, Texas 75275-0156
c           USA
```

```
c           e-mail: lshampin@mail.smu.edu
c
c  Details of the methods used and the performance of RKC can be
c  can be found in
c
c         B.P. Sommeijer, L.F. Shampine and J.G. Verwer
c         RKC: an Explicit Solver for Parabolic PDEs.
c         Report MAS-R9715, CWI, Amsterdam, 1997
c
c-----------------------------------------------------------------------
      integer          neqn,info(*),idid
      double precision y(neqn),t,tend,rtol,atol(*),work(*)
c
c*****************************************************************
c  uround is set here for IEEE double precision arithmetic.
      double precision uround
      parameter       (uround=2.22d-16)
c*****************************************************************
c
      double precision zero,rmax,rmin
      parameter       (zero=0d0,rmax=0.1d0,rmin=10d0*uround)
      integer          i,ptr1,ptr2,ptr3,ptr4
      logical          array,valid
      save
      integer          nfe,nsteps,naccpt,nrejct,nfesig,maxm
      common /rkcdid/  nfe,nsteps,naccpt,nrejct,nfesig,maxm
      external         f
c
      if(idid .eq. 0) then
c----------------------
c  Test the input data.
c----------------------
         array = info(4) .eq. 1
         valid = neqn .gt. 0
         if((rtol .gt. rmax) .or. (rtol .lt. rmin)) valid = .false.
         if(atol(1) .lt. zero) valid = .false.
         if(array) then
           do 10 i = 2, neqn
             if(atol(i) .lt. zero) valid = .false.
10         continue
         endif
         if(.not. valid) then
           idid = 5
           return
         endif
c-----------------------------------
c  Initialize counters and pointers.
c-----------------------------------
         nfe = 0
         nsteps = 0
         naccpt = 0
```

```
         nrejct = 0
         nfesig = 0
         maxm = 0
c-------------------------------------------------------------
c  work(*) contains information needed for interpolation,
c  continuation after a return, and working storage. Items
c  relevant here are:
c
c  The last step taken, hlast, is work(1).
c  The current t is work(2).
c  The number of equations, neqn, is work(3).
c  The unit roundoff, uround, is work(4).
c  The square root of uround, sqrtu, is work(5).
c  The maximum step size, hmax, is work(6).
c  The base address for the solution is ptr1 = nint(work(7)).
c  The solution at t starts at ptr1.
c  The derivative of the solution at t starts at ptr2.
c  The solution at t-hlast starts at ptr3.
c  The derivative of the solution at t-hlast starts at ptr4.
c  The estimated dominant eigenvector starts at ptr4 + neqn.
c-------------------------------------------------------------
        work(2) = t
        work(3) = neqn
        work(4) = uround
        work(5) = sqrt(uround)
        ptr1 = 8
        work(7) = ptr1
        ptr2 = ptr1 + neqn
        ptr3 = ptr2 + neqn
        ptr4 = ptr3 + neqn
      elseif(idid .ne. 2) then
        write(*,*) ' RKC was called with an illegal value of IDID.'
        stop
      endif
c
      call rkclow(neqn,t,tend,y,f,info,rtol,atol,work,
     &            work(ptr1),work(ptr2),work(ptr3),work(ptr4),idid)
      return
      end

      subroutine rkclow(neqn,t,tend,y,f,info,rtol,atol,work,
     &                  yn,fn,vtemp1,vtemp2,idid)
c---------------------------------------------------------------------
c  RKC is an interface to RKCLOW where the actual solution takes place.
c---------------------------------------------------------------------
      integer          neqn,info(*),idid
      double precision t,tend,y(*),rtol,atol(*),work(*),
     &                 yn(*),fn(*),vtemp1(*),vtemp2(*)
      external         f
c
      double precision one,onep1,onep54,p1,p4,p8,
```

```
     &                 ten,zero,one3rd,two3rd
      parameter       (one=1d0,onep1=1.1d0,onep54=1.54d0,
     &                 p1=0.1d0,p4=0.4d0,p8=0.8d0,ten=10d0,
     &                 zero=0d0,one3rd=1d0/3d0,two3rd=2d0/3d0)
      integer          i,m,mmax,nstsig
      double precision absh,est,err,errold,fac,h,hmax,hmin,hold,
     &                 spcrad,sprad,tdir,temp1,temp2,
     &                 uround,wt,ylast,yplast,at
      logical          array,last,newspc,jacatt
      save
      integer          nfe,nsteps,naccpt,nrejct,nfesig,maxm
      common /rkcdid/  nfe,nsteps,naccpt,nrejct,nfesig,maxm
c
c---------------------------------
c    Initialize on the first call.
c---------------------------------
      if(idid .eq. 0) then
        array = info(4) .eq. 1
        uround = work(4)
        mmax = nint(sqrt(rtol/(10d0*uround)))
        mmax = max(mmax,2)
        newspc = .true.
        jacatt = .false.
        nstsig = 0
        do 10 i = 1, neqn
          yn(i) = y(i)
10      continue
        call f(neqn,t,yn,fn)
        nfe = nfe + 1
        tdir = sign(one,tend - t)
        hmax = abs(tend - t)
        work(6) = hmax
        hmin = ten*uround*max(abs(t),hmax)
      endif
c-----------------------------------
c  Start of loop for taking one step.
c-----------------------------------
20    continue
c-------------------------------------------
c  Estimate the spectral radius of the Jacobian
c  when newspc = .true..  A convergence failure
c  in rkcrho is reported by idid = 6.
c-------------------------------------------
      if(newspc) then
        if(info(2) .eq. 1) then
          sprad = spcrad(neqn,t,yn)
        else
          call rkcrho(neqn,t,f,yn,fn,vtemp1,vtemp2,work,sprad,idid)
          if(idid .eq. 6) return
        endif
        jacatt = .true.
```

```
      endif
c-----------------------------
c  Compute an initial step size.
c-----------------------------
      if(nsteps .eq. 0) then
        absh = hmax
        if(sprad*absh .gt. one) absh = one/sprad
        absh = max(absh,hmin)
        do 30 i = 1,neqn
          vtemp1(i) = yn(i) + absh*fn(i)
30      continue
        call f(neqn,t+absh,vtemp1,vtemp2)
        nfe = nfe + 1
        est = zero
        at = atol(1)
        do 40 i = 1,neqn
          if(array) at = atol(i)
          wt = at + rtol*abs(yn(i))
          if(wt .eq. zero) then
            idid = 3
            return
          endif
          est = est + ((vtemp2(i) - fn(i))/wt)**2
40      continue
        est = absh*sqrt(est/neqn)
        if(p1*absh .lt. hmax*sqrt(est)) then
          absh = max(p1*absh/sqrt(est), hmin)
        else
          absh = hmax
        endif
      endif
c-------------------------------------------------------------
c  Adjust the step size and determine the number of stages m.
c-------------------------------------------------------------
      last = .false.
      if(onep1*absh .ge. abs(tend - t)) then
        absh = abs(tend - t)
        last = .true.
      endif
      m = 1 + int(sqrt(onep54*absh*sprad + one))
c-------------------------------------------------------------
c  Limit m to mmax to control the growth of roundoff error.
c-------------------------------------------------------------
      if(m .gt. mmax) then
        m = mmax
        absh = (m**2 - 1)/(onep54*sprad)
        last = .false.
      endif
      maxm = max(m,maxm)
c-----------------------------------------
c  A tentative solution at t+h is returned in
```

```
c  y and its slope is evaluated in vtemp1(*).
c-----------------------------------------
      h = tdir*absh
      hmin = ten*uround*max(abs(t),abs(t + h))
      call step(neqn,f,t,yn,fn,h,m,y,vtemp1,vtemp2)
      call f(neqn,t+h,y,vtemp1)
      nfe = nfe + m
      nsteps = nsteps + 1
c------------------------------------------------------------
c  Estimate the local error and compute its weighted RMS norm.
c------------------------------------------------------------
      err = zero
      at = atol(1)
      do 50 i = 1, neqn
        if(array) at = atol(i)
        wt = at + rtol*max(abs(y(i)),abs(yn(i)))
        if(wt .eq. zero) then
          idid = 3
          return
        endif
        est = p8*(yn(i) - y(i)) + p4*h*(fn(i) + vtemp1(i))
        err = err + (est/wt)**2
50    continue
      err = sqrt(err/neqn)
c
      if(err .gt. one) then
c-------------------
c  Step is rejected.
c-------------------
        nrejct = nrejct + 1
        absh = p8*absh/(err**one3rd)
        if(absh .lt. hmin) then
          idid = 4
          return
        else
          newspc = .not. jacatt
          goto 20
        endif
      endif
c-------------------
c  Step is accepted.
c-------------------
      naccpt = naccpt + 1
      t = t + h
      jacatt = info(3) .eq. 1
      nstsig = mod(nstsig+1,25)
      newspc = .false.
      if((info(2) .eq. 1) .or. (nstsig .eq. 0)) newspc = .not. jacatt
c-------------------------------------------------------
c  Update the data for interpolation stored in work(*).
c-------------------------------------------------------
```

```
      work(1) = h
      work(2) = t
      do 60 i = 1, neqn
         ylast = yn(i)
         yplast = fn(i)
         yn(i) = y(i)
         fn(i) = vtemp1(i)
         vtemp1(i) = ylast
         vtemp2(i) = yplast
60    continue
      fac = ten
      if(naccpt .eq. 1) then
        temp2 = err**one3rd
        if(p8 .lt. fac*temp2) fac = p8/temp2
      else
        temp1 = p8*absh*errold**one3rd
        temp2 = abs(hold)*err**two3rd
        if(temp1 .lt. fac*temp2) fac = temp1/temp2
      endif
      absh = max(p1,fac)*absh
      absh = max(hmin,min(hmax,absh))
      errold = err
      hold = h
      h = tdir*absh
      if(last) then
        idid = 1
        return
      elseif(info(1) .eq. 0) then
        idid = 2
        return
      else
        goto 20
      endif
      end

      subroutine step(neqn,f,t,yn,fn,h,m,y,yjm1,yjm2)
c-----------------------------------------------------
c  Take a step of size H from T to T+H to get Y(*).
c-----------------------------------------------------
      integer          neqn,m
      double precision t,yn(neqn),fn(neqn),h,
     &                 y(neqn),yjm1(neqn),yjm2(neqn)
      external         f
c
      double precision one,two,four,c13,zero
      parameter       (one=1d0,two=2d0,four=4d0,c13=13d0,zero=0d0)
      integer          i,j
      double precision ajm1,arg,bj,bjm1,bjm2,dzj,dzjm1,dzjm2,
     &                 d2zj,d2zjm1,d2zjm2,mu,mus,nu,
     &                 temp1,temp2,thj,thjm1,thjm2,w0,w1,
     &                 zj,zjm1,zjm2
```

```
c
      w0 = one + two/(c13*m**2)
      temp1 = w0**2 - one
      temp2 = sqrt(temp1)
      arg = m*log(w0 + temp2)
      w1 = sinh(arg)*temp1 / (cosh(arg)*m*temp2 - w0*sinh(arg))
      bjm1 = one/(two*w0)**2
      bjm2 = bjm1
c-------------------------
c  Evaluate the first stage.
c-------------------------
      do 10 i = 1, neqn
        yjm2(i) = yn(i)
10    continue
      mus = w1*bjm1
      do 20 i = 1, neqn
        yjm1(i) = yn(i) + h*mus*fn(i)
20    continue
      thjm2  = zero
      thjm1  = mus
      zjm1   = w0
      zjm2   = one
      dzjm1  = one
      dzjm2  = zero
      d2zjm1 = zero
      d2zjm2 = zero
c-----------------------------
c  Evaluate stages j = 2,...,m.
c-----------------------------
      do 50 j = 2, m
        zj   =   two*w0*zjm1 - zjm2
        dzj  =   two*w0*dzjm1 - dzjm2 + two*zjm1
        d2zj =   two*w0*d2zjm1 - d2zjm2 + four*dzjm1
        bj   =   d2zj/dzj**2
        ajm1 =   one - zjm1*bjm1
        mu   =   two*w0*bj/bjm1
        nu   = - bj/bjm2
        mus  =   mu*w1/w0
c-------------------------------------------
c  Use the y array for temporary storage here.
c-------------------------------------------
        call f(neqn,t + h*thjm1,yjm1,y)
        do 30 i = 1, neqn
          y(i) = mu*yjm1(i) + nu*yjm2(i) + (one - mu - nu)*yn(i) +
     &           h*mus*(y(i) - ajm1*fn(i))
30      continue
        thj = mu*thjm1 + nu*thjm2 + mus*(one - ajm1)
c-----------------------------------
c  Shift the data for the next stage.
c-----------------------------------
        if(j .lt. m) then
```

```
          do 40 i = 1, neqn
            yjm2(i) = yjm1(i)
            yjm1(i) = y(i)
40        continue
          thjm2  = thjm1
          thjm1  = thj
          bjm2   = bjm1
          bjm1   = bj
          zjm2   = zjm1
          zjm1   = zj
          dzjm2  = dzjm1
          dzjm1  = dzj
          d2zjm2 = d2zjm1
          d2zjm1 = d2zj
        endif
50    continue
      return
      end

      subroutine rkcint(work,arg,yarg)
c-----------------------------------------------------------------------
c  RKCINT is used to compute approximate solutions at specific t and to
c  compute cheaply the large number of approximations that may be needed
c  for plotting or locating when events occur.
c
c  After a step to T, RKC provides HLAST, the step just taken, in WORK(1).
c  In other entries of WORK(*) it provides the data needed to interpolate
c  anywhere in [T-HLAST, T]. YARG(*), the approximate solution at t = ARG
c  computed by interpolation in RKCINT has the same order of accuracy as
c  the Y(*) computed directly by RKC.
c
c  INPUT:
c
c    WORK(*)   Double precision array returned by RKC.
c
c    ARG       The point at which a solution is desired. Double precision.
c
c  OUTPUT:
c
c    YARG(*)   The approximate solution at t = ARG.  Double precision
c              array of length neqn.
c-----------------------------------------------------------------------
      double precision work(*),arg,yarg(*)
c
      double precision one,two,three
      parameter       (one=1d0,two=2d0,three=3d0)
      integer          i,neqn,ptr1,ptr2,ptr3,ptr4
      double precision a1,a2,b1,b2,s,hlast,t,tlast
c
c-----------------------------------------------------------------------
c  The data needed for interpolation are stored in work(*) as follows:
```

```
c
c  The last step taken, hlast, is work(1).
c  The current t is work(2).
c  The number of equations, neqn, is work(3).
c  The base address for the solution is ptr1 = nint(work(7))
c  The solution at t starts at ptr1.
c  The derivative of the solution at t starts at ptr2.
c  The solution at t-hlast starts at ptr3.
c  The derivative of the solution at t-hlast starts at ptr4.
c-----------------------------------------------------------------------
      hlast = work(1)
      t = work(2)
      tlast = t - hlast
      neqn = nint(work(3))
      ptr1 = nint(work(7))
      ptr2 = ptr1 + neqn
      ptr3 = ptr2 + neqn
      ptr4 = ptr3 + neqn
c
      s  = (arg - tlast)/hlast
      a1 = (one + two*s)*(s - one)**2
      a2 = (three - two*s)*s**2
      b1 = hlast*s*(s - one)**2
      b2 = hlast*(s - one)*s**2
c
      do 10 i = 1, neqn
        yarg(i) = a1*work(ptr3+i-1) + a2*work(ptr1+i-1) +
     &            b1*work(ptr4+i-1) + b2*work(ptr2+i-1)
10    continue
      return
      end

      subroutine rkcrho(neqn,t,f,yn,fn,v,fv,work,sprad,idid)
c-----------------------------------------------------------------
c  RKCRHO attempts to compute a close upper bound, SPRAD, on
c  the spectral radius of the Jacobian matrix using a nonlinear
c  power method.  A convergence failure is reported by IDID = 6.
c-----------------------------------------------------------------
      integer          neqn,idid
      double precision t,yn(neqn),fn(neqn),v(neqn),fv(neqn),work(*),
     &                 sprad
      external         f
c
      integer          itmax
      parameter       (itmax=50)
      double precision zero,one,onep2,p01
      parameter       (zero=0d0,one=1d0,onep2=1.2d0,p01=0.01d0)
      integer          i,iter,index,ptr5
      double precision uround,sqrtu,ynrm,sigma,sigmal,
     &                 dynrm,dfnrm,vnrm,small
      integer          nfe,nsteps,naccpt,nrejct,nfesig,maxm
```

```
      common /rkcdid/  nfe,nsteps,naccpt,nrejct,nfesig,maxm
c
      uround = work(4)
      sqrtu = work(5)
c-------------------------------------------------------------
c  hmax = work(6).  sprad smaller than small = 1/hmax are not
c  interesting because they do not constrain the step size.
c-------------------------------------------------------------
      small = one/work(6)
c-------------------------------------------------------
c  The initial slope is used as guess when nsteps = 0 and
c  thereafter the last computed eigenvector.  Some care
c  is needed to deal with special cases. Approximations to
c  the eigenvector are normalized so that their Euclidean
c  norm has the constant value dynrm.
c-------------------------------------------------------
      ptr5 = nint(work(7)) + 4*neqn
      if(nsteps .eq. 0) then
        do 10 i = 1,neqn
          v(i) = fn(i)
10      continue
      else
        do 20 i = 1,neqn
          v(i) = work(ptr5+i-1)
20      continue
      endif
      ynrm = zero
      vnrm = zero
      do 30 i = 1,neqn
        ynrm = ynrm + yn(i)**2
        vnrm = vnrm + v(i)**2
30    continue
      ynrm = sqrt(ynrm)
      vnrm = sqrt(vnrm)
      if(ynrm .ne. zero .and. vnrm .ne. zero) then
        dynrm = ynrm*sqrtu
        do 40 i = 1,neqn
          v(i) = yn(i) + v(i)*(dynrm/vnrm)
40      continue
      elseif(ynrm .ne. zero) then
        dynrm = ynrm*sqrtu
        do 50 i = 1, neqn
          v(i) = yn(i) + yn(i)*sqrtu
50      continue
      elseif(vnrm .ne. zero) then
        dynrm = uround
        do 60 i = 1,neqn
          v(i) = v(i)*(dynrm/vnrm)
60      continue
      else
        dynrm = uround
```

```
        do 70 i = 1,neqn
          v(i) = dynrm
70      continue
      endif
c-----------------------------------------
c  Now iterate with a nonlinear power method.
c-----------------------------------------
      sigma = zero
      do 110 iter = 1, itmax
        call f(neqn,t,v,fv)
        nfesig = nfesig + 1
        dfnrm = zero
        do 80 i = 1, neqn
          dfnrm = dfnrm + (fv(i) - fn(i))**2
80      continue
        dfnrm = sqrt(dfnrm)
        sigmal = sigma
        sigma = dfnrm/dynrm
c-------------------------------------------------------
c  sprad is a little bigger than the estimate sigma of the
c  spectral radius, so is more likely to be an upper bound.
c-------------------------------------------------------
        sprad = onep2*sigma
        if(iter .ge. 2 .and.
     &     abs(sigma - sigmal) .le. max(sigma,small)*p01) then
          do 90 i = 1,neqn
            work(ptr5+i-1) = v(i) - yn(i)
90        continue
          return
        endif
c-------------------------------------
c  The next v(*) is the change in f
c  scaled so that norm(v - yn) = dynrm.
c-------------------------------------
        if(dfnrm .ne. zero) then
          do 100 i = 1,neqn
            v(i) = yn(i) + (fv(i) - fn(i))*(dynrm/dfnrm)
100       continue
        else
c-----------------------------------------------------
c  The new v(*) degenerated to yn(*)--"randomly" perturb
c  current approximation to the eigenvector by changing
c  the sign of one component.
c-----------------------------------------------------
          index = 1 + mod(iter,neqn)
          v(index) = yn(index) - (v(index) - yn(index))
        endif
110   continue
c-----------------------------------------
c  Set flag to report a convergence failure.
c-----------------------------------------
```

```
      idid = 6
      return
      end
```

2. Example A

```
c                            Example A
c
c  This example shows how to use RKC.  It solves a system of ODEs that
c  arise from semi-discretization of the reaction-diffusion equation
c
c            U  = U   + (1 - U)*U**2    for   t >= 0,   0 <= x <= 10
c             t    xx
c
c  Dirichlet boundary conditions specify U(0,t) and U(10,t) for all t >= 0
c  and the initial values U(x,0) are specified. These values are taken from
c  an analytical solution that is evaluated in sol(x,t) so that the numerical
c  solution can be compared to a known solution.
c
c  A semi-discretization of the PDE is obtained by choosing a set of points
c  {x_i} in [0, 10] and approximating U(x_i,t) by a function y_i(t).  Here
c  neqn+2 equally spaced points x_i are used for neqn = 99.  When the second
c  partial derivative of U with respect to x is approximated by central
c  differences, a system of neqn ODEs is obtained for the y_i(t).  The
c  initial values y_i(0) are given by U(x_i,0) = sol(x_i,0).
c
c  A common way to present the computed results is to plot approximations
c  to U(x,tout) on [0, 10] for a selection of times tout.  This example
c  shows how to compute approximations to the y_i(t) at these specific times.
c  They are written to an output file for plotting; the most convenient way
c  to do this will depend on the system and the plotting package used.
c
c  Because an analytical solution U(x,t) is available, the maximum error
c  of the approximation to U(x,tend) is computed and displayed.  Here
c  tend = 15.  It should be appreciated that this error has two parts,
c  one the error made by RKC in the time integration and the other from
c  the spacial discretization.  Some statistics about the integration are
c  also displayed.
c
      integer           neqn,nout
      parameter        (neqn=99,nout=4)
      integer           info(4),idid
      double precision  t,tend,rtol,atol
      double precision  y(neqn),work(8+5*neqn)
      integer           i,next
      double precision  dx,delta,sol,tout(nout),yout(neqn),
     &                  truey,error
      integer           nfe,nsteps,naccpt,nrejct,nfesig,maxm
      common /rkcdid/   nfe,nsteps,naccpt,nrejct,nfesig,maxm
      external          f
c-------------------------------------------------------------
c  Specify the interval of integration in time.  The initial
c  values of the solution at mesh points are provided by the
c  analytical solution sol(x,t).
c-------------------------------------------------------------
```

```
      t = 0d0
      tend = 15d0
      dx = 10d0/(neqn+1)
      do 10 i = 1, neqn
        y(i) = sol(i*dx,t)
10    continue
c-----------------------------------------------------------------------
c  Initialize the output:  Define the times at which solutions are to
c  be reported for plotting and output the number of these times.  A
c  solution is computed on a mesh of equally spaced points in [0, 10].
c  Output the number of points in the mesh and then the mesh itself.
c  Because tout(1) = t, output the initial values for the neqn solution
c  components along with the values given at 0 and 10.
c-----------------------------------------------------------------------
      delta = (tend - t)/(nout-1)
      do 20 i = 1,nout
        tout(i) = t + (i-1)*delta
20    continue
      open(10,file='exaout')
      write(10,'(i10)') nout,neqn+2
      do 30 i = 0,neqn+1
        write(10,'(e10.4)') i*dx
30    continue
      next = 1
      write(10,'(e10.4)') sol(0d0,tout(next))
      write(10,'(e10.4)') y
      write(10,'(e10.4)') sol(10d0,tout(next))
      next = next + 1
c-------------------------------------------------
c  To compute results at specific times, the code
c  must return after each step.  Common choices for
c  info(*) have value 0.
c  info(1) = 0 -- return after each step.
c  info(2) = 0 -- RKC computes the spectral radius.
c  info(3) = 0 -- the Jacobian may not be constant.
c  info(4) = 0 -- ATOL is a scalar.
c-------------------------------------------------
      info(1) = 0
      info(2) = 0
      info(3) = 0
      info(4) = 0
c-------------------------
c  Specify the tolerances.
c-------------------------
      rtol = 1d-4
      atol = rtol
c-----------------------------
c  Initialize the integration.
c-----------------------------
      idid = 0
c---------------------
```

```
c  Take a single step:
c--------------------
40    continue
      call rkc(neqn,f,y,t,tend,rtol,atol,info,work,idid)
c-------------------------------------------------------------
c  Was the step successful?  If not, quit with an explanation.
c-------------------------------------------------------------
      if(idid .gt. 2) then
        write(*,*) ' Failed at t = ',t,' with idid = ',idid
        stop
      endif
c-------------------------------------------------------------
c  To get output at specific points, step towards TEND with RKC
c  until the integration passes the next output point. Compute
c  a result at the point using RKCINT.  There might be several
c  output points in the span of a single step by RKC.
c-------------------------------------------------------------
50    continue
      if(t .ge. tout(next)) then
        call rkcint(work,tout(next),yout)
        write(10,'(e10.4)') sol(0d0,tout(next))
        write(10,'(e10.4)') yout
        write(10,'(e10.4)') sol(10d0,tout(next))
        next = next + 1
        if(next .le. nout) goto 50
      endif
c-------------------------------------
c  Monitor the cost of the integration.
c-------------------------------------
      if(nsteps .ge. 5000) then
        write(*,*) ' Quit because of too much work.'
      endif
c-------------------------------------
c  If not done yet, take another step.
c-------------------------------------
      if(idid .eq. 2) goto 40
c-------------------------------------------------------
c  Done.  Compute the error and report some statistics.
c-------------------------------------------------------
      error = 0d0
      do 60 i = 1, neqn
        truey = sol(i*dx,t)
        error = max(error,abs(y(i) - truey))
60    continue
      write(*,'(/a,d8.1,a,f6.1,a,d8.2)') ' With rtol = atol =',rtol,
     &  ', the maximum error at tend =',tend,' was',error
      write(*,'(a,i5,a)') ' The integration cost',nfe,
     &  ' function evaluations.'
      write(*,'(a,i4,a,i3,a)') ' There were',nsteps,' steps (',
     &  nrejct,' rejected).'
      write(*,'(a,i4)') ' The maximum number of stages used was',maxm
```

```
      end

      double precision function sol(x,t)
c-----------------------------------------------------------
c  An analytical solution to the reaction-diffusion equation.
c-----------------------------------------------------------
      double precision x,t
      double precision v,z
      v = sqrt(0.5d0)
      z = x - v*t
      sol = 1d0/(1d0 + exp(v*z))
      return
      end

      subroutine f(neqn,t,y,dy)
c-------------------------------------------------------------
c  Semi-discretization of reaction-diffusion equation by central
c  differences.  The analytical solution sol(x,t) is used for
c  Dirichlet boundary conditions at x = 0 and x = 10.
c-------------------------------------------------------------
      integer           neqn
      double precision  t,y(neqn),dy(neqn)
      integer           i
      double precision  dx,dxsq,sol
c
      dx = 10d0/(neqn+1)
      dxsq = dx**2
      dy(1) = (sol(0d0,t)- 2d0*y(1) + y(2))/dxsq +
     &                                  (1d0 - y(1))*y(1)**2
      do 10 i = 2,neqn-1
         dy(i) = (y(i-1) - 2d0*y(i) + y(i+1))/dxsq +
     &                                  (1d0 - y(i))*y(i)**2
10    continue
      dy(neqn) = (y(neqn-1)- 2d0*y(neqn) + sol(10d0,t))/dxsq +
     &                                  (1d0 - y(neqn))*y(neqn)**2
      return
      end

      double precision function spcrad(neqn,t,y)
c--------------------------
c  This is a dummy routine.
c--------------------------
      integer          neqn
      double precision t,y(neqn)
      spcrad = 0d0
      return
      end
```

3. Example B

```
c                          Example B
c
c  This is a simplification of Example 1 of B.P. Sommeijer, L.F. Shampine,
c  and J.G. Verwer, RKC: an Explicit Solver for Parabolic PDEs that shows
c  the use of RKC on a substantial problem.  Semi-discretization of the
c  heat equation in three space variables results in 19**3 = 6859 equations.
c  The inhomogeneous term and the boundary conditions have been specified so
c  that there is an analytical solution evaluated in sol(x,y,z,t).  The
c  maximum error of the numerical solution at TEND is measured by comparison
c  to a reference solution computed to high accuracy, so the error reported
c  is the error of the time integration, not the difference between the
c  solutions of the ODEs and the PDE.  Some statistics about the integration
c  are also displayed.
c
c  WARNING: This program expects the file exb.ref containing the reference
c           solution to be present in the same directory.
c
      integer          ndim
      parameter       (ndim=19*19*19)
      integer          info(4),idid
      double precision t,tend,rtol,atol
      double precision y(ndim), work(8+4*ndim)
      integer          neqn,i
      double precision yref(ndim),error
      integer          nfe,nsteps,naccpt,nrejct,nfesig,maxm
      common /rkcdid/  nfe,nsteps,naccpt,nrejct,nfesig,maxm
      integer          nx,ny,nz
      common    /grid/ nx,ny,nz
      external         f
c
      t = 0d0
      tend = 0.7d0
c---------------------------------------
c  Define the mesh and the number of ODEs.
c  Define the initial values.
c---------------------------------------
      nx = 19
      ny = 19
      nz = 19
      neqn = nx*ny*nz
      call exact(neqn,t,y)
c--------------------------------------
c  Load the reference solution at TEND.
c--------------------------------------
      open(10,file='exb.ref')
      read(10,*) yref
c-----------------------------------------------------------------
c  info(1) = 1 -- compute a solution at TEND only.
c  info(2) = 1 -- SPCRAD returns a bound on the spectral radius.
```

```
c  info(3) = 1 -- the Jacobian is constant.
c  info(4) = 0 -- ATOL is a scalar.
c-----------------------------------------------------------------
      info(1) = 1
      info(2) = 1
      info(3) = 1
      info(4) = 0
c
      rtol = 1d-2
      atol = rtol
c
      idid = 0
      call rkc(neqn,f,y,t,tend,rtol,atol,info,work,idid)
c---------------------------------
c  Was the integration successful?
c---------------------------------
      if(idid .ne. 1) then
        write(*,*) ' Failed at t = ',t,' with idid = ',idid
        stop
      endif
c
      error = 0d0
      do 10 i = 1,neqn
          error = max(error,abs(y(i) - yref(i)))
10      continue
      write(*,'(/a,d8.1,a,f6.1,a,d8.2)') ' With rtol = atol =',rtol,
     &  ', the maximum error at tend =',tend,' was',error
      write(*,'(a,i5,a)') ' The integration cost',nfe,
     &  ' function evaluations.'
      write(*,'(a,i4,a,i3,a)') ' There were',nsteps,' steps (',
     &  nrejct,' rejected).'
      write(*,'(a,i4/)') ' The maximum number of stages used was',
     &  maxm
      end

      subroutine exact(neqn,t,y)
      integer          neqn
      double precision t,y(neqn)
      integer          i,j,k,l
      double precision dx,dy,dz,sol
      integer          nx,ny,nz
      common    /grid/ nx,ny,nz
c
      dx = 1d0/(nx+1)
      dy = 1d0/(ny+1)
      dz = 1d0/(nz+1)
      do 30 i = 1,nx
        do 20 j = 1,ny
          do 10 k = 1,nz
            l = i + (j-1)*nx + (k-1)*nx*ny
            y(l) = sol(i*dx,j*dy,k*dz,t)
```

```
10        continue
20      continue
30    continue
      return
      end

      double precision function sol(x,y,z,t)
      double precision x,y,z,t
      double precision arg
      arg = 5d0*(x + 2d0*y + 1.5d0*z - 0.5d0 - t)
      sol = tanh(arg)
      return
      end

      double precision function spcrad(neqn,t,y)
      integer          neqn
      double precision t,y(neqn)
      integer          nx,ny,nz
      common    /grid/ nx,ny,nz
      spcrad = 4d0*((nx+1)**2 + (ny+1)**2 + (nz+1)**2)
      return
      end

      subroutine f(neqn,t,y,dydt)
      integer          neqn
      double precision t,y(neqn),dydt(neqn)
      integer          i,j,k,l
      double precision u(0:20,0:20,0:20),dx,dy,dz,dxsq,dysq,dzsq,
     &                 arg,sh,ch,sol
      integer          nx,ny,nz
      common    /grid/ nx,ny,nz
c
      dx = 1d0/(nx+1)
      dy = 1d0/(ny+1)
      dz = 1d0/(nz+1)
      dxsq = dx*dx
      dysq = dy*dy
      dzsq = dz*dz
      do 30 i = 1,nx
        do 20 j = 1,ny
          do 10 k = 1,nz
            u(i,j,k) = y(i + (j-1)*nx + (k-1)*nx*ny)
10        continue
20      continue
30    continue
c
      do 50 i = 1,nx
        do 40 j = 1,ny
          u(i,j,0) = sol(i*dx,j*dy,0d0,t)
          u(i,j,nz+1) = sol(i*dx,j*dy,1d0,t)
40      continue
```

```
50    continue
c
      do 70 i = 1,nx
        do 60 k = 1,nz
          u(i,0,k) = sol(i*dx,0d0,k*dz,t)
          u(i,ny+1,k) = sol(i*dx,1d0,k*dz,t)
60      continue
70    continue
c
      do 90 j = 1,ny
        do 80 k = 1,nz
          u(0,j,k) = sol(0d0,j*dy,k*dz,t)
          u(nx+1,j,k) = sol(1d0,j*dy,k*dz,t)
80      continue
90    continue
c
      do 120 i = 1,nx
        do 110 j = 1,ny
          do 100 k = 1,nz
            arg = 5d0*(i*dx + 2d0*j*dy + 1.5d0*k*dz - 0.5d0 - t)
            sh = sinh(arg)
            ch = cosh(arg)
            l = i + (j-1)*nx + (k-1)*nx*ny
            dydt(l) = (u(i-1,j,k) - 2d0*u(i,j,k) + u(i+1,j,k))/dxsq +
     &                (u(i,j-1,k) - 2d0*u(i,j,k) + u(i,j+1,k))/dysq +
     &                (u(i,j,k-1) - 2d0*u(i,j,k) + u(i,j,k+1))/dzsq +
     &                (-5d0*ch + 362.5d0*sh)/(ch**3)
100       continue
110     continue
120   continue
c
      return
      end
```