Runge-Kutta-Chebyshev Methods for Advection-Diffusion-Reaction Problems

LIMEI ZHANG

KTH Numerical Analysis
and Computer Science

Master's Degree Project
Stockholm, Sweden 2004

TRITA-NA-E04151

Numerisk analys och datalogi
KTH
100 44 Stockholm

Department of Numerical Analysis
and Computer Science
Royal Institute of Technology
SE-100 44 Stockholm, Sweden

Runge-Kutta-Chebyshev Methods for Advection-Diffusion-Reaction Problems

LIMEI ZHANG

TRITA-NA-E04151

Master's Thesis in Numerical Analysis (20 credits)
at the Scientific Computing International Master Program,
Royal Institute of Technology year 2004
Supervisor at Nada was Michael Hanke
Examiner was Axel Ruhe

Abstract

This project is devoted to two Matlab solvers for the time integration of advection-diffusion-reaction equations discretized by the method of lines. They are based on two Runge-Kutta-Chebyshev methods (RKC). If the reaction terms are highly stiff, then the implicit-explicit Runge-Kutta-Chebyshev method can be used, otherwise the explicit second-order Runge-Kutta-Chebyshev method. Remarkable properties of RKC methods make it possible for the two programs to select at each step the most efficient stable formula as well as the most efficient step size. These characteristics of both programs make it especially attractive for problem in several spatial variables.

One code named RKC1, which uses the explicit Runge-Kutta-Chebyshev method, is compared to `oed45` on a test problem in one spatial variable. And the comparison between the other code named RKC2 and `ode23s` is done on a different test problem in 1-D.

Runge-Kutta-Chebyshev metoder för advektion-diffusion-reaktions ekvationer

Sammanfattning

Detta arbete rör teå Matlab-lösare för tidintegerering av advektion-diffusion-reaktions ekvationer diskretiserade med “ method lines”. De är baserade på två Runge-Kutta-Chebyshev metoder (RKCmetoder). Om reaktionstermerna är mycket styva kan den implicit-explicita RKCmetoden användas, annars används den explicita, andra ordningens RKCmetoden. Speciella egenskaper hos RKC-metoderna gör det möjligt för de två lösarna att i varje steg välja den mest effektiva stabila formuleringen och den mest effektiva stegländen. Dessa egenskaper hos de båda lösarna gör dem speciellt lämpliga för problem i flera rumsvariabler.

Vår kod RKC1, som använder den explicita RKC-metoden, jämförs med *ode*45 på ett testproblem i en rumdimension. Jämförelsen mellan den andra koden, RKC2, och `ode23s` görs på ett annat 1-D testproblem.

Contents

Acknowledgement

I thanks my supervisor, Michael Hank, for his continuous support and his fruitful guidance
I wish also to thank Jesper Oppelstrup and Axel Ruhe for many helpful disscussions.
I would like to my friends and especially Xiao Heng and Cui Zejun.
I am thankful my mother and daughter for their love and support.
Lastly, I want to thank my husband for his help and consideration.

Chapter 1

Introduction

This work focuses on implementing two Matlab programs that solve the time integration of stiff, nonlinear advection-diffusion-reaction equation with two RKC methods, respectively. Adopting the method of lines approach we assume that the PDE system with its boundary conditions has been spatially discretized, and thus we focus our research on ODE systems

$$w'(t) = F(t, w(t)), \quad t > 0, \quad w(0) = w_0. \tag{1.1}$$

representing semi-discrete advection-diffusion-reaction problems. In most practical applications the dimension of this ODE system is huge, especially for multi-space dimensional PDEs and/or PDE systems with many reacting species. The huge dimension and the simultaneous occurrence of advection, diffusion, reaction terms and stiffness can severely complicate the use of standard implicit integrators leaning on modified Newton and (preconditioned iterative) linear solvers. On the other hand, the stiffness included by diffusion and reaction terms rules out easy-to-use standard explicit solvers.

Historically the principal goal when constructing Runge-Kutta methods was to achieve the highest order possible with a given number of stages s. Stabilized methods like RKC are different in that only a few stages are used to achieve a usually low order whereas additional stages are exploited to increase the region of absolute stability, depending on the particular application.

The original explicit RKC method is a stabilized second-order integration method for pure diffusion problems. Recently it has been extended in an implicit-explicit manner to also incorporate highly stiff reaction terms. This implicit-explicit RKC method thus treats diffusion terms explicitly and highly stiff reaction terms implicitly. And this diffusion terms can be treated simultaneously and explicitly with the advection terms. So the system (1.1) can be modified to

$$w'(t) = F_E(t, w(t)) + F_I(t, w(t)), \quad t > 0, \quad w(0) = w_0, \tag{1.2}$$

where F_I represents the reaction term and F_E contains other terms. In these codes for two problems, the step size is selected automatically to achieve both reliability and efficiency. At each step the estimated local error, which comes from an asymptotically correct estimate, is controlled so that the accuracy tolerances specified are met. Through this procedure a optimal step size τ is given. Because the length of stability domain $\beta(s)$ is approximated well, the smallest number of stages can be found for the step size τ. Therefore the formula of each step is variable and efficient.

Chapter 2 reviews some basis concept of the differential equations. There are ordinary differential equation (ODE), partial differential equation (PDE), ODE system, initial condition, boundary condition and time-dependent PDE. A example of a 3-D advection-diffusion-reaction equation is given.

Both are Euler method and Runge-Kutta method are described in chapter 3. The regions of absolute stability of these methods are discussed in detail. So that we can understand the stabilized Runge-Kutta methods. Finally, the manner which is used to select step size automatically is introduced.

Chapter 4 includes the implementation of two RKC methods, the result of tests and comparisons, programs as well as conclusion.

Chapter 2

Differential Equation

Differential equation are the most common and important mathematical model in science and engineering. Important areas include to describe the phenomena such as population dynamics, the dispersion of pollutants in the oceans and atmosphere, even satellite orbits about the earth, and many, many more.

A differential equation is a relation between a function and its derivatives. If the function w depends on only one variable t, i.e. $w = w(t)$, the differential equation is called *ordinary*. If w depends on at least two variables t and x, i.e. $w = w(t)$, the differential equation is called *partial*.

2.1 Ordinary Differential Equation

An ordinary differential equation (ODE) of *first order* has the following form

$$w' = f(t,w) \tag{2.1}$$

with only one independent variable which in this case is t. The solution $w(t)$ of (2.1) isn't unique unless an *initial condition*

$$w(t_0) = w_0.$$

The differential equation (2.1) together with the initial condition is called an *initial value problem* (IVP).

An ordinary differential equation of *second order* can be written as

$$w'' = f(x,w,w') \tag{2.2}$$

The two conditions are required if the solution $w(t)$ is unique. One way is to specify the *boundary values* of x i.e.

$$w(a) = w_0, \quad w(b) = w_1,$$

where $x \in [a,b]$. The differential equation (2.2) and these boundary values constitute a *boundary value problem*(BVP).

For a system of ODEs

$$\begin{aligned}\frac{dw_1}{dt} &= f_1(t, w_1, w_2, ..., w_n)\\ \frac{dw_2}{dt} &= f_2(t, w_1, w_2, ..., w_n)\\ &\qquad\qquad\qquad\qquad ......\\ \frac{dw_n}{dt} &= f_n(t, w_1, w_2, ..., w_n)\end{aligned}$$

it is practical to use vector formula

$$\frac{d\mathbf{w}}{dt} = f(t, \mathbf{w})$$

where the dependent variables are collected in the vector $\mathbf{w} = (w_1, w_2, ..., w_n)^T$ and the right hand sides in $\mathbf{f} = (f_1, f_2, ..., f_n)^T$.

2.2 Partial Differential Equation

A general formula of a *partial differential equation* (PDE) is

$$F(t, x, y, z, w, w_x, w_y, w_z, w_{xx}, ...) = 0. \tag{2.3}$$

The variable t, x, y, z, the *independent* variables, are defined in some region, bounded or unbounded, and the variable w, the *dependent* variable, is a solution of (2.4) if it satisfies the PDE at all t, x, y, z in the region. Often the independent variables t is used to denote *time* and x, y, z denote the *space* variables. A problem formulated in the three space dimensions, x, y, z is called a *3-D* problem. If time is among the independent variables, the PDE-problem is called an *evolution* or *time-dependent* problem.

The formulation (2.4) is very general. Therefore a set of PDE's of much simpler form is presented as generic problems. A 3-D time-dependent advection-diffusion-reaction equation is expressed as

$$w_t + a_1 w_x + a_2 w_y + a_3 w_z = d\Delta w + f(w) \tag{2.4}$$

where a_k are given constants, d is a given constant diffusion coefficient and $f(w)$ is a reaction term.

A time-dependent PDE can be transformed into a large system of ODEs by replacing the spatial derivative in one or more dimensions by a discrete approximation (via finite difference, finite volume, or finite element methods). So the equation can be given by the form (2.1). And this form is in good agreement with those in (1.1) and (1.2).

A PDE has infinitely many solutions. To obtain a unique solution it is necessary to add to the PDE problem *initial* and *boundary* conditions.

Chapter 3

Numerical method

In general it is not possible to find the analytical solution of a differential equation analytically. So the numerical methods are used to obtain the trajectories of this problem.

The aim of this chapter is to introduce several numerical method of solving ODEs.

3.1 Euler method

To approximate (2.1) with $t \in [0, b]$ and $w(0) = c$, the interval of integration is discretized by a mesh

$$0 = t_0 < t_1 < ... < t_{n-1} < t_n = b$$

and let $h_n = t_n - t_{n-1}$ be the nth step size. So the following approximations are constructed

$$w_0 (= c), w_1, ..., w_{n-1}, w_n$$

where w_n is an intended approximation of $w(t_n)$. Use the Taylor expansion, obtain:

$$w(t_n) = w(t_{n-1}) + h_n w'(t_{n-1}) + O(h_n^2).$$

The *explicit Euler* method

$$w_n = w_{n-1} + h_n f(t_{n-1}, w_{n-1}) \tag{3.1}$$

is derived. Using the same way the *implicit Euler* method is obtained by

$$w_n = w_{n-1} + h_n f(t_n, w_n). \tag{3.2}$$

Absolute stability Assume the scalar *test equation*

$$w' = \lambda w, \quad w_0 = 1 \tag{3.3}$$

where λ is a complex constant. The exact solution is

$$w(t) = e^{\lambda t},$$

whereas explicit Euler method, with a uniform step size $h_n = h$ and $z = h\lambda$, gives

$$\begin{aligned} w_n &= w_{n-1} + h\lambda w_{n-1} \\ &= (1+z)w_{n-1} \\ &= R(z) \cdot w_{n-1} \\ &= \ldots \\ &= (1+z)^n \end{aligned}$$

If $Re(\lambda) < 0$, then $|w_n|$ decays exponentially. This problem is asymptotically stable, and it yield an additional *absolute stability* requirement,

$$|w_n| \leq |w_{n-1}|, \quad n = 1, 2, \ldots. \tag{3.4}$$

For a given numerical method, the *region of absolute stability* is the region of the complex z-plane such that applying the method for the test equation (3.3), yields an approximate solution satisfying the absolute stability requirement (3.4).

For the explicit Euler method we obtain the condition

$$|R(z)| \leq 1 \Rightarrow |1+z| \leq 1$$

which yields the region of absolute stability. Applying the implicit Euler method (3.2) to the test equation, obtain

$$\begin{aligned} w_n &= w_{n-1} + h\lambda w_n \\ w_n &= \frac{1}{1-z} w_{n-1} \\ &= R(z) \cdot w_{n-1} \end{aligned}$$

Therefore satisfy

$$|R(z)| \leq 1 \Rightarrow |1-z| \geq 1.$$

The stability domain of implicit Euler method is larger. So for a given stiff problem, the implicit Euler method needs fewer steps than the explicit Euler method unless accuracy is restricting the stepsize. But the unknown vector w_n at each step appears on both sides of the equation (3.2), generally in a nonlinear expression. Consequently, a nonlinear system of algebraic equations has to be solved at each step. It is so expensive in terms of computing time.

3.2 Runge-Kutta method

Both Euler methods are first order methods. In order to develop efficient, highly accurate approximation algorithm, higher-order difference methods are designed.

Assume $t_{n-1/2} = t_n - \frac{1}{2}h$, then

$$w(t_n) = w(t_{n-1/2}) + \frac{h}{2}w'(t_{n-1/2}) + \frac{h^2}{8}w''(t_{n-1/2}) + \frac{h^3}{48}w'''(t_{n-1/2}) + \dots$$

and

$$w(t_{n-1}) = w(t_{n-1/2}) - \frac{h}{2}w'(t_{n-1/2}) + \frac{h^2}{8}w''(t_{n-1/2}) - \frac{h^3}{48}W'''(t_{n-1/2}) + \dots$$

which divided by h and subtracted, they given

$$\frac{w(t_n) - w(t_{n-1})}{h} = w'(t_{n-1/2}) + \frac{h^2}{24}w'''(t_{n-1/2}) + O(h^4).$$

Finally, by replaying $w'(t_{n-1/2})$ with $\frac{1}{2}(w'(t_n) + w'(t_{n-1}))$ and w' with f, we obtain the *implicit trapezoidal method*

$$w_n = w_{n-1} + \frac{h}{2}(f(t_n, w_n) + f(t_{n-1}, w_{n-1})) \tag{3.5}$$

which is an implicit *second order Runge-Kutta* (RK) method. To obtain an explicit method based on this idea, we can approximate w_n in $f(t_n, w_n)$ by the explicit Euler method, yielding

$$\begin{aligned} \hat{w} &= w_{n-1} + hf(t_{n-1}, w_{n-1}), \\ w_n &= w_{n-1} + \frac{h}{2}(f(t_n, \hat{w}_n) + f(t_{n-1}, w_{n-1})), \end{aligned}$$

This is called the *explicit trapezoidal* method which is an explicit second order RK method.

The formula of *classical fourth-order* Runge-Kutta method is given by

$$\begin{aligned} W_1 &= w_{n-1}, \\ W_2 &= w_{n-1} + \frac{h}{2}f(t_{n-1}, W_1), \\ W_3 &= w_{n-1} + \frac{h}{2}f(t_{n-1/2}, W_2), \\ W_4 &= w_{n-1} + hf(t_{n-1/2}, W_3), \\ w_n &= w_{n-1} + \frac{h}{6}(f(t_{n-1}, W_1) + 2f(t_{n-1}, W_2) \\ &\quad +2f(t_{n-1}, W_3) + f(t_{n-1}, W_4)). \end{aligned}$$

General formulation of RK method In general an s-stage Runge-Kutta method for the ODE system

$$w' = f(t,w)$$

can be written in the form

$$\begin{aligned} W_i &= w_{n-1} + h\sum_{j=1}^{s} a_{ij} f(t_{n-1}+c_j h, W_j), \quad 1 \le i \le s \\ w_n &= w_{n-1} + h\sum_{i=1}^{s} b_i f(t_{n-1}+c_i h, W_i). \end{aligned}$$

The method can be represented conveniently in a shorthand notation

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}$$

where

$$c_i = \sum_{j=1}^{s} a_{ij}, \quad i = 1,\dots,s.$$

The Runge-Kutta methods is *explicit* if $a_{ij} = 0$ for $j \ge i$.

Absolute stability Solving the test equation (3.3) with explicit s-stage Runge-Kutta method, obtain:

$$\begin{aligned} W_i &= w_{n-1} + z\sum_{j=1}^{s} a_{ij} W_j, \\ w_n &= w_{n-1} + z\sum_{i=1}^{s} b_i W_i. \end{aligned}$$

Let A be the $s \times s$ coefficient matrix from the tableau defining the Runge-Kutta method , $\mathbf{b}^T = (b_1, b_2, \dots, b_s)$ and $\mathbf{1} = (1,1,\dots,1)^T$

$$\begin{aligned} w_n &= [1 + z\mathbf{b}^T(I - zA)^{-1}\mathbf{1}]w_{n-1} \\ &= [1 + z\mathbf{b}^T(I + zA + \dots + z^k A^k + \dots)\mathbf{1}]w_{n-1} \\ &= R(z)\cdot w_{n-1} \end{aligned}$$

Because the exact solution is $w(t) = e^{\lambda t}$, we have

$$w(0) = 1, \quad w'(0) = \lambda, \quad w''(0) = \lambda^2, \quad \dots, \quad w^{(n)}(0) = \lambda^n, \quad \dots$$

so for an s-stage explicit method of order p, get

$$R(z) = 1 + z + \frac{z^2}{2} + \ldots + \frac{z^p}{p!} + \sum_{j=p+1}^{s} z^j \mathbf{b}^T A^{j-1} \mathbf{1}.$$

If $s = p$, then the region of absolute stability of an explicit pth-order Runge-Kutta method is given by

$$R(z) \leq 1 \Rightarrow |1 + h\lambda + \frac{(h\lambda)^2}{2} + \ldots + \frac{(h\lambda)^p}{p!}| \leq 1$$

This is a very small stability domain.

The explicit Runge-Kutta-Chebyshev method For many problems, usually not very stiff, of large dimension, and eigenvalues known to lie in a certain region, explicit methods with large stability domains can be very efficient. It was found that the stability domains are extended along the negative real axis if $R(z)$ for these methods are the Chebyshev polynomials

$$T_s(x) = \cos(s \arccos x) \tag{3.6}$$

or

$$T_s(x) = 2xT_{s-1}(x) - T_{s-2}(x), \quad T_0(x) = 1, \quad T_1(x) = x. \tag{3.7}$$

which remain for $-1 \leq x \leq 1$ between -1 and $+1$ and among these polynomials have the largest possible derivative $T_s'(1) = s^2$. Therefore one must set

$$R_s(z) = T_s(1 + z/s^2)$$

so that $R_s(0) = 1$, $R_s'(0) = 1$, and $|R_s(z)| \leq 1$ for $-2s^2 \leq z \leq 0$. In the points where $T_s(1 + z/s^2) = \pm 1$, there is no damping at all of the higher frequencies and the stability domain has zero width. Choose a small $\varepsilon > 0$, and put

$$R_s(z) = \frac{1}{T_s(\omega_0)} T_s(\omega_0 + \omega_1 z), \quad \omega_0 = 1 + \frac{\varepsilon}{s^2}, \quad \omega_1 = \frac{T_s(\omega_0)}{T_s'(\omega_0)},$$

these polynomial oscillate between approximately $1 - \varepsilon$ and$1 + \varepsilon$ and again satisfy $R_s(z) = 1 + z + O(z^2)$. But the stability domains become a bit shorter.

An elegant idea for a second realization has been found by van der Houwen and Sommeijer (1980). Set

$$R_s(z) = a_s + b_s T_s(\omega_0 + \omega_1 z) \quad w_0 = 1 + \varepsilon/s^2, \quad \varepsilon \approx 0.15.$$

The condition for second order

$$R_s(0) = 1, \quad R_s'(0) = 1, \quad R_s''(0) = 1$$

lead to

$$\omega_1 = \frac{T_s'(\omega_0)}{T_s''(\omega_0)}, \quad b_s = \frac{T_s''(\omega_0)}{(T_s'(\omega_0))^2}, \quad a_s = 1 - b_s T_s(\omega_0).$$

For internal stages put

$$R_j(z) = a_j + b_j T_j(\omega_0 + \omega_1 z), \quad j = 0, 1, ..., s-1.$$

It has been discovered that these $R_j(z)$ can, for $j \geq 2$, be approximations of second order at center points $t_0 + c_j h$ if

$$R_j(0) = 1, \quad R_j'(0) = c_j, \quad R_j''(0) = c_j^2$$

which gives

$$R_j(z) - 1 = b_j(T_j(\omega_0 + \omega_1 z) - T_j(\omega_0)), \quad b_j = \frac{T_j''(\omega_0)}{(T_j'(\omega_0))^2}.$$

The three-term recurrence relation(3.7) now lead to

$$R_j(z) - 1 = \mu_j(R_{j-1}(z) - 1) + \nu_j(R_{j-2}(z) - 1) + \tilde{\mu}_j \cdot z \cdot (R_{j-1}(z) - a_{j-1})$$

where

$$\mu = \frac{2b_j\omega_0}{b_{j-1}}, \quad \nu = \frac{-b_j}{b_{j-2}}, \quad \tilde{\mu} = \frac{2b_j\omega_1}{b_{j-1}}, \quad j = 2, 3, ..., s.$$

This formula allows, in the case of a nonlinear differential , system, to define the scheme

$$\begin{aligned}
g_0 - w_0 &= 0, \\
g_1 - w_0 &= \tilde{\mu}_1 h f(g_0), \\
g_j - w_0 &= \mu_j(g_{j-1} - w_0) + \nu_j(g_{j-2} - w_0) + \tilde{\mu}_j h f(g_{j-1}) + \tilde{\mu}_j a_{j-1} h f(g_0),
\end{aligned}$$

which, being of second order for $w' = \lambda w$, is of second order for nonlinear equations too.

Let τ be the step size (i.e. $\tau = t_{n+1} - t_n$), for the problem (1.1) the second-order explicit RKC formula has the form:

$$\begin{aligned}
W_0 &= w_n, \\
W_1 &= W_0 + \tilde{\mu}_1 \tau F_0, \\
W_j &= (1 - \mu_j - \nu_j) W_0 + \mu_j W_{j-1} + \nu_j W_{j-2} + \tilde{\mu}_j \tau F_{j-1} + \tilde{\gamma}_j \tau F_0, \\
j &= 2, \ldots, s, \\
w_{n+1} &= W_s
\end{aligned}$$

where W_k are internal vectors and F_k denotes $F(t_n + c_k \tau, W_k)$. The additional coefficient is

$$\tilde{\gamma}_j = -a_{j-1} \tilde{\mu}_j.$$

And put

$$b_0 = b_2, \quad b_1 = 1/\omega_0$$

$$c_0 = 0, \quad c_1 = c_2, \quad c_j = \frac{T_s'(\omega_0)}{T_s''(\omega_0)} \frac{T_j''(\omega_0)}{T_j'(\omega_0)}, \quad c_s = 1, \quad 2 \leq j \leq s-1.$$

This method is a stabilized method that uses two stages to obtain second order whereas additional stages are exploited to increase the length of the stability domain along the negative real axis $\beta(s)$.

The implicit-explicit RKC method An implicit-explicit extension of the explicit RKC scheme designed for parabolic PDEs is proposed for diffusion-reaction problems with severely stiff terms. The implicit-explicit treats these reaction terms implicitly and diffusion terms explicitly.

Therefore the system (1.1) can be transformed into the form (1.2). For this IMEX scheme all the stage function $R_j(z_E, z_I)$ are take to be the form

$$R_j(z_E, z_I) = a_j + b_j T_j \left(\frac{\omega_0 + \omega_1 z_E}{1 - \frac{\omega_1}{\omega_0} z_I} \right)$$

where $z_E = \tau \lambda_E$ and $z_I = \tau \lambda_I$. Consider the scalar stability test equation (3.3) with λ_E and λ_I standing the eigenvalues of Jacobian $F_E'(t, w(t))$ and $F_I'(t, w(t))$, respectively. The IMEX RKC method is given by

$$\begin{aligned}
W_1 &= w_n, \\
W_1 &= W_0 + \tilde{\mu}_1 \tau F_{E,0} + \tilde{\mu}_1 \tau F_{I,1}, \\
W_j &= (1 - \mu_j - \nu_j) W_0 + \mu_j W_{j-1} + \nu_j W_{j-2} + \tilde{\mu}_j \tau F_{E,j-1} + \tilde{\gamma}_j \tau F_{E,0} + \\
&\quad (\tilde{\gamma}_j - (1 - \mu_j - \nu_j) \tilde{\mu}_1) \tau F_{I,0} - \nu_j \tilde{\mu}_1 \tau F_{I,j-2} - \tilde{\mu}_1 \tau F_{I,j}, \\
w_{n+1} &= W_s,
\end{aligned}$$

where $F_{E,j}$ denotes $F_E(t_n + c_j \tau, W_j)$.

3.3 Error estimation and control

The error is simply the difference between the true solution and the numerical solution. It is due to the differential approximation, and can't be eliminated. But it can be decreased. A good idea is to estimate and control the local error or the local truncation error, rather than the global error. Basically, by specifying an error tolerance tol_k one can require a more accurate approximate solution or a less accurate solution. The result is to select the next step size automatically, so that both reliability and efficiency are achieved.

Variable step size Assume a step size, h_n, have been chosen, and $\mathbf{l}_n$ is a estimations of the local error , this error satisfy

$$|\mathbf{l}_{n,k}| \leq tol_k, \quad tol_k = atol_k + max(|y_{n,k}|, |y_{n-1,k}|) \cdot rtol_k$$

where $atol_k$ is an absolute error tolerance and $rtol_k$ is a relative error tolerance which are prescribed by the user. $y_{n,k}$ is the k-th component of y_n and $1 \leq k \leq m$. As a measure of the error we take

$$err_n = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(\frac{\mathbf{l}_{n,i}}{tol_i}\right)^2};$$

other norms, such as the max norm, are also of frequent use. Then the optimal step size is obtained as

$$h_{opt} = h_n \cdot (err_{n-1})^{\beta}/(err_n)^{\alpha}. \tag{3.8}$$

When q is the order of this method, A fairly good choice, found by Gustafsson, is

$$\alpha \approx 0.7/(q+1), \quad \beta \approx 0.4/(q+1).$$

And if this is the second step, choose:

$$\alpha = 1/(q+1), \quad \beta = 0.$$

Some care is now necessary for a good code to multiply (3.8) by a safety factor fac, usually $fac = 0.8, 0.9$ and so on, so that the error will be acceptable the next time with high probability. Further h is not allowed to increase nor to decrease too fast, so we put

$$h_{new} = h_n \cdot \min(facmax, \max(facmin, fac \cdot (err_{n-1})^{\beta}/(err_n)^{\alpha}))$$

for the new step size. Because the step size h_{n-1} has been accepted, if $err_n \leq 1$ then the computed step is accepted and the solution is advanced with y_n and a new step is tried with h_{new} as step size. Else, the step is rejected and the computation are repeated with the new step size h_{new}. [6]

Initial step size Determining a good starting step size is important. Otherwise it is repaired by the step size control. The algorithm, which is furnished to determined the starting step size of pth-order RK method, for $y' = f(t,y)$ is the following one: [5]

1. Do the function evaluate $f(t_0,y_0)$ at the initial point.
2. Put $d_0 = norm(y_0)$ and $d_1 = norm(f(t_0,y_0))$.
3. If d_0 and d_1 are bigger than 10^{-5}, then $h_0 = 0.01 \cdot (d_0/d_1)$ or else $h_0 = 10^{-6}$.
4. Perform one explicit Euler step, $y_1 = y_0 + h_0 f(t_0,y_0)$.
5. Compute $f(t_0 + h_0, y_1)$
6. Compute $d_2 = norm(f(t_0+h_0,y_1) - f(t_0,y_0))$ as an estimation of the second derivative of the solution.
7. If $\max(d_1,d_2) \leq 10^{-15}$, then $h_1 = \max(10^{-6}, h_0 \cdot 10^{-3})$ or else $h_1 = (0.01/\max(d_1,d_2))^{1/(p+1)}$.
8. Choose the starting step size $h = \min(100 \cdot h_0, h_1)$.

Chapter 4

Matlab Implementation

4.1 RKC1 function

RKC1 is a variable step size, variable formula code that uses explicit Runge-Kutta-Chebyshev formulas to solve efficiently a class of large systems of mildly stiff ordinary differential equations (ODEs). The systems arising when an advection-diffusion-reaction equation (PDE) is approximated by semi-discre -tization exemplify the problems for which RKC1 is intended. This ODE system has the form (1.1).

The family of formulas implemented in RKC1 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.

Error control Let τ be stepsize and Est_{n+1} be the approximation of the local error, obtain an asymptotically correct estimate:

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

where $S_n = F_I'(w(t_n))F(w(t_n))$. There are a scalar relative error tolerance $rtol$ and a scalar absolute error tolerance $atol$. These tolerances are used in the weighted RMS norm

$$\| Est_{n+1} \| = \| \omega^{-1} Est_{n+1} \|_2, \quad \omega = \sqrt{m} \cdot diag(Tol_1, \cdots, Tol_m),$$

where

$$Tol_k = atol + rtol|w_{n+1,k}|,$$

m is the dimension of the ODE system and $w_{n+1,k}$ the k-th component of w_{n+1}. Hence the step is accepted if $\| Est_{n+1} \| \leq 1$ and otherwise rejected and redone. The prediction for the new step size after a successful step is given by

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

withe 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 predictions is obtained by deleting the parenthesized term. It is used after a step reject.

Initial step size 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 at a cost of a single function evaluation:

$$\tau_0 = 1/\sigma(F'(t_0, w_0)), \quad Est = \tau_0(F(t_0 + \tau_0, w_0 + \tau_0 F(t_0, w_0)) - F(t_0, w_0))$$

where σ denotes the spectral radius . 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 this method 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 formula used are strips containing a segment of the negative real axis, and for $\varepsilon = 10$ the length of the segment $\beta(s)$ is approximated well by

$$\beta(s) = \begin{cases} 2, & s = 2 \\ (s^2 - 1)(0.340 + 0.189(2/(s-1))^{1.3}), & s \geq 3 \end{cases}$$

Assuming that the eigenvalue of local Jacobian 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

$$\tau\sigma(F'(t, w)) \leq \beta(s).$$

4.2 RKC2 function

RKC2 is a variable step size, variable formula code too. It is based on the implicit-explicit RKC method that advection and diffusion terms are treated simultaneously and explicitly and the highly stiff reaction terms implicitly. The ODE system used is shown in formula (1.2). Because in this system two terms, an implicit term and an explicit term, are included, this formula is more complicated than that of the explicit RKC method. Fortunately, except the formula there are only two different parts. One is that additional term is introduced to the local error which is proportional to $\tau^2/(s^2-1)$, the other is the used spectral radius $\sigma(F_E'(w(t_n)))$.

Error control The estimation of the local error is

$$Est_{n+1} = 1/15[12(w_n - w_{n+1}) + 6\tau(F(w_n) + F(w_{n+1}))] - \eta_s \tau S_n$$

where $S_n = F_I'(w(t_n))F(w(t_n))$. $F_I'(w(t_n))$ denotes the Jacobian of F_I. And

$$\eta_s = \begin{cases} 1, & s = 2 \\ 3/(s^2-1), & s \geq 3 \end{cases}$$

where s is the stage which is obtained by the following inequation

$$\tau\sigma(F_E'(t,w)) \leq \beta(s),$$

with

$$\beta(s) = \begin{cases} 2, & s = 2 \\ (s^2-1)(0.340 + 0.189(2/(s-1))^{1.3}), & s \geq 3. \end{cases}$$

Initial step size A tentative size is expressed by

$$\tau_0 = 1/\sigma(F_E'(t_0, w_0)).$$

4.3 Test examples

test problem

1. The test problem is the following:

$$u_t = u_{xx} + u, \quad 0 \leq x \leq 1, \quad 0 \leq t \leq 0.5$$

with initial condition $u(0,x) = \sin x$, and boundary condition $u(t,0) = u(t,1) = 0$.

The exact solution is $u(t,x) = e^{(1-\pi^2)t}\sin(\pi x)$. The right hand of differential equation is discretized by three points central differences, so obtain:

$$\begin{aligned} u_{xx} + u &= \frac{u_{i+1} - 2u_i - u_{i-1}}{\Delta x^2} + u_i \\ &= \frac{1}{\Delta x^2}u_{i+1} - \frac{2}{\Delta x^2}u_i + \frac{1}{\Delta x^2}u_{i-1} + u_i \end{aligned}$$

because

$$u_t = u_{xx} + u$$

so

$$\begin{aligned} F(u(t),t)_i &= u_t(x_i,t) \\ &= \frac{1}{\Delta x^2}u_{i+1} - (1 - \frac{2}{\Delta x^2})u_i + \frac{1}{\Delta x^2}u_{i-1} \\ i &= 1,\ldots,n-1. \end{aligned}$$

A uniform grid with spacing $h = 0.025$ is used. An approximative bound for the spectral radius of the Jacobian can be found easily by using Gersgorin's circle theorem. For $h = 0.025$, $\sigma \approx 6393$, so this problem is rather stiff for RKC1.

The result reported here were computed with scalar tolerances $tol = rtol = atol$. Table 4.1 shows results of RKC1. For a range of tol, it presents the following quantities: the integration error at the end of the integration measured in the L_2 norm, the total number of accepted steps with the number of rejected ones parenthesized, the total number of F-evaluations, and CPU time in seconds. The error in the table is the difference between the numerical solutions and exact solutions of the spatially discretized problem. Table 4.2 presents the results for `ode45`.

tol	error	steps	F-evals	cputime
1e-1	0.0229	54(18)	3158	1.516
1e-2	0.0201	43(17)	2472	1.450
1e-3	0.023	54(13)	2417	1.050
1e-4	0.0236	80(9)	2319	1.220
1e-5	0.0238	136(4)	2456	1.360
1e-6	0.0239	283(0)	3409	2.320
1e-7	0.0239	628(0)	5574	4.590

Table 4.1. Results for RKC1

Both RKC1 and `ode45` successfully solve the problem for all the tolerances of Table 4.1 and 4.2. We can't obtain solutions from `ode45` as $tol \leq 10^{-3}$. The results of RKC2 are satisfactory when tol changes from 10^{-1} to 10^{-5} due to less cputime and error.

tol	error	steps	F-evals	cputime
1e-4	0.0239	1509(103)	9673	1.870
1e-5	0.0239	1512(97)	9655	1.850
1e-6	0.0239	1518(102)	9721	1.970
1e-7	0.0239	1530(95)	9751	2.060

Table 4.2. Results for ode45

2. The test problem is the following:

$$u_t = u_{xx} - 10u, \quad 0 \le x \le 1, \quad 0 \le t \le 1.2$$

with initial condition $u(0,x) = \sin x$, and boundary condition $u(t,0) = u(t,1) = 0$.

The exact solution is $u(t,x) = e^{(-10-\pi^2)t}\sin(\pi x)$. A uniform grid with spacing $h = 0.1$ is used. An approximative bound for the spectral radius of the Jacobian can be found easily by using Gersgorin's circle theorem. For $h = 0.1$, $\sigma \approx 393$. Because the reaction term is highly stiff, RKC2 is used to solve this problem.

So $F(u(t),t)_i$ can be defined by

$$F(u(t),t)_i = F_E(u(t),t) + F_I(u(t),t)$$

where

$$\begin{aligned} F_E(u(t),t)_i &= \frac{1}{\Delta x^2}u_{i+1} - \frac{2}{\Delta x^2}u_i + \frac{1}{\Delta x^2}u_{i-1}, \\ F_I(u(t),t)_i &= -10u_i. \end{aligned}$$

The result reported here were computed with scalar tolerances $tol = rtol = atol$. Table 4.3 shows results of RKC2. Table 4.4 presents the results for `ode23s`. Tables 4.3, 4.4 present results in the same way as for the first example.

tol	error	steps	F-evals	cputime
1e-1	0.0031	9(0)	313	0.128
1e-2	9.3834e-5	18(1)	447	0.270
1e-3	2.5593e-5	52(1)	669	0.366
1e-4	1.0718e-6	168(1)	1481	1.140
1e-5	4.2982e-7	527(1)	3958	3.036
1e-6	2.2976e-7	1662(1)	11887	9.100
1e-7	8.4996e-9	5249(1)	36968	27.360

Table 4.3. Results for RKC2

Both RKC2 and `ode23s` successfully solve the problem for all the tolerances of Table 4.3 and 4.4. The solutions can be obtained by `ode23s` as $tol \ge 10^{-5}$.

tol	error	steps	F-evals	cputime
1e-6	5.2508e-9	193(0)	2713	0.580
1e-7	4.8738e-9	406(0)	5847	1.830

Table 4.4. Results for ode23s

4.4 Conclusion

We made two Matlab solvers for the time integration of advection-diffusion-reaction equations. RKC1, which uses explicit RKC method, can solve mildly stiff advection-diffusion-reaction problem. RKC2, which is based on implicit-explicit RKC method, can deal with advection-diffusion-reaction problem with highly stiff reaction terms. Both are variable step size, variable fomula codes.

As $0 \leq t \leq 0.5$ and $tol \geq 1e-5$, RKC1 with less cputime and smaller error is more efficient than `ode45`. Both RKC1 and RKC2 are available to solve problem in coarse tolerances.

A more rigorous implementation in order to handle more complex equations will be necessary.

Appendix A

1.Source of RKC1

```
%
    % The functions RKC1 integrates initial value problem for
    % system of first order differential equations. It is based on a
    % family of explicit Runge-Kutta-Chebyshev formulas of order two.
    % An estimate of the spectral radius is used at step to select the
    % smallest stage s resulting in a stable integration. RKC1 is approximate
    % for the solution to modest accuravy of mildly stiff problems with
    % eigenvalues of Jacobian that are close to the negative real axis.
    %
    %

function [w,t]=rkc1(f,tspan,tol,w0,s,x);
%% Input  -f is the object function input as a string ’f’
%%        -tspan is a vector that contains time interval
%%        -tol is a accurcy tolerance
%%        -w0 is a vector that contains initial values
%%        -s is the value of stage
%%        -x is a vector that contains the points computed
%%         in x-axis
%% Output -w is a matrix that contains the solution to test
%%         function
%%        -t is a vector that contains the time values

[mk,mv]=size(w0);
 % t is a vector that contains the value of time
t=zeros(2,1);
t(1)=tspan(1);
% dx is a the value of the stepsize in x-axis
dx=x(2)-x(1);
```

```
w=zeros(mk,2);
w(:,1)=w0;
% dt is a vector that contains two time stepsizes
dt=zeros(2,1);
% r is a vector that contains the coefficient of reaction term
r=(-1)*ones(mk,1)*1e+1;
 % Est is a mk*2 matrix that contains two weighted RMS norm
Est=zeros(2,1);
% estn is a vector contains the errors of each points in x-axis
estn=zeros(mk,1);
es=estn;
% comp is a scale value of computer
comp=1;
pp=zeros(mk,1);
ss=0;
% false is a scale value. The iteration is stopped if false=0.
false=1;
while false==1
    if comp==1
      %%Begin initialing the first time stepsize [1]
      dtt=t_star(f,w(:,1),t(1),tol,r,x,s);
      dt(1)=dtt;
      comp=comp+1;
      t(comp)=t(comp-1)+dtt;
      %%End [1]
  elseif comp==2
          %%Begin computing second time stepsize[2]
          estn=0.8*(w(:,comp-1)-w(:,comp))+0.4*dtt*(feval(f,
       w(:,comp-1),dx,t(comp-1),r,0)+feval(f,w(:,
       comp),dx,t(comp),r,0));
          pp=max(abs(w(:,comp)),abs(w(:,comp-1)));
          ttol=tol+tol*pp;
          omega=sqrt(mk)*ttol;
          Est(1)=norm(estn./omega);
          if Est(1)>1
              fac=0.8/(Est(1)^(1/3));
              dtt=min(10,max(0.1,fac))*dt(1);
              t(comp)=t(comp-1)+dtt;
              dt(1)=dtt;
          else
              fac=0.8/(Est(1)^(1/3));
```

```
                  dtt=min(10,max(0.1,fac))*dt(1);
                  dt(2)=dtt;
                  comp=comp+1;
                  t(comp)=t(comp-1)+dtt;
              end
               %%End [2]
        else
            %% Begin computing other stepsize. [3]
            estn=0.8*(w(:,comp-1)-w(:,comp))+0.4*dtt*(feval(f,
                 w(:,comp-1),dx,t(comp-1),r,0)+feval(f,w(:,
         comp),dx,t(comp),r,0));
            pp=max(abs(w(:,comp)),abs(w(:,comp-1)));
            ttol=tol+tol*pp;
            omega=sqrt(mk)*ttol;
            Est(2)=norm(estn./omega);
            if(Est(2)>1)
                fac=0.8/(Est(2)^(1/3));
                dtt=min(10,max(0.1,fac))*dt(1);
                t(comp)=t(comp-1)+dtt;
                dt(2)=dtt;
            else
                fac=(0.8*Est(1)^(1/3)*dt(2))/(Est(2)^(2/3)*dt(1));
                dtt=min(10,max(0.1,fac))*dt(2);
                dt(1)=dt(2);
                dt(2)=dtt;
                comp=comp+1;
                t(comp)=t(comp-1)+dtt;
                Est(1)=Est(2);
            end
            %%End [3]
        end
        %% Control loop [4]
        if t(comp)>=tspan(2)
          t(comp)=tspan(2);
          dtt=t(comp)-t(comp-1);
          false=2;
          dt(2)=dtt;
      end
      %% End [4]
      %% Begin find stage [5]
     options=optimset('TolX',1e-2);
```

```
radius=abs(spec1(w(:,comp-1),t(comp-1),x,r))*dtt;
ss=ceil(fzero(@myfun,s,options,radius));
if ss<s
    ss=s;
end
k=ss+1;
%% End [5]
%% c,mu,mut,nu,gat are coefficients of RKC method
[c,mu,mut,nu,gat]=coso(ss);
%%Begin computing the solution using implicit-explicit RKC
%%method [6]
W=zeros(mk,k);
W(:,1)=w(:,comp-1);
W(:,2)=W(:,1)+mut(1)*dtt*feval(f,W(:,1),dx,t(comp-1),r,0);
for j=2:ss
  W(:,j+1)=(1-mu(j)-nu(j))*W(:,1)+mu(j)*W(:,j)+nu(j)*W(:,j-1)
          +mut(j)*dtt*feval(f, W(:,j),dx,t(comp-1)+c(j)*dtt,r
  ,0)+gat(j)*dtt*feval(f,W(:,1),dx,t(comp-1),r,0);
 end
 w(:,comp)=W(:,k);
 %%End [6]
 if false==2
    estn=0.8*(w(:,comp-1)-w(:,comp))+0.4*dtt*(feval(f,
         w(:,comp-1),dx,t(comp-1),r,0) +feval(f,w(:,
 comp),dx,t(comp),r,0));
    ttol=tol+tol*abs(w(:,comp));
    omega=sqrt(mk)*ttol;
    Est(2)=norm(estn./omega);
    if Est(2)>1
        false=1;
    else
        false=0;
    end
end
end
```

%
%
% Compute an initial step size
%
%

```
%% This is a function that initials the first time stepsize
function dts=t_star1(f,u0,t0,tol,r,x,s);
%%Input   -u0 is a vector that contains initial values
%%        -t0 is the begining time
%%        -tol is a accurcy tolerance
%%        -r is a vector of coefficien
%%        -dx is a stepsize in x-axis
%%Out     -dts is the first time stepsize
[mk,mv]=size(u0);
dx=x(2)-x(1);
%% sigma is a scale value that contains the spectral
%% radius of the Jacobian
sigma=abs(spec1(u0,t0,x,r));
%% step0 is a scale value that contains tentative stepsize
step0=(s^2-1)*(0.34+0.189*(2/(s-1))^(1.3))/sigma;
%% f0 is a vector that contains the value of the first derivative
f0=feval(f,u0,dx,t0,r,0);
est=step0*(feval(f,u0+step0*f0,t0+step0,dx,r,0)-f0);
%% ttol is a vector that contains the accuracy tolerances
ttol=tol+tol*abs(u0);
omega=sqrt(mk)*ttol;
Est=norm(est./omega);
dts=0.1*step0/sqrt(Est);
```

```
  %
  %
  % Compute spectral radius
  %
  %
```

```
function lampda1=spec1(uu,tt,xx,rr);
%%Input   -uu is a vector that contains solutions to test function
%%        -tt is a time scale
%%        -xx is a vector that contains the points in x-axis
%%        -rr is a vector of coefficien
%%Output  -lampda1 is a spectral radius
[mk,mv]=size(uu);
dx=xx(2)-xx(1);
j1=ones(mk-1,1)/dx^2;
j2=rr-2*ones(mk,1)/dx^2;
j3=ones(mk-1,1)/dx^2;
```

```
A=diag(j1,1)+diag(j2)+diag(j3,-1);
ctmax = 10;
z=ones(mk,1); %Guess
w=A*z;
alpha=max(abs(w));
z=w/alpha;
w=A*z;
[z_km1,z_i]=max(z);
if z_km1==0
    num=1;
   for i=1:mk
       if z(i)~=0
           if num==1
               z_km1=z(i);
               z_i=i;
               num=num+1;
           else
               if z_km1<z(i)
                   z_km1=z(i);
                   z_i=i;
               end
           end
       end
   end
end
w_k=w(z_i,1);
lampda1=w_k/z_km1;
ct = 0;
while 1
    alpha=max(abs(w));
    z=w/alpha;
    w=A*z;
    [z_km1,z_i]=max(z);
if z_km1==0
    num=1;
   for i=1:mk
       if z(i)~=0
           if num==1
               z_km1=z(i);
               z_i=i;
               num=num+1;
           else
               if z_km1<z(i)
                   z_km1=z(i);
                   z_i=i;
               end
           end
       end
   end
end
w_k=w(z_i,1);
lampda2=w_k/z_km1;
lampda=abs(lampda2-lampda1);
if lampda<1e-1 | ct >= ctmax
   lampda1=lampda2;
   break
end
lampda1=lampda2;
ct = ct+1;
end
```

%
%
%Compute coefficient
%
%

```
function [c,mu,mut,nu,gat] = coso(s)
%%Input  -s is a value of stage
%%Output they are vectors respectively which contain
%%       the coefficients
    mu = zeros(s,1);
    mut = mu;
    nu = mu;
    gat = mu;
    c = mu;
    om0 = 1+10/(s*s);
    t1 = om0*om0-1;
    t2 = sqrt(t1);
    a = s*log(om0+t2);
    om1 = sinh(a)*t1/(cosh(a)*s*t2-om0*sinh(a));
    zjm1 = om0;
    zjm2 = 1;
    dzjm1 = 1;
    dzjm2 = 0;
    d2zjm1 = 0;
    d2zjm2 = 0;
    bjm1 = 1/om0;
    bjm2 = (2*om0*d2zjm1-d2zjm2+4*dzjm1)/(2*om0
          *dzjm1-dzjm2+2*zjm1)^2;
    mut(1) = om1*bjm1;
    for i = 2:s
        zj = 2*om0*zjm1-zjm2;
        dzj = 2*om0*dzjm1-dzjm2+2*zjm1;
        d2zj = 2*om0*d2zjm1-d2zjm2+4*dzjm1;
        bj = d2zj/(dzj*dzj);
        ajm1 = 1-zjm1*bjm1;
        mu(i) = 2*om0*bj/bjm1;
        nu(i) = -bj/bjm2;
        mut(i) = mu(i)*om1/om0;
        gat(i) = -ajm1*mut(i);
        c(i) = d2zj/dzj;
        bjm2 = bjm1;
        bjm1 = bj;
        zjm2 = zjm1;
        zjm1 = zj;
        dzjm2 = dzjm1;
        dzjm1 = dzj;
        d2zjm2 = d2zjm1;
        d2zjm1 = d2zj;
    end
    c = (dzj/d2zj)*c;
```

```
%
%
% Discretize in space
%
%
```

```
% m is the number of segments in x-axis
m=100;
xs=[0 1];
```

```
mk=m+1;
% x is a vector that contains the value of points in x axis
x=linspace(xs(1),xs(2),mk);
```

%
%
% Give the initial value
%
%

```
% w0 is a vector that contains the initial value of w
w0=sin(x);
% w0(1)=0 and w0(mk)=0 are boundary conditions
w0(mk)=0;
```

%
%
% Compute the stage
%
%

```
function f = myfun(x,ra);
%% Input  -x is a initial value
%%        -ra is the length of stability domain
%% Output -f is a value of stage
if x<2
    f=-2-ra;
elseif x==2
    f=2-ra;
elseif x<=3
    f=2.232*(x-2)+2-ra;
else
    f=(x^2-1)*(0.340+0.189*(2/(x-1))^(1.3))-ra;
end
```

%
%
% Compute the derivative of the solution w_j
%
%

```
function dw=F(wj,dx,t,r,ie);
%% Input    -wj is a vector that contains solution of
%%           test function at t
%%          -r is a vector of coefficient
%%          -dx is a stepsize in x-axis
%%          -ie is a scale value
%% Output   -dw is a vector that contains the derivative of wj
%% If ie==0 then dw=F(w,t)
%% If ie==1 then dw=F_E(w,t)
%% If otherwise then dw=F_I(w,t)
[mk mv]=size(wj);
dw=zeros(mk,1);
if ie==0
for i=2:mk-1
    dw(i)=wj(i+1)/dx^2+(r(i)-2/dx^2)*wj(i)+wj(i-1)/dx^2;
end
elseif ie==1
    for i=2:mk-1
       dw(i)=wj(i+1)/dx^2-2*wj(i)/dx^2+wj(i-1)/dx^2;
    end
else
    dw=r.*wj;
end
```

%
% End

Appendix B

Source of RKC2

%

% The functions RKC2 integrates initial value problem for
% system of first order ordinary differential equations.
% It is based on a family of implicit-explicit Runge-Kutta-Chebyshev
% formulas.
% An estimate of the spectral radius is used at step to select the
% smallest stage s resulting in a stable integration. RKC2 is approximate
% for the solution to advection-diffusion-reaction problems with highly
% stiff reaction terms.
% RKC2 is different with RKC1 in formula, computing spectral radius
% and estimation of the local error.
%
%

```
function [w,t]=rkc2(f,tspan,tol,w0,s,x);
%% Input  -f is the object function input as a string ’f’
%%        -tspan is a vector that contains time interval
%%        -tol is a accurcy tolerance
%%        -w0 is a vector that contains initial values
%%        -s is the value of stage
%%        -x is a vector that contains the points computed
%%         in x-axis
%% Output -w is a matrix that contains the solution to
%%         test function
%%        -t is a vector that contains the time values

[mk,mv]=size(w0);
 % t is a vector that contains the value of time
t=zeros(2,1);
```

```
t(1)=tspan(1);
% dx is a the value of the stepsize in x-axis
dx=x(2)-x(1);
w=zeros(mk,2);
w(:,1)=w0;
% dt is a vector that contains two time stepsizes
dt=zeros(2,1);
% r is a vector that contains the coefficient of reaction term
r=(-1)*ones(mk,1)*1e+1;
 % Est is a mk*2 matrix that contains two weighted RMS norm
Est=zeros(2,1);
% estn is a vector contains the errors of each points in x-axis
estn=zeros(mk,1);
es=estn;
% comp is a scale value of computer
comp=1;
pp=zeros(mk,1);
ss=0;
% false is a scale value. The iteration is stopped if false=0.
false=1;
while false==1
    if comp==1
      %%Begin initialing the first time stepsize [1]
      dtt=t_star(f,w(:,1),t(1),tol,r,x,s);
      dt(1)=dtt;
      comp=comp+1;
      t(comp)=t(comp-1)+dtt;
      %%End [1]
  elseif comp==2
          %%Begin computing second time stepsize[2]
          estn=0.8*(w(:,comp-1)-w(:,comp))+0.4*dtt*(feval(f,
       w(:,comp-1),dx,t(comp-1),r,0)+feval(f,
       w(:,comp),dx,t(comp),r,0));
          es=r.*feval(f,w(:,comp-1),dx,t(comp-1),r,0);
          estn=estn-(3*dtt^2/(ss^2-1))*es;
          pp=max(abs(w(:,comp)),abs(w(:,comp-1)));
          ttol=tol+tol*pp;
          omega=sqrt(mk)*ttol;
          Est(1)=norm(estn./omega);
          if Est(1)>1
              fac=0.8/(Est(1)^(1/3));
```

```
                dtt=min(10,max(0.1,fac))*dt(1);
                t(comp)=t(comp-1)+dtt;
                dt(1)=dtt;
            else
                fac=0.8/(Est(1)^(1/3));
                dtt=min(10,max(0.1,fac))*dt(1);
                dt(2)=dtt;
                comp=comp+1;
                t(comp)=t(comp-1)+dtt;
            end
             %%End [2]
    else
        %% Begin computing other stepsize. [3]
        estn=0.8*(w(:,comp-1)-w(:,comp))+0.4*dtt*(feval(f,
             w(:,comp-1),dx,t(comp-1),r,0)+feval(f,
     w(:,comp),dx,t(comp),r,0));
        es=r.*feval(f,w(:,comp-1),dx,t(comp-1),r,0);
        estn=estn-(3*dtt^2/(ss^2-1))*es;
        pp=max(abs(w(:,comp)),abs(w(:,comp-1)));
        ttol=tol+tol*pp;
        omega=sqrt(mk)*ttol;
        Est(2)=norm(estn./omega);
        if(Est(2)>1)
            fac=0.8/(Est(2)^(1/3));
            dtt=min(10,max(0.1,fac))*dt(1);
            t(comp)=t(comp-1)+dtt;
            dt(2)=dtt;
        else
            fac=(0.8*Est(1)^(1/3)*dt(2))/(Est(2)^(2/3)*dt(1));
            dtt=min(10,max(0.1,fac))*dt(2);
            dt(1)=dt(2);
            dt(2)=dtt;
            comp=comp+1;
            t(comp)=t(comp-1)+dtt;
            Est(1)=Est(2);
        end
        %%End [3]
    end
    %% Control loop [4]
    if t(comp)>=tspan(2)
      t(comp)=tspan(2);
      dtt=t(comp)-t(comp-1);
      false=2;
      dt(2)=dtt;
  end
  %% End [4]
  %% Begin find stage [5]
 options=optimset('TolX',1e-2);
 radius=abs(spec2(w(:,comp-1),t(comp-1),x,r))*dtt;
 ss=ceil(fzero(@myfun,s,options,radius));
 if ss<s
     ss=s;
 end
 k=ss+1;
 %% End [5]
 %% c,mu,mut,nu,gat are coefficients of RKC method
 [c,mu,mut,nu,gat]=coso(ss);
 %%Begin computing the solution using implicit-explicit RKC
 %%method [6]
 W=zeros(mk,k);
 W(:,1)=w(:,comp-1);
 W(:,2)=W(:,1)+mut(1)*dtt*feval(f,W(:,1),dx,t(comp-1),r,1);
 W(:,2)=W(:,2)./(1-r*mut(1)*dtt);
 for j=2:ss
   W(:,j+1)=(1-mu(j)-nu(j))*W(:,1)+mu(j)*W(:,j)+nu(j)*W(:,j-1)+
            mut(j)*dtt*feval(f,W(:,j),dx,t(comp-1)+c(j)*dtt,r,1)
    +gat(j)*dtt*feval(f,W(:,1),dx,t(comp-1),r,1);
   W(:,j+1)=W(:,j+1)+(gat(j)-(1-nu(j)-mu(j))*mut(1))*dtt*feval(f,
            W(:,1),dx,t(comp-1),r,2)-nu(j)*mut(1)*dtt*feval(f,W(:,
    j-1),dx,t(comp-1)+c(j-1)*dtt,r,2);
   W(:,j+1)=W(:,j+1)./(1-r*mut(1)*dtt);
  end
  w(:,comp)=W(:,k);
  %%End [6]
  if false==2
     estn=0.8*(w(:,comp-1)-w(:,comp))+0.4*dtt*(feval(f,w(:,comp-1),
     dx,t(comp-1),r,0)+feval(f,w(:,comp),dx,t(comp),r,0));
     es=r.*feval(f,w(:,comp-1),dx,t(comp-1),r,0);
     estn=estn-(3*dtt^2/(ss^2-1))*es;
     ttol=tol+tol*abs(w(:,comp));
     omega=sqrt(mk)*ttol;
     Est(2)=norm(estn./omega);
     if Est(2)>1
         false=1;
     else
         false=0;
     end
 end
 end
```

%

%

% Compute an initial step size

%

%

```
%% This is a function that initials the first time stepsize
function dts=t_star2(f,u0,t0,tol,r,x,s);
%%Input   -u0 is a vector that contains initial values
%%        -t0 is the begining time
%%        -tol is a accurcy tolerance
%%        -r is a vector of coefficien
%%        -dx is a stepsize in x-axis
%%Out     -dts is the first time stepsize
[mk,mv]=size(u0);
dx=x(2)-x(1);
%% sigma is a scale value that contains the spectral
%% radius of the Jacobian
sigma=abs(spec2(u0,t0,x,r));
%% step0 is a scale value that contains tentative stepsize
step0=(s^2-1)*(0.34+0.189*(2/(s-1))^(1.3))/sigma;
%% f0 is a vector that contains the value of the first derivative
f0=feval(f,u0,dx,t0,r,0);
est=step0*(feval(f,u0+step0*f0,t0+step0,dx,r,0)-f0);
%% ttol is a vector that contains the accuracy tolerances
ttol=tol+tol*abs(u0);
omega=sqrt(mk)*ttol;
Est=norm(est./omega);
dts=0.1*step0/sqrt(Est);
```

%

%

% Compute spectral radius

%

%

```
function lampda1=spec2(uu,tt,xx,rr);
%%Input   -uu is a vector that contains solutions to test function
%%        -tt is a time scale
%%        -xx is a vector that contains the points in x-axis
%%        -rr is a vector of coefficien
%%Output  -lampda1 is a spectral radius
[mk,mv]=size(uu);
dx=xx(2)-xx(1);
j1=ones(mk-1,1)/dx^2;
j2=-2*ones(mk,1)/dx^2;
j3=ones(mk-1,1)/dx^2;
A=diag(j1,1)+diag(j2)+diag(j3,-1);
ctmax = 10;
z=ones(mk,1); %Guess
w=A*z;
alpha=max(abs(w));
z=w/alpha;
w=A*z;
[z_km1,z_i]=max(z);
if z_km1==0
    num=1;
   for i=1:mk
       if z(i)~=0
           if num==1
               z_km1=z(i);
               z_i=i;
               num=num+1;
           else
               if z_km1<z(i)
                   z_km1=z(i);
                   z_i=i;
               end
           end
       end
   end
end
w_k=w(z_i,1);
lampda1=w_k/z_km1;
ct = 0;
while 1
    alpha=max(abs(w));
    z=w/alpha;
    w=A*z;
    [z_km1,z_i]=max(z);
if z_km1==0
    num=1;
   for i=1:mk
       if z(i)~=0
           if num==1
               z_km1=z(i);
               z_i=i;
               num=num+1;
           else
               if z_km1<z(i)
                   z_km1=z(i);
                   z_i=i;
               end
           end
       end
   end
end
w_k=w(z_i,1);
lampda2=w_k/z_km1;
lampda=abs(lampda2-lampda1);
if lampda<1e-1 | ct >= ctmax
   lampda1=lampda2;
   break
end
lampda1=lampda2;
ct = ct+1;
end
```

%

% End

References

[1] J.G.Verwer, B.P.Sommeijer, W.Hundsdorfer *RKC time-dependent for advection-diffusion-reaction problems*. Modelling, Analysis and Simulation, (2004). pp. 1-23.

[2] B.P.Sommeijer, L.F.Shampine, J.G.Verwer *RKC: an Explicit Solver for Parabolic PDEs*. Modelling, Analysis and Simulation, (1997). 88, pp.315-326.

[3] J.G.Verwer, B.P.Sommeijer, *An implicit-explicit Runge-Kutta-Chebyshev scheme for diffusion-reaction equations*. SIAM J. Sci. Comput, (2003). 46, pp1824-1835.

[4] U.M.Ascher, L.R.Petzold, *Computer Methods for Ordinary Differential Equations and Differetial-Algebraic Equations*. SIAM, (1998).

[5] E.Hairer, S.P.Nosett, G.Wanner *Solving Ordinary Differential Equations I - nonstiff problems*. Springer, (1993).

[6] E.Hairer, G.Wanner *Solving Ordinary Differential Equations II - stiff and differential-algebraic problems*. Springer, (1996).