K A R D O S

– KAskade Reaction DiffusiOn System

Jens Lang

Konrad–Zuse–Zentrum für Informationstechnik Berlin, Heilbronner Strasse 10, D–10711 Berlin–Wilmersdorf

Telefon: (030) 89604202
Telefax: (030) 89604125
e–mail: lang@zib–berlin.de

April 1996

Abstract

A software package for the adaptive solution of time–dependent reaction–diffusion systems in one space dimension is presented. The used algorithm is based on fundamental arguments in LANG [4], LANG/WALTER [5] and the KASKADE toolbox [3].

Key words: embedded Runge–Kutta method,
adaptive finite element method.

1 Introduction

Using the program KARDOS, systems of nonlinear parabolic initial boundary value problems of the form

$$
(1)\qquad
\begin{aligned}
B(x,t,u)u_t - (D(x,t,u)u_x)_x &= F(x,t,u) - V(x,t,u)u_x, \\
& \ x \in \Omega \subset R^1, t \in [t_a, t_e], \\
u &= \xi_1(x,t),\ x \in \Gamma_1 \subset \partial\Omega \\
n^T D(x,t,u)u_x &= \xi_2(x,t,u),\ x \in \Gamma_2 \subset \partial\Omega \\
u(t_a, x) &= u_0(x)\ ,
\end{aligned}
$$

are solvable. Here, $u = (u_1, u_2, ..., u_r)$ is a vector function. The matrix function $B(x,t,u)$ may vanish on a subset of Ω. The time discretization is done by a special embedded Runge–Kutta method of order 3(2) [6]. This method keeps its accuracy even in the case of differential–algebraic equations ($B(x,t,u) = 0$ somewhere). The step size control mechanism employs a third order error model with a time–dependent constant. The space discretization is computed by an adaptive finite element method [2]. KARDOS uses standard linear elements connected with a local error estimator of BABUSKA–RHEINBOLDT type [1] .
The norm which KARDOS uses is a weighted L_2–norm, given by

$$
(2)\qquad \|u_1 - \hat{u}_1\|_{0,w} := \left(\frac{1}{r} \sum_{i=1}^{r} \frac{\|u_{1,i} - \hat{u}_{1,i}\|_0^2}{w_i^2} \right)^{\frac{1}{2}}
$$

where

$$
w_i := \begin{cases} uAbs_i & : \ \|u_{1,i}\|_0 < uAbs_i \\ uRelMax_i & : \ uAbs_i \le \|u_{1,i}\|_0 < uRelMax_i \\ \|u_{1,i}\|_0 & : \ uRelMax_i \le \|u_{1,i}\|_0 \end{cases}
$$

with

$$
\begin{aligned}
uRelMax_i &:= RTOL_i * max_t \|u_{1,i}\|_0\ , \\
uAbs_i &:= ATOL_i * |\Omega|^{1/2}\ .
\end{aligned}
$$

Scaling can be turned on or off. If the scaled error norm is used, the tolerances $RTOL_i$ and $ATOL_i$ have to be selected very carefully to reflect accurately the scale of the problem. The tolerance $ATOL_i$ should indicate the absolute value at which the i–th component is essentially insignificant. On the other hand, the value $RTOL_i$ affects the relative accuracy of the i–th component with respect to its maximal value in time. This control turns out to be quite efficient and robust for a wide class of problems. However, it is clear that it is not a universal method.
In the case of inconsistent starting values for differential–algebraic systems it is possible to use an implicit Euler method in the first time step coupled with a damped Newton method.

2 Defining a problem

At first the user has to set the desired number of equations `noOfEquations` in the main file *kardos.c*. This is necessary to prepare the memory in the functions

```
InitProblem("kardos",100,noOfEquations,250,FULL),
InitTimeProblem("kardos",100,noOfEquations).
```

At the moment the maximal number is 20. It can be changed easily in *nodes.h* setting the fixed parameter MAX_NODE_GROUPS.
The actual problem is defined by the functions

`Parabolic`	:	$B(x,t,u)$
`typeOpB`	:	type of the operator $B(x,t,u)$
`Laplace`	:	diffusion matrix $D(x,t,u)$
`Convection`	:	transport direction $V(x,t,u)$
`Helmholtz`	:	put in the right–hand side (IGNORE)
`typeOpA`	:	type of the operator $D(x,t,u)+V(x,t,u)$
`Source`	:	force term $F(u)$
`Jacobian`	:	jacobian of the force term $F_u(u)$
`InitialFunc`	:	initial values $u_0(x)$
`Cauchy`	:	boundary values $\xi_2(x,t,u)$
`Dirichlet`	:	boundary values $\xi_1(x,t)$
`Solution`	:	solution if known.

The procedure `SetTimeProblem` announces the problem defined that way to the KARDOS toolbox. According to the fact, that one example is better than thousand explanations, the reader is referred to study intensively the file *standard.c*. Note that the analytic declaration of the Jacobian matrix is not necessary since the program also computes a numerical approximation.

3 Reading a grid

KARDOS requires two common starting grids, which first have to be read with the help of the command *readtri*. During the computation one grid concerns the starting values, on the other grid the current solution is computed adaptively.
Beside a geometrical description of the coarse space discretization, the sort of boundary conditions has to be declared. For that, the parameters *B, I, D* and *C*, which stand for Boundary, Interior, Dirichlet and Cauchy, are available. Furthermore, it's possible to characterize different subdomains through a class–parameter. A general input reads as follows:

> grid name
> Dimension:(number of points,number of edges)
> point index:x–coordinate,B or I to characterize
> the location of the point, I or D or C to characterize
> the boundary conditions for all components, class–parameter
> to characterize a special property only on the boundary
> END
> edge index:(left point index,right point index), class–parameter
> to characterize a special property of a subdomain
> END

An example concerning two components with mixed boundary conditions on [0,1] and two different materials in [0,0.5) and (0.5,1] looks like the following input stream:

```
example1
Dimension:(4,3)
0:0.0,B,DC,0
1:0.25,I,II
2:0.5,I,II
3:1.0,B,CD,1
END
0:(0,1),0
1:(1,2),0
2:(2,3),1
END
```

Inner boundary points are available.

4 Command language

Here all special commands which KARDOS needs for the time integration are explained in the following. A list of all commands can be found in *kardos.def*. All other commands not explained here are used as in KASKADE. KARDOS reads at the start a file *kardos.startup* which contains a sequence of commands and executes these commands. If no *quit* is included, more commands are requested from standard input.

`seltimeinteg`

A time integrator is selected. The parameters *rodas* and *euler* are available. The Euler method can be used only for the starting step.

`timestepping`

This command corresponds to the solve–command of KASKADE. The time integration is carried out. The following parameters can be used:

`maxsteps`	Int	-	number of maximal time steps
`tstart`	Real	-	starting time
`tend`	Real	-	final time
`timestep`	Real	-	initial time step
`globtol`	Real	-	accuracy demanded for the integrator (internal spacetol and timetol are set automatically)
`spacetol`	Real	-	desired space accuracy
`timetol`	Real	-	desired time accuracy (if spacetol and timetol are selected, then internal globtol is set)
`showsol`	Int	-	graphic is plotted every Int steps, standard is 1

`timeproblem`
A time problem is selected.

`inftimeproblem`
informs about the current time problem.

`seldirect`
For the time integration only the parameter *ma28* is available.

`directparam`
The sparse direct solver needs sometimes additional memory. This is done with the help of the parameters *licnfac* and *lirnfac*. The underlying fill-in procedure can be controlled by setting *dropfac*.

`selrefine`
For different refinements the parameters *meanval* or *extrapol* are available.

`setscaling`
The tolerance parameters *atol* should indicate the absolute value at which the corresponding component is essentially insignificant. The tolerance parameters *rtol* affect the relative accuracy with respect to the maximal value in time.

`setplot`
After each time step the solutions and the corresponding grid can be plottet into the directory ../plot. The files are written in MATLAB format. Several

routines such as plotgrid.m or plotsol.m are available to plot the data within MATLAB.

5 An example

In this chapter we show how to add a new problem to the list of predefined problems:

- define a set of functions to characterize the initial boundary value problem,
- add the problem to the list of predefined problems (call `SetProblem`, using `DefUserTimeProblems` in *user.c* of the kardos–directory,
- add two common coarse grids in the grids–directory,
- use `kardos.startup` for an automatic command language dialog with the system.

5.1 User functions for a population ecology model

Let us solve the following equations

$$\begin{aligned} u_t \; - 0.0125 \; u_{xx} &= \left(\tfrac{35+16u-u^2}{9} - v\right) \cdot u \\ v_t \; - \; v_{xx} &= \left(u - \tfrac{5+2v}{5}\right) \cdot v \end{aligned}$$

in the domain $\Omega = (0, 2.5)$ for $t > 0$, with the boundary conditions

$$u_x = v_x = 0 \quad \text{for} \quad t > 0\ ,\ x = 0 \text{ and } x = 2.5$$

and the initial condition

$$u_0 = \begin{cases} 5 & \text{for} \quad 0 \leq x < 1.0 \\ 4x+1 & \text{for} \quad 1.0 \leq x < 1.25 \\ -4x+11 & \text{for} \quad 1.25 \leq x < 1.5 \\ 5 & \text{for} \quad 1.5 \leq x < 2.5 \end{cases}$$

$$v_0 = \begin{cases} 10 & \text{for} \quad 0 \leq x < 1.0 \\ 4x+6 & \text{for} \quad 1.0 \leq x < 1.25 \\ -4x+16 & \text{for} \quad 1.25 \leq x < 1.5 \\ 10 & \text{for} \quad 1.5 \leq x < 2.5. \end{cases}$$

This system of nonlinear equations models a certain planktonit predator–prey situations in which crowding is a factor. The KARDOS user defines in *user.c* the following functions:

```
static int EcoParabolic(real x, int classA, real t,
        real *u, int equ, int var, real *fVal)
  {
        if (equ!=var) return F_IGNORE;

        switch (var)
        {
          case 0: fVal[0] = 1.0;
                  break;
          case 1: fVal[0] = 1.0;
                  break;
        }
        return F_CONSTANT;
  }

static int EcoLaplace(real x, int classA, real t,
        real *u, int equ, int var, real *fVal)
  {
        if (equ!=var) return F_IGNORE;

        switch (var)
        {
          case 0: fVal[0] = 0.0125;
                  break;
```

```
          case 1: fVal[0] = 1.0;
          break;
        }
        return F_CONSTANT;
  }

static int EcoSource(real x, int classA, real t,
        real *u, int equ, real *fVal)
  {
        switch (equ)
        {
         case 0: fVal[0] = ((35.0+16.0*u[0]-u[0]*u[0])/
                             9.0-u[1])*u[0];
                 break;
         case 1: fVal[0] = (u[0]-1.0-0.4*u[1])*u[1];
                 break;
        }
        return F_VARIABLE;
  }

static int EcoJacobian(real x, int classA, real t,
        real *u, int equ, int var, real *fVal)
  {
        switch (equ)
        {
         case 0: if (var==0)
                  fVal[0] = (35.0+32.0*u[0]-3.0*u[0]*u[0])/
                             9.0-u[1];
                 else if (var==1)
                  fVal[0] = -u[1];
                 break;
         case 1: if (var==0)
                  fVal[0] = u[1];
                 else if (var==1)
                  fVal[0] = u[0]-1.0-0.8*u[1];
                 break;
        }
        return F_VARIABLE;
  }
```

```
static real EcoInitialFunc(real x, int classA, int var)
 {
   real val = 0.0;

   switch (var)
   {
     case 0: if ((0.0<=x)&&(x<1.0)) val = 5.0;
             else if ((1.0<=x)&&(x<1.25)) val = 4.0*x+1.0;
             else if ((1.25<=x)&&(x<1.5)) val = -4.0*x+11.0;
             else if ((1.5<=x)&&(x<=2.5)) val = 5.0;
             break;
     case 1: if ((0.0<=x)&&(x<1.0)) val = 10.0;
             else if ((1.0<=x)&&(x<1.25)) val = 4.0*x+6.0;
             else if ((1.25<=x)&&(x<1.5)) val = -4.0*x+16.0;
             else if ((1.5<=x)&&(x<=2.5)) val = 10.0;
             break;
    }
   return val;
  }

static int EcoDirichlet(real x , int classA, real t, int var,
        real *fVal)
  {
        return F_IGNORE ;
  }

static int EcoCauchy(real x, int classA, real t, real *u,
        int equ, real *fVal)
  {
        switch (equ)
        {
          case 0: if ((x==0.0)||(x==2.5)) fVal[0]  = 0.0;
                  else ZIBStdOut("error in EcoCauchy\n");
                  break;
          case 1: if ((x==0.0)||(x==2.5)) fVal[0]  = 0.0;
                  else ZIBStdOut("error in EcoCauchy\n");
                  break;
        }
        return F_CONSTANT;
  }
```

5.2 Defining the new problem

Next we define the new problem with the help of the following procedure:

```
int DefUserTimeProblems()
  {
    if (!SetTimeProblem("test-ecology",varName,
            EcoParabolic,
            OP_LINEAR,
            EcoLaplace,
            (int(*)(real,int,real,real*,int,int,real*))nil,
            (int(*)(real,int,real,real*,int,int,real*))nil,
            OP_LINEAR,
            EcoSource,
            EcoJacobian,
            EcoInitialFunc,
            EcoCauchy,
            EcoDirichlet,
            (int(*)(real,int,real,int,real*))nil)) return false;
  }
```

A call of `DefUserTimeProblems` is carried out automatically from the main program.

5.3 Getting ready two common coarse grids

Now we insert two coarse grids in the grids–directory, using the files *ecology1.g* and *ecology2.g*.

```
ecology1             ecology2
Dimension:(11,10)    Dimension:(11,10)
0:0.0,B,CC           0:0.0,B,CC
1:0.25,I,II          1:0.25,I,II
2:0.5,I,II           2:0.5,I,II
3:0.75,I,II          3:0.75,I,II
4:1.0,I,II           4:1.0,I,II
5:1.25,I,II          5:1.25,I,II
6:1.5,I,II           6:1.5,I,II
```

```
7:1.75,I,II            7:1.75,I,II
8:2.0,I,II             8:2.0,I,II
9:2.25,I,II            9:2.25,I,II
10:2.5,B,CC            10:2.5,B,CC
END                    END
0:(0,1)                0:(0,1)
1:(1,2)                1:(1,2)
2:(2,3)                2:(2,3)
3:(3,4)                3:(3,4)
4:(4,5)                4:(4,5)
5:(5,6)                5:(5,6)
6:(6,7)                6:(6,7)
7:(7,8)                7:(7,8)
8:(8,9)                8:(8,9)
9:(9,10)               9:(9,10)
END                    END
```

5.4 An automatic dialog with the system

KARDOS reads at the start a file *kardos.startup* which contains a sequence of commands and executes these commands. That way we get an automatic dialog with the system. Of course, a step–by–step dialog is available too. We deposit in *kardos.startup* the following commands:

```
read ../grids/ecology1.g
read ../grids/ecology2.g

timeproblem test-ecology

selestimate babuska
selrefine meanval
seldirect ma28

window new automatic name solution1
graphic solution triangulation
seldraw 0
window new automatic name solution2
graphic solution triangulation
seldraw 1
```

```
seltimeinteg rodas
setscaling atol 0.001 0.001 rtol 0.01 0.01
setplot grid 1 nocomponents 2
timestepping timestep 0.0001 maxsteps 200 globTol 0.001 tEnd 10.0
```

Now we can start the program.

References

[1] I. Babuska, W.C. Rheinboldt: *Error estimates for adaptive finite element computations* SIAM J. Numer. Anal. **15**, 736–754 (1978)

[2] P. Deuflhard, P. Leinen, H. Yserentant: *Concepts of an Adaptive Hierarchical Finite Element Code* IMPACT **1**, 3–35 (1989)

[3] B. Erdmann, J. Lang, R. Roitzsch: *KASKADE–Manual* Technical Report TR 93–5, Konrad–Zuse–Zentrum (ZIB), (1993)

[4] J. Lang: *High–Resolution Selfadaptive Computations on Chemical Reaction–Diffusion Problems with Internal Boundaries* Chem. Engrg. Science **71**, No. 7, 1055–1070 (1996)

[5] J. Lang, A. Walter: *An Adaptive Rothe Method for Nonlinear Reaction–Diffusion Systems* Applied Numerical Mathematics **13**, 135–146 (1993)

[6] M. Roche: *Rosenbrock Methods for Differential Algebraic Equations* Numer. Math. **52**, 45–63 (1988)