Jens Lang

KARDOS — KAskade Reaction Diffusion One–dimensional System

Technical Report TR 93–9 (July 1993)

Jens Lang

KARDOS — KAskade Reaction Diffusion One–dimensional System

Abstract. A software package for the adaptive solution of time–dependent reaction–diffusion systems and linear elliptic systems in one space dimension is presented. The used algorithm is based on fundamental arguments in LANG/WALTER [4]. Here, only brief outlines of the algorithm are given. This software package is based on the KASKADE toolbox [3].

Key words: Rothe method, embedded Runge–Kutta method, adaptive Multilevel Finite Element method.

Contents

Chapter 1

Introduction

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

$$\begin{aligned}
P(x)u_t - (D(x)u_x)_x &= F(u) \quad , x \in \Omega \subset R^1, t \in [t_a, t_e] \\
u &= \xi_1(t,x) \quad , x \in \Gamma_1 \subset \partial\Omega \\
u_n + \sigma(x)u &= \xi_2(t,x) \quad , x \in \Gamma_2 \subset \partial\Omega \\
u(t_a, x) &= u_0(x) \, ,
\end{aligned} \tag{1.1}$$

are solvable. Here, $u = (u_1, u_2, ..., u_r)$ is a vector function. The matrix function $P(x)$ may vanish on a subset of Ω. The discretization is done by the Rothe method. In contrast to the widespread method of lines, time is discretized first than space. The main advantage of this sequence is the possibility to compute the space discretization optimal during the time integration by an adaptive multilevel finite element method. Therefore the KASKADE toolbox is modified to handle one–dimensional problems. In KARDOS a special embedded Runge–Kutta method of order 3(2) has been implemented. This method keeps its accuracy even in the case of differential–algebraic equations ($P(x) = 0$ somewhere).
The above differential equations can be reformulated into an abstract Cauchy problem possibly of differential–algebraic type in an appropriate Hilbert space H.

$$\begin{aligned}
Pu_t &= f(u) \quad , u \in H \ , t \in [t_a, t_e] \\
u(0) &= u_0 \, .
\end{aligned} \tag{1.2}$$

Now, a three–stage embedded Rosenbrock method for this pure time problem looks as follows:

$$(P - \gamma_{ii}\tau f_u(u_0))k_i = \tau f(u_0 + \sum_{j=1}^{i-1} \alpha_{ij}k_j) + \tau f_u(u_0) \sum_{j=1}^{i-1} \gamma_{ij}k_j \quad , i = 1, 2, 3$$

$$\begin{aligned}
u_1 &= u_0 + \sum_{j=1}^{3} b_j k_j \\
\hat{u}_1 &= u_0 + \sum_{j=1}^{3} \hat{b}_j k_j
\end{aligned} \tag{1.3}$$

Yet this form is not suited to be implemented straight–forward, because along with function evaluations there are still matrix∗vector products on the right–hand side. The transformation

$$l_i := \sum_{j=1}^{i} \gamma_{ij} k_j \quad , i = 1, 2, 3$$

leads to the new system

$$\begin{aligned} \left(\frac{1}{\tau\gamma_{ii}} P - f_u(u_0) \right) l_i &= f(u_0 + \sum_{j=1}^{i-1} a_{ij} l_j) + P \sum_{j=1}^{i-1} \frac{c_{ij}}{\tau} l_j \quad , i = 1, 2, 3 \\ u_1 &= u_0 + \sum_{j=1}^{3} m_j l_j \\ \hat{u}_1 &= u_0 + \sum_{j=1}^{3} \hat{m}_j l_j \ . \end{aligned} \tag{1.4}$$

In KARDOS a special set of parameters, which guarantees L–stability of the chosen method, is used. Setting $\gamma_{11} = \gamma_{22} = \gamma_{33} := \gamma$ only two function evaluations and one matrix inversion are needed on the right–hand side. The corresponding method was given by ROCHE [6], who gives further statements about convergence and consistency.

coefficients					
γ	=	0.435866521508459			
m_1	=	2.236727045296589	$\hat{m}_1$	=	2.059356167645941
m_2	=	2.250067730969645	$\hat{m}_2$	=	0.169401431934653
m_3	=	-0.209251404439032	$\hat{m}_3$	=	0.0
a_{21}	=	1.605996252195329	c_{21}	=	0.8874044410657823
a_{31}	=	1.605996252195329	c_{31}	=	23.98747971635035
a_{32}	=	0.0	c_{32}	=	5.263722371562130

Due to the difference of the approximated values of order 3 and 2

$$\text{timeError} := \| u_1 - \hat{u}_1 \| \tag{1.5}$$

we have a good estimator of the main error term describing the local error of the second order method. The proposal for the new step size is

$$\tau_{new} := \text{safeFactor} * \left(\frac{\text{timeTol}}{\text{timeError}} \right)^{\frac{1}{3}} * \tau \ . \tag{1.6}$$

The norm which KARDOS uses is a weighted L_2–norm, given by

$$\|u_1 - \hat{u}_1\|_{0,w} := \left(\frac{1}{r} \sum_{i=1}^{r} \frac{\|u_{1,i} - \hat{u}_{1,i}\|_0^2}{w_i^2} \right)^{\frac{1}{2}} \tag{1.7}$$

where

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

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.

To implement one time step, one elliptic problem has to be solved at each stage of the Rosenbrock method. This is done by means of an adaptive multilevel finite element method elaborated in recent years, see DEUFLHARD et al. [2]. This method is an excellent tool to adapt the space discretization for the current solution in such a way, that a prescribed tolerance *spaceTol* is achieved. KARDOS uses standard linear elements connected with a local error estimator of BABUŠKA–RHEINBOLDT [1] type. We get at each stage, discretizations of the form

$$B_\tau(l_i^n, v_n) = r_i(v_n), l_i^n \in S_n, \forall v_n \in S_n^0, i = 1, 2, 3, \tag{1.8}$$

where S_n denotes the space of all continuous, piecewise linear functions on a partition $\triangle_n$ of Ω, satisfying the Dirichlet boundary conditions. In S_n^0 they vanish on the Dirichlet boundary. The equations only differ in the respective right–hand side and in the special boundary conditions taken into consideration.

To control the space discretization, we restrict ourselves to the first stage of the Rosenbrock method which is exactly the semi–implicit Euler method applied to the system (1.2). According to a local principle, on each current grid an error estimation is computed. Therefore we consider the corresponding elliptic differential equation on each element, imposing the current solution of the Euler method u_E as boundary conditions, and solve it approximately by a quadratic finite element method. The difference ϵ_k between the linear and quadratic approximation over the k–th element satisfies

$$B_\tau(\epsilon_k,\phi) = r_1(\phi) - B_\tau(u_E,\phi), \epsilon_k, \phi \in Q_k^0, \tag{1.9}$$

where Q_k^0 only consists of the quadratic bubble function. Finally, we end up with an error estimator for the k–th element

$$\delta_k := \|\epsilon_k\|_{0,w} \quad .$$

The error estimator computed that way makes sense whenever the quadratic approximation is better than the linear one. In most cases this assumption is fulfilled.

Equipped with such an error estimator, the solution of the semi–implicit Euler method, and hence the solution u_1, can be improved using an adaptive refinement process to equilibriate the global discretization error over the whole grid. Usually, an element should be refined if its local error estimator exceeds a certain level *cut*. In KARDOS two different strategies for computing *cut* are realized. One of them is simply to compute the average of all local estimators

$$\mathrm{cut}_1 = \frac{1}{n}\sum_{k=1}^{n}\delta_k \quad .$$

This approach uses only the current space discretization. Another possibility consists in bringing in the multilevel structure. Beside δ_k the error of the direct father–element $\delta_{old,k}$ is taken into account. Then, a predictor

$$\hat{\delta}_k := \min\left\{\frac{\delta_k^2}{\delta_{old,k}}\ ,\ \delta_k\right\}$$

shows the effect if the k–th element is divided. The refinement level is set to

$$\mathrm{cut}_2 = \mathrm{max}_k \hat{\delta}_k \ .$$

This method is often named extrapolation method. The adaptive refinement process stops, whenever the following condition holds

$$\left(\sum_{k=1}^{n} \delta_k^2\right)^{1/2} < \text{ spaceTol} .$$

The final grid is used for solving the other elliptic problems too. Only one grid is applied for all unknowns of the system.
Setting the parameter *globTol*, KARDOS controls the space and time discretization due to the special tolerances

$$\text{spaceTol} = \text{globTol}/3.0 \ , \quad \text{timeTol} = \text{globTol}/2.0 \ .$$

This specific splitting results from the chosen parameters of the method. On the other hand, the parameters *spaceTol* and *timeTol* allow special tuning, if desired. For more details see LANG/WALTER [4,5].

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.

As a program, which handles a very general class of problems, KARDOS cannot succeed for all problems. Experiments have shown, that the computations have often been improved setting the parameters very carefully. In extreme cases the iterative solver may fail.

Chapter 2

Defining a problem

At first the user has to check that in the main file *kardos.c* the static parameter *time* is set to one. Next the desired number of equations `noEqn` used in the functions

```
InitProblem("kardos",100,noEqn,250,FULL)
InitTimeProblem("kardos",100,noEqn)
```

to prepare the memory has to be inserted. At the moment the maximal number is 10. It can be changed easily in *nodes.h* setting the fixed parameter MAX_NODE_GROUPS.
The actual problem is defined by the functions

`Parabolic`	:	$P(x)$
`Laplace`	:	diffusion matrix $D(x)$
`Convection`	:	not carried out! (IGNORE)
`Helmholtz`	:	put in the right–hand side (IGNORE)
`Source`	:	force term $F(u)$
`Jacobian`	:	jacobian of the force term $F_u(u)$
`InitialFunc`	:	initial values $u_0(x)$
`Cauchy`	:	boundary values $\xi_2(t,x), \sigma(x)$
`Dirichlet`	:	boundary values $\xi_1(t,x)$.

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 *stdtimeprob.c*.
Hint: Setting *time=0* KARDOS can also be used to solve systems of linear elliptic partial differential equations in one space dimension comparable to ELLKASK [3].

Chapter 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.

Chapter 4

Command language

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.

`timeproblem`
A time problem is selected.

`inftimeproblem`
informs about the current time problem.

`seliterate`
For the time integration the parameters *ssortime* and *pbicgstabtime* are available.

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

`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:

`verboseP`	0,1	-	Important data are printed. At the moment, always true.
`scaling`	0,1	-	step size control: 0 absolute, 1 for relative. If scaling is set to be 1, so all corresponding parameters atoli, rtoli, i=1,2,...,number of unknowns have to be set too.
`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)
`itertol`	Real	-	desired relative accuracy of the iterative solver for the linear systems, standard is 0.0001
`itermaxsteps`	Int	-	maximal number of iterations, standard is 1000
`showsol`	Int	-	graphic is plotted every Int steps, standard is 1
`atoli`	Real	-	
`rtoli`	Real	-	i=1,...,number of unknowns, necessary input for every unknown to control the time step size relatively (scaling 1). The tolerance atol should indicate the absolute value at which the corresponding component is essentially insignificant. The tolerance rtol affects the relative accuracy with respect to the maximal value in time.

Chapter 5

For programmers only

This chapter gives some information to get a feeling for the internal structure of the program. Of course, it cannot mention all details. Furthermore, for a better understanding it is necessary to read all header– and c–files of KARDOS very carefully.

5.1 Defining a time problem

To include all data of a initial boundary value problem two new data structures are added.

```
struct  timeProblem
        char *name;
        int (*Parabolic)(...);
        int (*Laplace)(...);
        int (*Convection)(...);
        int (*Helmholtz)(...);
        int (*Source)(...);
        int (*Jacobian)(...);
        real (*InitialFunc)(...);
        int (*Cauchy)(...);
        int (*Dirichlet)(...);
        int (*Sol)(...);

struct  timeProblemType
        char *name;
        int maxNoOfTimeProblems;
        int noOfEquations;
```

The access to these structures is realized by the pointers

```
timeProblem      *actTimeProblem, *timeProblems;
timeProblemType  *theTimeProblem;
```

Furthermore, there are some functions to announce certain problems within the program.

```
int SetTimeProblem(char *name, char **varNames,
                   int *Parabolic,
```

```
                    int *Laplace,
                    int *Convection,
                    int *Helmholtz,
                    int *Source,
                    int *Jacobian,
                    real *InitialFunc,
                    int *Cauchy,
                    int *Dirichlet,
                    int *Sol)
```

A time problem is announced.

`int SetStdTimeProblems()`
Some standard time problems are announced.

`void SetTimeProblemAddresses()`
Special addresses are set to select time problems within the command language.

```
int InitTimeProblem(char *name,
                    int noOfTimeProblems,
                    int noOfEquations)
```

The current timeProblemType–structure is built up.

5.2 Defining a time integrator

To define a time integrator the structure `timeIntegMethod` is used. It consists of the following elements:

`char *name`
name of the time integrator

`int verboseP`
control of the information output, standard: true

`int scaling`
control of the norm for the time error, 0: absolute, 1: relative

`int showSol`
control of the picture plot, standard: 1

`int nonNegativity`
control of the non–negativity of the solution, unused

`int noOfTimeSteps`
number of the current time steps

`int maxSteps`
maximal number of time steps

`int noOfStepReductions`
number of the time step reductions within an integration step

`int maxStepReductions`
maximal allowed time step reductions within an integration step,
standard: 10

`int stepTypePar`
differentiation from the starting step
START_STEP: starting step, LATER_STEP: later step

`int backDepth`
grid depth, on which every solution process is started again, standard: 0

`real time`
local starting time

`real newTime`
local final time

`real tEnd`
global final time

`real timeStep`
length of the time step

`real newTimeStep`
new proposed time step

`real tRest`
remained global time

`real gammaStab`
stability parameter of the method

`real timeTol`
allowed tolerance of the local time error

`real spaceTol`
allowed tolerance of the local space error

`real globTol`
allowed tolerance of the global error

`real timeError`
current local time error

`real globError`
current global error

`real trueError`
true error if solution known, unused

`real normSol`
norm of the solution

`real timeStepFactor`
factor by which the old time step is divided

`real rho`
timeTol/globTol, standard: 0.5

`real maxTimeStep`
maximal time step, standard: 10.0

`real timeStepSafe`
safety factor for the control of the time step, standard: 0.9

`real spaceTolFac`
spaceTol/globTol, standard: 1.0/3.0

`TRIANGULATION *compTriang`
current computation grid

`TRIANGULATION *resultTriang`
grid for the starting values

`int *InitTimeInteg`
initializes the time integration process

`int *TimeInteg`
realizes the time integration process

`int *CloseTimeInteg`
closes the time integration process

`int *AssRhs1`
computes the right–hand side of the first stage

`int *AssRhs2`
computes the right–hand side of the second stage

`int *AssRhs3`
computes the right–hand side of the third stage

`problem *firstStage`
elliptic problem of the first stage

`problem *secondStage`
elliptic problem of the second stage

`problem *thirdStage`
elliptic problem of the third stage

The access to the different time integrators is done by the pointers

```
timeIntegMethod  *actTimeIntegtor, *timeIntegMethods;
```

To build up a time integrator the following functions are available:

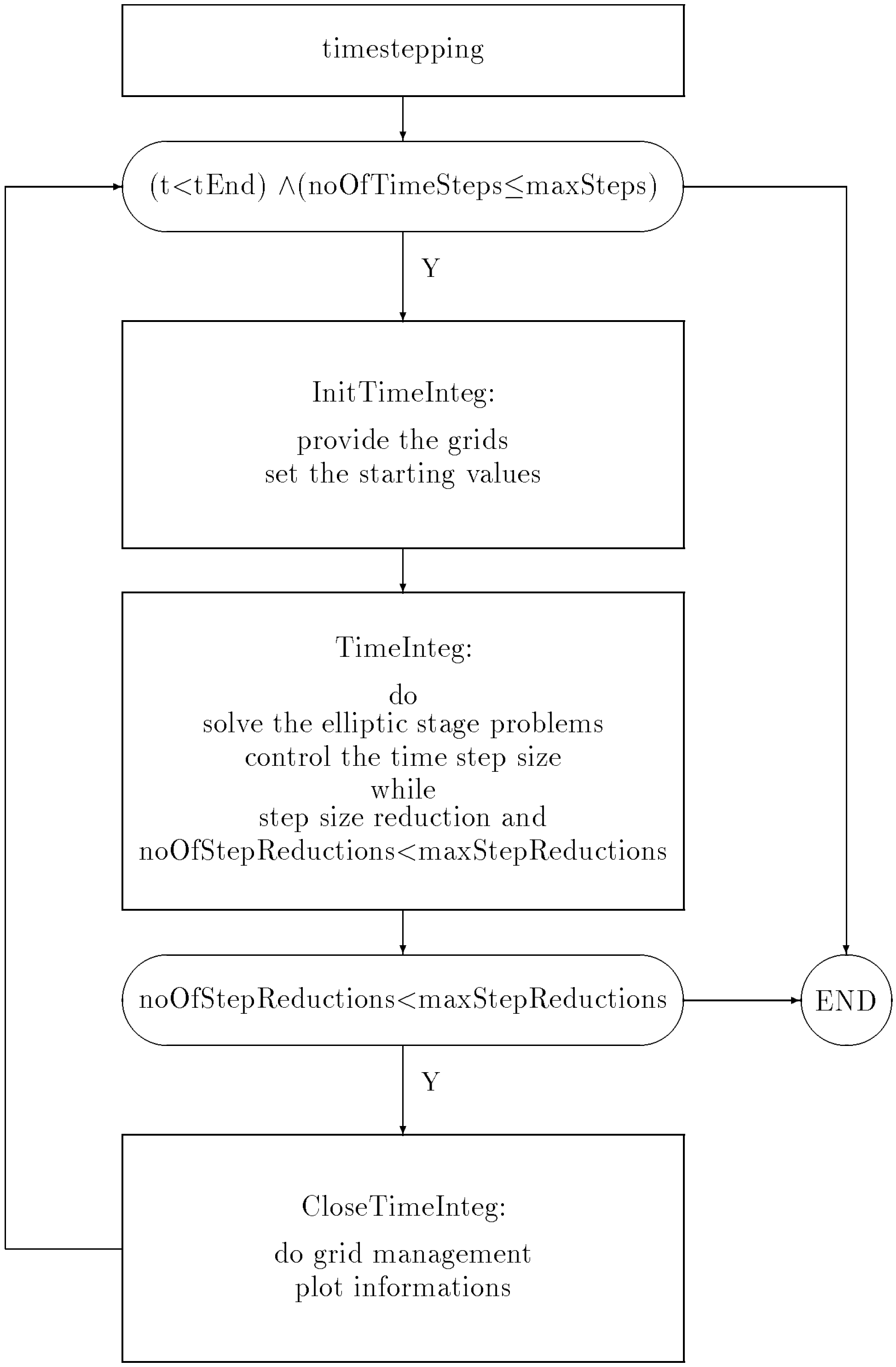
```
int DefTimeIntegMethod(char *name,
                       int *InitTimeInteg,
                       int *TimeInteg,
                       int *CloseTimeInteg,
                       char *nameFirstStage,
                       char *nameSecondStage,
                       char *nameThirdStage,
                       int *AssRhs1,
                       int *AssRhs2,
                       int *AssRhs3,
                       real spaceTolFactor,
                       real gammaStability)
```

announces a time integrator. Beside a name, functions for the realization of the method (`InitTimeInteg`, `TimeInteg`, `CloseTimeInteg`), the corresponding elliptic problems (`"nameFirstStage"`, `"nameSecondStage"`, `"nameThirdStage"`), the functions for the computation of the right–hand sides (`AssRhs1`, `AssRhs2`, `AssRhs3`), the parameter `spaceTolFac` and the stability parameter `gammaStab` have to be inserted,
such as `DefTimeIntegMethod("rodas",InitIntegStep,RodasStep,Close-IntegStep,"rodas1", "rodas2","rodas3",AssRHSRodas1,AssRHS-Rodas2,AssRHSRodas3,THIRD,RODAS_GAMMA)`.

`int SetStageProblems()`
The elliptic problems occurring in each stage of the method are announced within `SetStageProblems` by calling the function `SetProblem`.

5.3 Time integration process

Picture 1 shows the whole process of the time integration method. The program comes to an end when the final time or the maximal step number are reached or when the maximal number of step size reductions within one integration step is exceeded.



Picture 1: Time integration process

5.4 Grid management

The program KASKADE supports work with several grids which are connected to a list. A time integration requires to exchange values on differently refined grids. This leads to costly seek algorithms. To be more effective here, the structure EDG is enlarged by the new element

```
EDG *twin;
```

If the considered edge exists in the other triangulation, then this element points at the corresponding element, otherwise it is nil. In the structure TRIANGULATION the new element `int twinRelation` stands for the existence of such a relation between two triangulations.
To work with two grids the following functions can be used:

`int SetTwinRelOnCoarseGrids(TRIANGULATION *t,TRIANGULATION *t)`
initializes the twin–relation of two coarse grids.

`EDG *FindBrotherEdge(real x,EDG *edFrom)`
finds the edge of the other grid in which a given point lies.

`real InterpolateInEdge(real x,EDG *edIn,int var,int index)`
gives back an interpolated value.

Chapter 6

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 `InitUserTime` 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.

6.1 User functions for a population ecology model

Let us solve the following equations

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

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

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

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

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 *fVals,
                        int equation, int variable)
  {
        if (equation!=variable) return F_IGNORE;
        switch (variable)
        {
          case 0: fVals[0] = 1.0;
                  break;
          case 1: fVals[0] = 1.0;
                  break;
        }
        return F_CONSTANT;
  }

static int EcoLaplace(real x, int classA, real t, real *fVals,
                      int equation, int variable)
  {
        if (equation!=variable) return F_IGNORE;
        switch (variable)
        {
          case 0: fVals[0] = 0.0125;
                  break;
          case 1: fVals[0] = 1.0;
                  break;
        }
        return F_CONSTANT;
  }

static int EcoConvection(real x, int classA, real t, real *fVal,
                         int equation, int variable)
  {
        return F_IGNORE;
  }

static int EcoHelmholtz(real x, int classA, real t, real *fVal,
                        int equation, int variable)
  {
        return F_IGNORE;
  }

static int EcoSource(real x, int classA, real t, EDG* ed,
```

```
                     real *fVal, int equation,
                     real (*Func)(real,EDG*,int,timeIntegMethod*))
  {
    real val0, val1;

        switch (equation)
        {
         case 0: val0 = Func(x,ed,0,actTimeIntegtor);
                 val1 = Func(x,ed,1,actTimeIntegtor);
                 fVal[0] = ((35.0+16.0*val0-val0*val0)/9.0-val1)*val0;
                 break;
         case 1: val0 = Func(x,ed,0,actTimeIntegtor);
                 val1 = Func(x,ed,1,actTimeIntegtor);
                 fVal[0] = (val0-1.0-0.4*val1)*val1;
                 break;
        }
    return F_VARIABLE;
  }

static int EcoJacobian(real x, int classA, real t, EDG* ed, real *fVal,
                       int equation, int variable,
                       real (*Func)(real,EDG*,int,timeIntegMethod*))
  {
    real val0, val1;

        switch (equation)
        {
         case 0: if (variable==0)
                 {
                  val0 = Func(x,ed,0,actTimeIntegtor);
                  val1 = Func(x,ed,1,actTimeIntegtor);
                  fVal[0] = (35.0+32.0*val0-3.0*val0*val0)/9.0-val1;
                 }
                 else if (variable==1)
                 {
                  val1 = Func(x,ed,1,actTimeIntegtor);
                  fVal[0] = -val1;
                 }
                 else ZIBStdOut("error in EcoJacobian\n");
                 break;
         case 1: if (variable==0)
                 {
                  val1 = Func(x,ed,1,actTimeIntegtor);
                  fVal[0] = val1;
                 }
```

```
                else if (variable==1)
                {
                 val0 = Func(x,ed,0,actTimeIntegtor);
                 val1 = Func(x,ed,1,actTimeIntegtor);
                 fVal[0] = val0-1.0-0.8*val1;
                }
                else ZIBStdOut("error in EcoJacobian\n");
                break;
      }
    return F_VARIABLE;
  }

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

   switch (variable)
   {
     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;
             else ZIBStdOut("error in EcoInitialFunc\n");
     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;
             else ZIBStdOut("error in EcoInitialFunc\n");
     break;
    }
   return val;
  }

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

static int EcoCauchy(real x, int classA, real t, real *fVals, int variable)
  {
        switch (variable)
        {
          case 0: if ((x==0.0)||(x==2.5)) {
                      fVals[0]  = 0.0;  /* Sigma */
```

```
                    fVals[1]  = 0.0;  /*     Xi      */
                }
                else ZIBStdOut("error in EcoCauchy\n");
                break;
        case 1: if ((x==0.0)||(x==2.5)) {
                    fVals[0]  = 0.0;  /* Sigma */
                    fVals[1]  = 0.0;  /*     Xi      */
                }
                else ZIBStdOut("error in EcoCauchy\n");
                break;
      }
      return F_CONSTANT;
  }
```

6.2 Defining the new problem

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

```
int InitUserTime()
  {
    if (!SetTimeProblem("example1",varName,
                        EcoParabolic,
                        EcoLaplace,
                        EcoConvection,
                        EcoHelmholtz,
                        EcoSource,
                        EcoJacobian,
                        EcoInitialFunc,
                        EcoCauchy,
                        EcoDirichlet,
                        (int(*)(real,int,real,real*,int))nil));
    return true;
  }
```

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

6.3 Providing 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
```

6.4 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 example1
seliterate pbicgstabtime
selmatmul sparse
selestimate babuska
selrefine meanval
seltimeinteg rodas
window new automatic name solution1
graphic solution triangulation
seldraw 0
window new automatic name solution2
graphic solution triangulation
```

```
seldraw 1
timestepping timestep 0.0001 maxsteps 200 globTol 0.02 tEnd 10.0
```

Now we can start the program.

References

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

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

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

[4] J. Lang, A. Walter: *A Finite Element Method Adaptive in Space and Time for Nonlinear Reaction–Diffusion Systems.* IMPACT of Computing in Science and Engineering, **4**, p. 269–314 (1992)

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

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

Acknowledgements. I acknowledge R. Roitzsch for inspiring discussions on the field of programming and for making available always the current version of KASKADE. I also thank U. Nowak for supplying me with reference solutions obtained by his finite difference method.