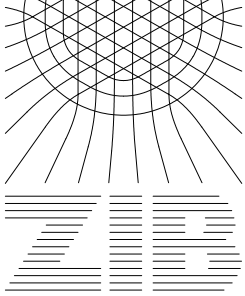
Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany



B. ERDMANN J. LANG R. ROITZSCH

KARDOS™ – User's Guide

ZIB-Report 02-42 (November 2002)

KARDOS™ – User's Guide

Bodo Erdmann* Jens Lang† Rainer Roitzsch*

Abstract

The adaptive finite element code KARDOS solves nonlinear parabolic systems of partial differential equations. It is applied to a wide range of problems from physics, chemistry, and engineering in one, two, or three space dimensions. The implementation is based on the programming language C. Adaptive finite element techniques are employed to provide solvers of optimal complexity. This implies a posteriori error estimation, local mesh refinement, and preconditioning of linear systems. Linearly implicit time integrators of *Rosenbrock* type allow for controlling the time steps adaptively and for solving nonlinear problems without using *Newton's* iterations. The program has proved to be robust and reliable.

The user's guide explains all details a user of KARDOS has to consider: the description of the partial differential equations with their boundary and initial conditions, the triangulation of the domain, and the setting of parameters controlling the numerical algorithm. A couple of examples makes familiar to problems which were treated with KARDOS.

We are extending this guide continuously. The latest version is available by network: http://www.zib.de/SciSoft/kardos/ (*Downloads*).

*Konrad-Zuse-Zentrum für Informationstechnik, Takustr. 7, D-14195 Berlin, Germany. E-Mail: {erdmann,roitzsch}@zib.de

†TU Darmstadt, Fachbereich Mathematik, Schlossgartenstr. 7, D-64289 Darmstadt, Germany. E-Mail: lang@mathematik.tu-darmstadt.de

Contents

1 Introduction

Dynamical process simulation is nowadays the central tool to assess the modelling process for large scale physical problems arising in such fields as biology, chemistry, metallurgy, medicine, and environmental science. Moreover, successful numerical methods are very attractive to design and control plants quickly and efficiently. Due to the great complexity of the established models, the development of fast and reliable algorithms has been a topic of continuing investigation during the last years.

One of the important requirements that software must meet today is to judge the quality of the numerical approximations in order to assess safely the modelling process. Adaptive methods have proven to work efficiently providing a posteriori error estimates and appropriate strategies to improve the accuracy where needed. They are now entering into real–life applications and starting to become a standard feature in simulation programs. In a set of publications (e.g., [51], [56], [60], [54], [59], [63], [67]) we presented one successful way to construct discretization methods adaptive in space and time which are applicable to a wide range of relevant problems. The proposed algorithms were implemented in the code KARDOS at the Zuse Institute Berlin. Here, the development of adaptive finite element codes started 1988. In the beginning there was the implementation of the adaptive multilevel code KASKADE solving linear elliptic problems. Starting with the basic ideas of Deuflhard, Leinen and Yserentant ([19]) there were a lot of extensions and some applications, e.g., [42], [43], [79], [10], [44], [11], [12], [13], [14], [45], [26], [28], [15], [8], [22], [17]. Technical informations about the C– and C++– versions of KASKADE can be found in [27] and [7], or on the KASKADE website [2].

KARDOS is based on the elliptic solver but includes essential extensions. It treats nonlinear systems of parabolic type by coupling adaptive control in time and in space.

We consider the nonlinear initial boundary value problem

$$\begin{aligned} B(x,t,u,\nabla u)\partial_t u &= \nabla\cdot(D(x,t,u,\nabla u)\nabla u) + F(x,t,u,\nabla u), \\ &\qquad\qquad x\in\Omega, t\in(0,T], \\ \mathcal{B}u(x,t) &= g(x,t,u(x,t)), \quad x\in\partial\Omega, t\in(0,T], \\ u(x,0) &= u_0(x), \quad x\in\overline{\Omega}, \end{aligned} \tag{1}$$

where $\Omega\subset\mathbb{R}^d$, $d=1,2$ or 3, is a bounded open domain with smooth boundary

$\partial\Omega$ lying locally on one side of Ω, and $T>0$. The coefficient functions $B = B(x,t,u,\nabla u)$, $D = D(x,t,u,\nabla u)$ and the right-hand side $F = F(x,t,u,\nabla u)$ may depend on the solution u and its gradient ∇u. In particular, F allows for a convective term $C \cdot \nabla u$, which also can be specified explicitely in KARDOS, compare Section 5. $C = C(x,t,u)$ may depend on the solution u. The boundary operator $\mathcal{B} = \mathcal{B}(x,t,u,\nabla u)$ stands for an appropriate system of boundary conditions and has to be interpreted in the sense of traces. The following boundary conditions are implemented:

- DIRICHLET type ($\mathcal{B} = I$), i.e.

$$u(x,t) = g(x,t,u(x,t)).$$

- CAUCHY type, i.e.

$$D(x,t,u,\nabla u)\frac{\partial u(x,t)}{\partial n} = g(x,t,u(x,t)).$$

- NEUMANN type, i.e.

$$D(x,t,u,\nabla u)\frac{\partial u(x,t)}{\partial n} = 0.$$

The function $u_0(x)$ describes the intial values. The unknown $u = u(x,t)$ is allowed to be vector–valued.

This guide is organized as follows. In Section 2, we give a summary of the underlying numerical concept. A more detailed description can be found in the book of LANG ([60]). A user who is not interested in the mathematical background is recommended to skip this section and to continue with Section 3, where we present a set of problems giving the motivation for using adaptive finite element techniques. Afterwards, in Section 4, we explain the structure of the code and give hints how to install the code. In Section 5 we finally describe in detail how the user can prepare the program in order to solve his problem. KARDOS can be controlled by its own command language which is presented in Section 6. Finally in the appendix, we give a collection of user functions corresponding to the examples we introduced in Section 3. These examples support the more general explanations given in section 5.

We are extending this guide continuously. The latest version is available by network: http://www.zib.de/SciSoft/kardos/ (*Downloads*).

Acknowledgement.

The authors want to thank all their collaborators which helped to make KARDOS to a successful code by bringing their problems into the code.

2 The Numerical Concept

In the classical method of lines (MOL) approach, the spatial discretization is done once and kept fixed during the time integration. Discrete solution values correspond to points on lines parallel to the time axis. Since adaptivity in space means to add or delete points, in an adaptive MOL approach new lines can arise and later on disappear. Here, we allow a local spatial refinement in each time step, which results in a discretization sequence first in time then in space. The spatial discretization is considered as a perturbation, which has to be controlled within each time step. Combined with a posteriori error estimates this approach is known as adaptive Rothe method. First theoretical investigations have been made by Bornemann [13] for linear parabolic equations. Lang and Walter [46] have generalized the adaptive Rothe approach to reaction–diffusion systems. A rigorous analysis for nonlinear parabolic systems is given in Lang [60]. For a comparative study, we refer to Deuflhard, Lang, and Nowak [24].

Since differential operators give rise to infinite stiffness, often an implicit method is applied to discretize in time. We use linearly implicit methods of Rosenbrock type, which are constructed by incorporating the Jacobian directly into the formula. These methods offer several advantages. They completely avoid the solution of nonlinear equations, that means no Newton iteration has to be controlled. There is no problem to construct Rosenbrock methods with optimum linear stability properties for stiff equations. According to their one–step nature, they allow a rapid change of step sizes and an efficient adaptation of the spatial discretization in each time step. Moreover, a simple embedding technique can be used to estimate the error in time satisfactorily. A description of the main idea of linearly implicit methods is given in a first subsection.

Stabilized finite elements are used for the spatial discretization to prevent numerical instabilities caused by advection–dominated terms. To estimate the error in space, the hierarchical basis technique has been extended to Rosenbrock schemes in Lang [60]. Hierarchical error estimators have been accepted to provide efficient and reliable assessment of spatial errors. They can be used to steer a multilevel process, which aims at getting a successively improved spatial discretization drastically reducing the size of the arising linear algebraic systems with respect to a prescribed tolerance (Bornemann, Erdmann, and Kornhuber [15], Deuflhard, Leinen and Yserentant [19], Bank and Smith [6]). A brief introduction to multilevel finite element methods is given in a second subsection.

The described algorithm has been coded in the fully adaptive software package KARDOS at the Zuse Institute Berlin. Several types of embedded Rosenbrock solvers and adaptive finite elements were implemented. KARDOS is based on the KASKADE–toolbox [27], which is freely distributed at [1]. Nowadays both codes are efficient and reliable workhorses to solve a wide class of PDEs in one, two, or three space dimensions.

2.1 Linearly Implicit Methods

In this section a short description of the linearly implicit discretization idea is given. More details can be found in the books of HAIRER and WANNER [40], DEUFLHARD and BORNEMANN [21], STREHMEL and WEINER [87]. For ease of presentation, we firstly set $B=I$ in (1) and consider the autonomous case. Then we can look at (1) as an abstract Cauchy problem of the form

$$\partial_t u = f(u)\,, \quad u(t_0) = u_0\,, \quad t > t_0\,, \tag{2}$$

where the differential operators and the boundary conditions are incorporated into the nonlinear function $f(u)$. Since differential operators give rise to infinite stiffness, often an implicit discretization method is applied to integrate in time. The simplest scheme is the implicit (backward) Euler method

$$u_{n+1} = u_n + \tau\, f(u_{n+1})\,, \tag{3}$$

where $\tau = t_{n+1} - t_n$ is the step size and u_n denotes an approximation of $u(t)$ at $t = t_n$. This equation is implicit in u_{n+1} and thus usually a Newton–like iteration method has to be used to approximate the numerical solution itself. The implementation of an efficient nonlinear solver is the main problem for a fully implicit method.

Investigating the convergence of Newton's method in function space, DEUFLHARD [23] pointed out that one calculation of the Jacobian or an approximation of it per time step is sufficient to integrate stiff problems efficiently. Using u_n as an initial iterate in a Newton method applied to (3), we find

$$(I - \tau\, J_n)\, K_n \;=\; \tau f(u_n)\,, \tag{4}$$

$$u_{n+1} \;=\; u_n + K_n\,, \tag{5}$$

where J_n stands for the Jacobian matrix $\partial_u f(u_n)$. The arising scheme is known as the *linearly implicit* Euler method. The numerical solution is now effectively computed by solving the system of linear equations that defines the increment K_n. Among the methods which are capable of integrating stiff

equations efficiently, linearly implicit methods are the easiest to program, since they completely avoid the numerical solution of nonlinear systems.

One important class of higher–order linearly implicit methods consists of extrapolation methods that are very effective in reducing the error, see DEUFLHARD [20]. However, in the case of higher spatial dimension, several drawbacks of extrapolation methods have shown up in numerical experiments made by BORNEMANN [11]. Another generalization of the linearly implicit approach we will follow here leads to Rosenbrock methods (ROSENBROCK [81]). They have found wide–spread use in the ODE context. Applied to (2) a so–called s–stage Rosenbrock method has the recursive form

$$(I - \tau\gamma_{ii}\, J_n)\, K_{ni} \;=\; \tau f(u_n + \sum_{j=1}^{i-1} \alpha_{ij}\, K_{nj}) + \tau J_n \sum_{j=1}^{i-1} \gamma_{ij}\, K_{nj}\,,\; i = 1(1)s\,, \quad (6)$$

$$u_{n+1} \;=\; u_n + \sum_{i=1}^{s} b_i K_{ni}\,, \quad (7)$$

where the step number s and the defining formula coefficients b_i, α_{ij}, and γ_{ij} are chosen to obtain a desired order of consistency and good stability properties for stiff equations (see e.g. HAIRER and WANNER [40], IV.7). We assume $\gamma_{ii} = \gamma > 0$ for all i, which is the standard simplification to derive Rosenbrock methods with one and the same operator on the left–hand side of (6). The linearly implicit Euler method mentioned above is recovered for $s{=}1$ and $\gamma{=}1$.

For the general system

$$B(t,u)\partial_t u = f(t,u)\,, \quad u(t_0) = u_0\,, \quad t > t_0\,, \quad (8)$$

an efficient implementation that avoids matrix–vector multiplications with the Jacobian was given by LUBICH and ROCHE [71]. In the case of a time– or solution–dependent matrix B, an approximation of $\partial_t u$ has to be taken into account, leading to the generalized Rosenbrock method of the form

$$\begin{aligned}\left(\frac{1}{\tau\gamma}B(t_n,u_n) - J_n\right) U_{ni} \;=\; & f(t_i,U_i) - B(t_n,u_n) \sum_{j=1}^{i-1} \frac{c_{ij}}{\tau}\, U_{nj} + \tau\gamma_i C_n \\ & +\; (B(t_n,u_n) - B(t_i,U_i))\, Z_i\,, \quad i = 1(1)s\,,\end{aligned} \quad (9)$$

where the internal values are given by

$$t_i{=}t_n + \alpha_i\tau\,, \quad U_i{=}u_n + \sum_{j=1}^{i-1} a_{ij}\, U_{nj}\,, \quad Z_i{=}(1-\sigma_i)z_n + \sum_{j=1}^{i-1} \frac{s_{ij}}{\tau}\, U_{nj},$$

and the Jacobians are defined by

$$\begin{aligned} J_n &:= \partial_u (f(t,u) - B(t,u)z)_{|u=u_n, t=t_n, z=z_n} \,, \\ C_n &:= \partial_t (f(t,u) - B(t,u)z)_{|u=u_n, t=t_n, z=z_n} \,. \end{aligned}$$

This yields the new solution

$$u_{n+1} = u_n + \sum_{i=1}^{s} m_i \, U_{ni}$$

and an approximation of the temporal derivative $\partial_t u$

$$z_{n+1} = z_n + \sum_{i=1}^{s} m_i \, \big(\frac{1}{\tau} \sum_{j=1}^{i} (c_{ij} - s_{ij}) U_{nj} + (\sigma_i - 1) z_n \big) \,.$$

The new coefficients can be derived from α_{ij}, γ_{ij}, and b_i [71]. In the special case $B(t,u) = I$, we get (6) setting $U_{ni} = \tau \sum_{j=1,\ldots,i} \gamma_{ij} K_{nj}$, $i = 1, \ldots, s$.

Various Rosenbrock solvers have been constructed to integrate systems of the form (8). An important fact is that the formulation (8) includes problems of higher differential index. Thus, the coefficients of the Rosenbrock methods have to be specially designed to obtain a certain order of convergence. Otherwise, order reduction might happen. In [73, 71], the solver ROWDAIND2 was presented, which is suitable for semi–explicit index 2 problems. Among the Rosenbrock methods suitable for index 1 problems we mention ROS2 [18], ROWDA3 [74], ROS3P [64], and RODASP [86]. More informations can be found in [60].

Usually, one wishes to adapt the step size in order to control the temporal error. For linearly implicit methods of Rosenbrock type a second solution of inferior order, say $\hat{p}$, can be computed by a so–called embedded formula

$$\begin{aligned} \hat{u}_{n+1} &= u_n + \sum_{i=1}^{s} \hat{m}_i U_{ni} \,, \\ \hat{z}_{n+1} &= z_n + \sum_{i=1}^{s} \hat{m}_i \, \big(\frac{1}{\tau} \sum_{j=1}^{i} (c_{ij} - s_{ij}) U_{nj} + (\sigma_i - 1) z_n \big) \,, \end{aligned}$$

where the original weights m_i are simply replaced by $\hat{m}_i$. If p is the order of u_{n+1}, we call such a pair of formulas to be of order $p(\hat{p})$. Introducing an appropriate scaled norm $\| \cdot \|$, the local error estimator

$$r_{n+1} = \| u_{n+1} - \hat{u}_{n+1} \| + \| \tau (z_{n+1} - \hat{z}_{n+1}) \| \tag{10}$$

can be used to propose a new time step by

$$\tau_{n+1} = \frac{\tau_n}{\tau_{n-1}} \left(\frac{TOL_t \, r_n}{r_{n+1} \, r_{n+1}} \right)^{1/(\hat{p}+1)} \tau_n \, . \tag{11}$$

Here, TOL_t is a desired tolerance prescribed by the user. This formula is related to a discrete PI–controller first established in the pioneering works of Gustaffson, Lundh, and Söderlind [38, 37]. A more standard step size selection strategy can be found in Hairer, Nørsett, and Wanner ([39], II.4).

Rosenbrock methods offer several structural advantages. They preserve conservation properties like fully implicit methods. There is no problem to construct Rosenbrock methods with optimum linear stability properties for stiff equations. Because of their one–step nature, they allow a rapid change of step sizes and an efficient adaptation of the underlying spatial discretizations as will be seen in the next section. Thus, they are attractive for solving real world problems.

2.2 Multilevel Finite Elements

In the context of PDEs, system (9) consists of linear elliptic boundary value problems, possibly advection–dominated. In the spirit of spatial adaptivity a multilevel finite element method is used to solve this system. The main idea of the multilevel technique consists of replacing the solution space by a sequence of discrete spaces with successively increasing dimension to improve their approximation property. A posteriori error estimates provide the appropriate framework to determine where a mesh refinement is necessary and where degrees of freedom are no longer needed. Adaptive multilevel methods have proven to be a useful tool for drastically reducing the size of the arising linear algebraic systems and to achieve high and controlled accuracy of the spatial discretization (see e.g. Bank [5], Deuflhard, Leinen, and Yserentant [19], Lang [56]).

Let T_h be an admissible finite element mesh at $t = t_n$ and S_h^q be the associated finite dimensional space consisting of all continuous functions which are polynomials of order q on each finite element $T \in T_h$. Then the standard Galerkin finite element approximation $U_{ni}^h \in S_h^q$ of the intermediate values U_{ni} satisfies the equation

$$(L_n \, U_{ni}^h, \phi) = (r_{ni}, \phi) \quad \text{for all } \phi \in S_h^q \, , \tag{12}$$

where L_n is the weak representation of the differential operator on the left–hand side in (9) and r_{ni} stands for the entire right–hand side in (9). Since the operator L_n is independent of i its calculation is required only once within each time step.

It is a well–known inconvenience that the solutions U_{ni}^h may suffer from numerical oscillations caused by dominating convective and reactive terms as well. An attractive way to overcome this drawback is to add locally weighted residuals to get a stabilized discretization of the form

$$(L_n\, U_{ni}^h, \phi) + \sum_{T\in T_h} (L_n\, U_{ni}^h, w(\phi))_T = (r_{ni}, \phi) + \sum_{T\in T_h} (r_{ni}, w(\phi))_T\,, \tag{13}$$

where $w(\phi)$ has to be defined with respect to the operator L_n (see e.g. Franca and Frey [33], Lube and Weiss [70], Tobiska and Verfürth [88]). Two important classes of stabilized methods are the streamline diffusion and the more general Galerkin/least–squares finite element method.

The linear systems are solved by direct or iterative methods. While direct methods work quite satisfactorily in one–dimensional and even two–dimensional applications, iterative solvers such as Krylov subspace methods perform considerably better with respect to CPU–time and memory requirements for large two– and three–dimensional problems. We mainly use the Bicgstab–algorithm [90] with ILU–preconditioning.

After computing the approximate intermediate values U_{ni}^h a posteriori error estimates can be used to give specific assessment of the error distribution. Considering a hierarchical decomposition

$$S_h^{q+1} = S_h^q \oplus Z_h^{q+1}\,, \tag{14}$$

where Z_h^{q+1} is the subspace that corresponds to the span of all additional basis functions needed to extend the space S_h^q to higher order, an attractive idea of an efficient error estimation is to bound the spatial error by evaluating its components in the space Z_h^{q+1} only. This technique is known as hierarchical error estimation and has been accepted to provide efficient and reliable assessment of spatial errors (Bornemann, Erdmann, and Kornhuber [15], Deuflhard, Leinen and Yserentant [19], Bank and Smith [6]). In Lang [60], the hierarchical basis technique has been carried over to time–dependent nonlinear problems. Defining an a posteriori error estimator $E_{n+1}^h \in Z_h^{q+1}$ by

$$E_{n+1}^h = E_{n0}^h + \sum_{i=1}^{s} m_i E_{ni}^h \tag{15}$$

with E_{n0}^h approximating the projection error of the initial value u_n in Z_h^{q+1} and E_{ni}^h estimating the spatial error of the intermediate value U_{ni}^h, the local spatial error for a finite element $T \in T_h$ can be estimated by $\eta_T := \|E_{n+1}^h\|_T$. The error estimator E_{n+1}^h is computed by linear systems which can be derived from (13). For practical computations the spatially global calculation of E_{n+1}^h is normally approximated by a small element–by–element calculation. This leads to an efficient algorithm for computing a posteriori error estimates which can be used to determine an adaptive strategy to improve the accuracy of the numerical approximation where needed. A rigorous a posteriori error analysis for a Rosenbrock–Galerkin finite element method applied to nonlinear parabolic systems is given in LANG [60]. In our applications we applied linear finite elements and measured the spatial errors in the space of quadratic functions.

In order to produce a nearly optimal mesh, those finite elements T having an error η_T larger than a certain threshold are refined. After the refinement improved finite element solutions U_{ni}^h defined by (13) are computed. The whole procedure solve–estimate–refine is applied several times until a prescribed spatial tolerance $\|E_{n+1}^h\| \leq TOL_x$ is reached. To maintain the nesting property of the finite element subspaces coarsening takes place only after an accepted time step before starting the multilevel process at a new time. Regions of small errors are identified by their η–values.

3 Applications

Equation (1) comprises all problems which can be treated by KARDOS.

In this section we present some examples. Details of implementation follow in the next chapters and in the Appendix.

3.1 Determination of Thermal Conductivity

Measurement of the thermal conductivity of molten materials is very difficult, mainly because the mathematical modelling of heat transfer processes at high temperatures, with several different media involved, is far from being solved. However, the scatter of the experimental data presented by different authors using several methods is so large that any scientific or technological application is strongly limited without serious approximations. The development of new instruments for the measurement of the thermal conductivity of molten salts, metals, and semiconductors implies the design of a specific sensor for the measurement of temperature profiles in the melt, apart from the necessary electronic equipment for the data acquisition and processing, furnaces, and gas/vacuum manifolds.

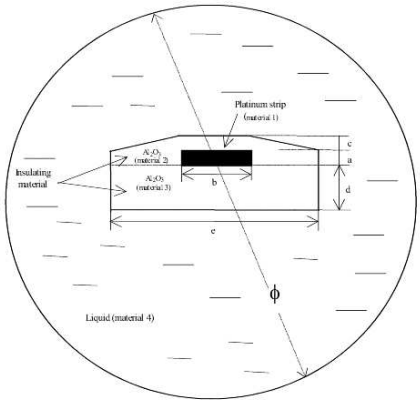


Figure 1: Scheme of the hot-strip sensor.

In our application ([68], [69]), we consider a planar, electrically conducting (metallic) element mounted within an insulating substratum. This equipment is surrounded by a material whose thermal properties have to be determined, see Figure 1. From an initial state of equilibrium, Ohmic dissipation within the metallic strip results in a temperature rise on the strip, and a

conductive thermal wave spreads out from it through the substratum into the surrounding material. The temperature history of the metallic strip, as indicated by its change of electrical resistance, is determined partly by the thermal conductivity and the diffusivity of this material.

In order to identify the thermal conductivity of the material from available measurements, a heat transfer equation in two space dimensions has to be solved several times

$$\rho C_p \partial T/\partial t = \nabla \cdot (\lambda \nabla T) + Q$$

The properties λ, ρ, and C_p of the materials are piecewise constant. This equation has to be applied to three distinct regions: to the strip, to the substrate, and to the material.

Due to strongly localised source terms and different properties of the involved materials, we observe at the beginning steep gradients of the temperature profiles that decrease in time. In such a situation, a method with automatic control of spatial and temporal discretization is an appropriate tool, see Figure 2.

Figure 3 shows the result obtained for a specially designed sensor. The agreement between experimental and numerical data is quite satisfactory, and it results in a water thermal conductivity of $0.606 W m^{-1} K^{-1}$ at $25^o C$, a value within 0.1% of the recommended one.

3.2 Vertical Bubble Reactor

Gas–fluid systems give rise to propagating phase boundaries changing their shape and size in time. In the following we consider a synthesis process of two gaseous chemicals A and B in a cylindrical bubble reactor filled with a catalytic fluid (see Figure 4).

The bubbles stream in at the lower end of the reactor and rise to the top while dissolving and reacting with each other. The right proportions of such reactors depend, among other things, on the rising behaviour of the bubbles and specific reaction velocities. Therefore, modelling and simulation of the underlying two–phase system can provide engineers with useful knowledge necessary to construct economical plants.

A fully three–dimensional description of the synthesis process would become too complicated. We have used a one–dimensional two–film model developed by Ruppel [82]. It is based mainly on the assumption that the interaction

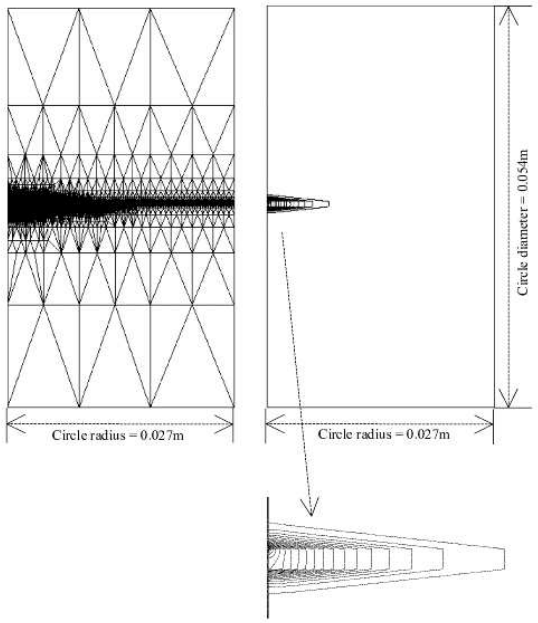


Figure 2: Adaptive grid and isotherms of the solution at time t = 1.0.

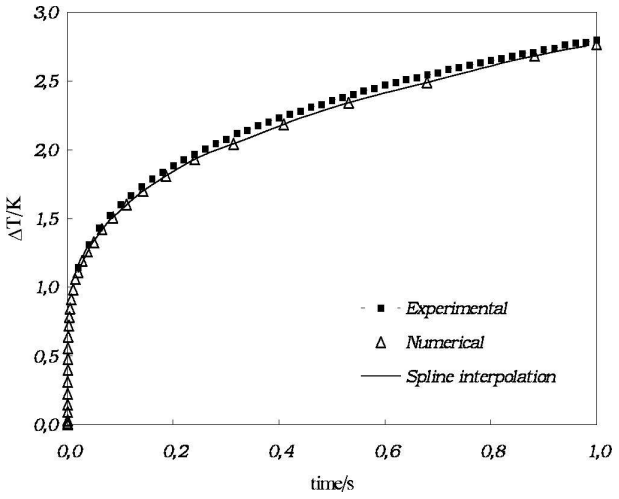


Figure 3: Simulation for water at 25^oC.

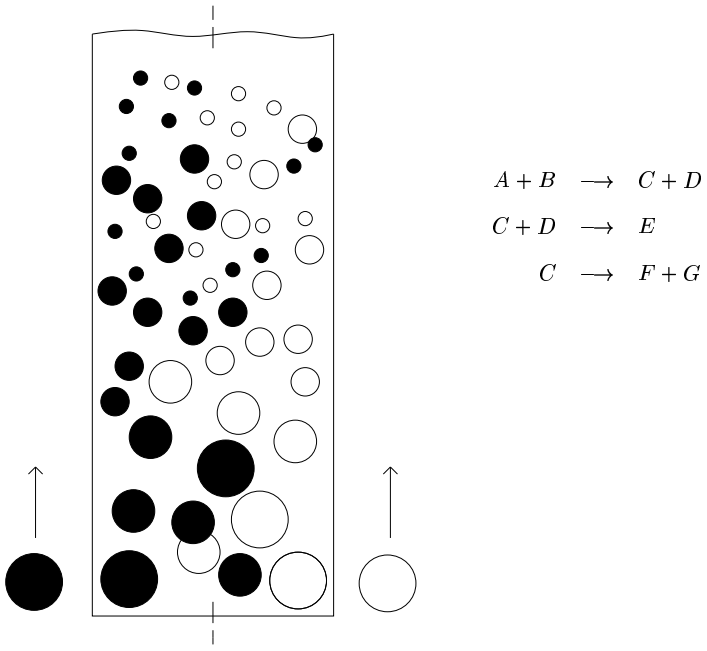


Figure 4: Bubble reactor in section and reaction mechanism.

between the gas and the reactor fluid (bulk) takes place in very thin layers (films) with time–independent thickness (see Figure 5). In the first film F_1 the chemical A dissolves into the bulk. From there it is transported very fast to the second film F_2 where reaction with chemical B takes place. As a result new chemicals C and D are produced causing further reactions.

Defining the assignment $(A, B, C, D, E, F, G) \to (u_1, \ldots, u_7)$ the model can be expressed by the following equations.

Diffusive process in F_1 only for the chemical A:

$$\begin{aligned} -D_1 \frac{\partial^2 u_1}{\partial x^2} &= 0, \qquad x \in (\xi_1, \xi_2), \\ \frac{\beta_1}{\alpha_1 D_1} u_1(\xi_1) - \frac{\partial u_1}{\partial x}(\xi_1) &= c_2 \frac{\beta_1}{D_1}, \qquad u_1(\xi_2^-) = u_1(\xi_2^+)\,. \end{aligned} \tag{16}$$

Transport of all the chemicals through the bulk:

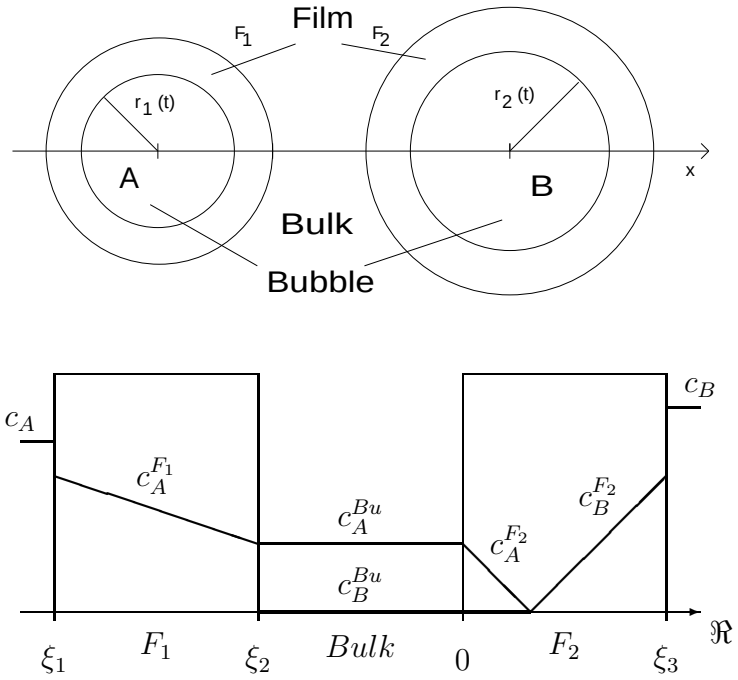


Figure 5: Two–film model based on interaction zones with constant thickness (top). Behaviour of the chemicals A and B on the computational domain (bottom).

$$
\begin{aligned}
c_1\frac{\partial u_i}{\partial t} &= S_1(t)D_i\frac{\partial u_i}{\partial x}(\xi_2^-) + S_2(t)D_i\frac{\partial u_i}{\partial x}(0^+),\ x\in(\xi_2,0), t>0\,,\\
u_1(\xi_2^+) &= u_1(\xi_2^-),\ \ \frac{\partial u_i}{\partial x}(\xi_2^+) = 0,\ i\neq 1,\ \ u_i(0^-) = u_i(0^+)\,.
\end{aligned}
\tag{17}
$$

Reaction and diffusion in F_2:

$$
\begin{aligned}
-D_i\frac{\partial^2 u_i}{\partial x^2} &= \sum_j k_{i,j}u_j u_i,\ x\in(0,\xi_3), t>0.\\
u_i(0^+) = u_i(0^-),\ \ \frac{\beta_2}{\alpha_2 D_2}u_2(\xi_3) &+ \frac{\partial u_2}{\partial x}(\xi_3) = c_3\frac{\beta_2}{D_2},\\
\frac{\partial u_i}{\partial x}(\xi_3) &= 0,\ i\neq 2,\ \ u_i(x,0) = u_i^0.
\end{aligned}
\tag{18}
$$

Here, D_i and β_i denote the diffusion and the coupling coefficient of the i–th component, and α_i represents the Henry coefficient. The specific exchange areas S_1 and S_2 depend nonlinearly on the decreasing bubble radii $r_1(t)$ and $r_2(t)$.

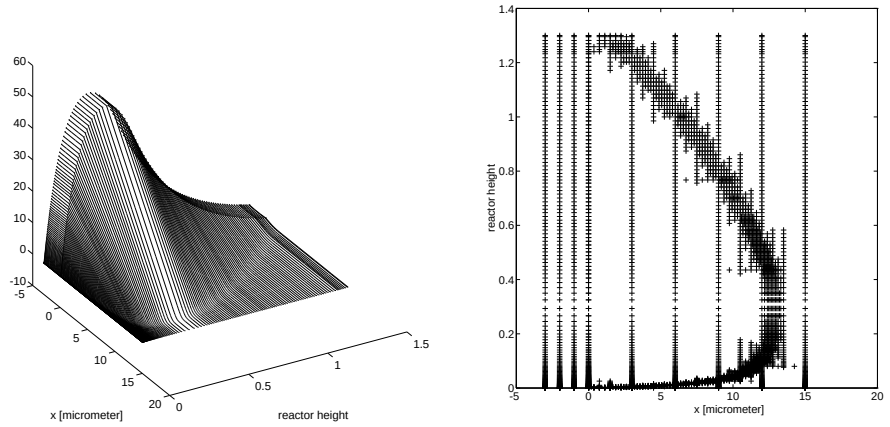


Figure 6: Evolution of the chemical component A (left) and of the grid (right) where the reactor height is taken as time axis.

As a consequence of the applied two–film model the dynamical synthesis process can be simulated with a fixed spatial domain involving the bulk and the film F_2. Equation (16) is solved analytically. Clearly, the spatial discretization needs some adaptation due to the presence of internal boundary conditions between bulk and film. We refer to [52] for a more thorough discussion. Here we will report only on the temporal evolution of the grid used to resolve the reaction front in the film $F_2 = [0, 15]\mu m$. Fig. 6 shows that at the beginning the reaction front is travelling very fast from the outer to the inner boundary of the film where the chemical B enters permanently. During the time period $[0.1, 0.5]$ the reaction zone does not change its position which allows larger time steps. After that with decreasing concentration of the chemical A at the outer boundary the front travels back, but now with moderate speed. Obviously, the adaptively controlled discretization is able to follow automatically the dynamics of the problem.

3.3 Semiconductor

An elementary process step in the fabrication of silicon-based integrated circuits is the diffusion mechanism of dopant impurities into silicon. The study of diffusion processes is of great technological importance since their quality strongly influences the quality of electronical materials. Impurity atoms of higher or lower chemical valence, such as arsenic, phosphorus, and boron, are introduced under high temperatures ($900 - 1100^oC$) into a silicon crystal to change its electrical properties. This is the central process of modern silicon technology. Various pair-diffusion models have been developed to allow accurate modelling of device processing (see Figure 7).

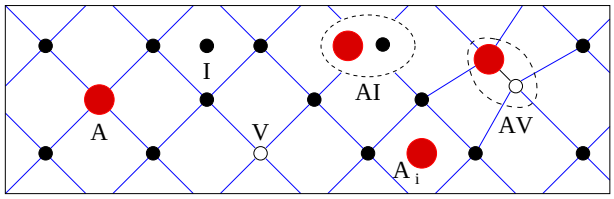


Figure 7: Scheme of pair diffusion.

See for more details in [65].

Multiple Species Diffusion: Dopant atoms occupy substitutional sites in the silicon crystal lattice, losing (donors such as arsenic and phosphorus) or gaining (acceptors such as boron) at the same time an electron. One fundamental interest in semiconductor devices modelling is to study the interaction of two unequally charged dopants and the influence of the chemical potential. Here, we select arsenic (As) and boron (B). In the following Figure 8, the shape of the initial dopant implantations at 950^oC is visualized. The solutions obtained after thirty minutes show that the boron profile is mainly influenced by the chemical potential while the arsenic concentration is changed only slowly by diffusion. It can nicely be seen that the dynamic mesh chosen by KARDOS is well-fitted to the local behaviour of the solution.

Phosphorus Diffusion: Here, we simulate phosphorus diffusion using a detailed par-diffusion model. Since a diffusion mechanism based only on the direct interchange with neighbouring silicon atoms turns out to be energetically unfavourable, native point defects called interstitials and vacancies are taken into account. The phosphorus concentration shows its typical "kink and tail" behaviour, a phenomenon which is known as anomalous diffusion of phosphorus.

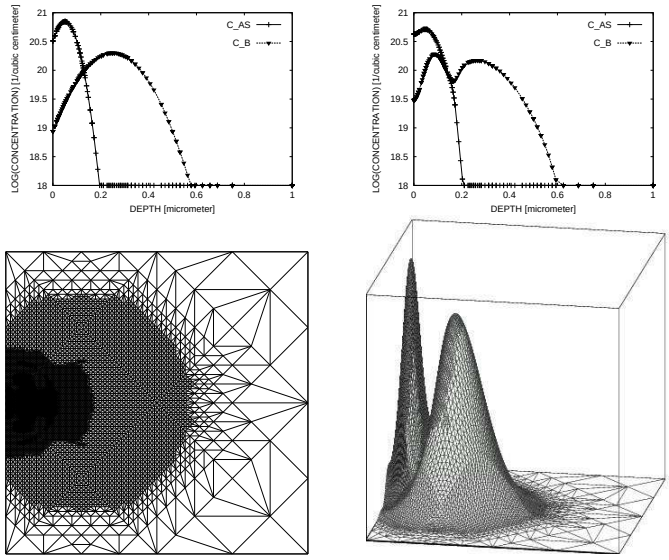


Figure 8: .

3.4 Pattern Formation

Numerical simulations of a simple reaction-diffusion model reveal a surprising variety of irregular patterns changing in time and in space. These patterns arise in response to finite-amplitude perturbations. Some of them resemble the steady irregular patterns recently observed in thin gel reactor experiments. Others consist of spots that grow until they reach a critical size, at which time they devide in two. If in some region the spots become overcrowded, all of the spots in that region decay into the uniform background. For details of modelling we refer to Pearson [77].

Patterns occur in nature at very different scales, which explains the great scientific interest in new pattern formation phenomena.

In this application, we collect some numerical studies recently done to test our adaptive code.

Gray-Scott Model: Spot-Replication: The spot–replication in the model of Gray-Scott is defined by reaction-diffusion equations in dimensionless units of the following type:

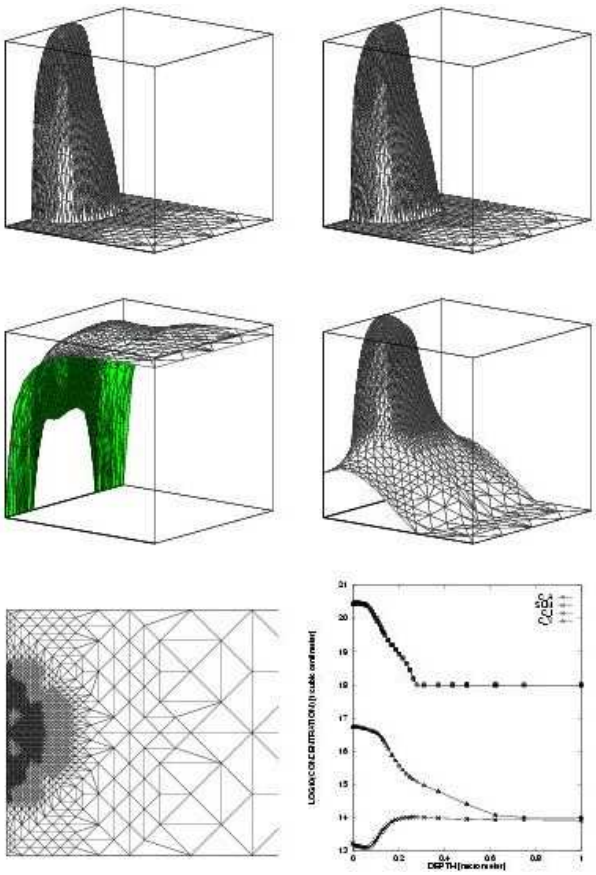


Figure 9: Snapshot at t=3min of total and substitutional phosphorus concentration, interstitials and vacancies and 1D-cut of all.

$$\begin{aligned} \frac{\partial u}{\partial t} - \nabla(\mu_1 \nabla u) &= -uv^2 + F(1-u) \\ \frac{\partial v}{\partial t} - \nabla(\mu_2 \nabla v) &= uv^2 - (F+k)v. \end{aligned}$$

Here k is the dimensionless rate constant of the second reaction and F is the dimensionless feed rate. The system size is 2.5 by 2.5, and the diffusion coefficients are $\mu_1 = 2.0 \cdot 10^{-5}$ and $\mu_2 = 10^{-5}$. The boundary conditions are periodic. Initially, the entire system was placed in the trivial state ($u = 1$, $v = 0$). An area located symmetrically about the center of the grid was then perturbed to ($u = 1/2$, $v = 1/4$). These conditions were then perturbed with 1% random noise in order to break the square symmetry.

We show some results in the Figures 10 and 11.

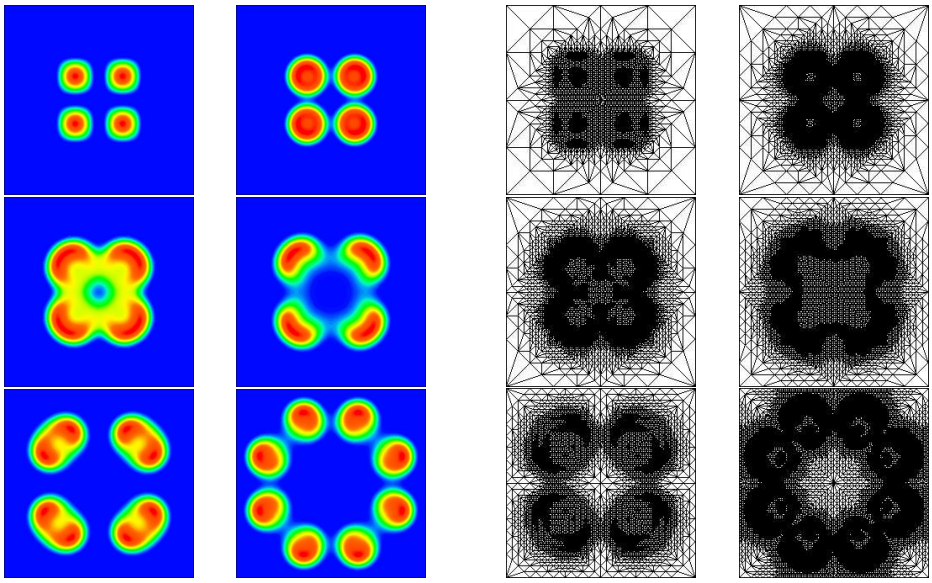


Figure 10: Spot-Replication: Concentration of second component at t=0, 50, 100, 150, 350, 550 (left). Corresponding adaptive FE–grids (right).

Complex patterns in a simple system:

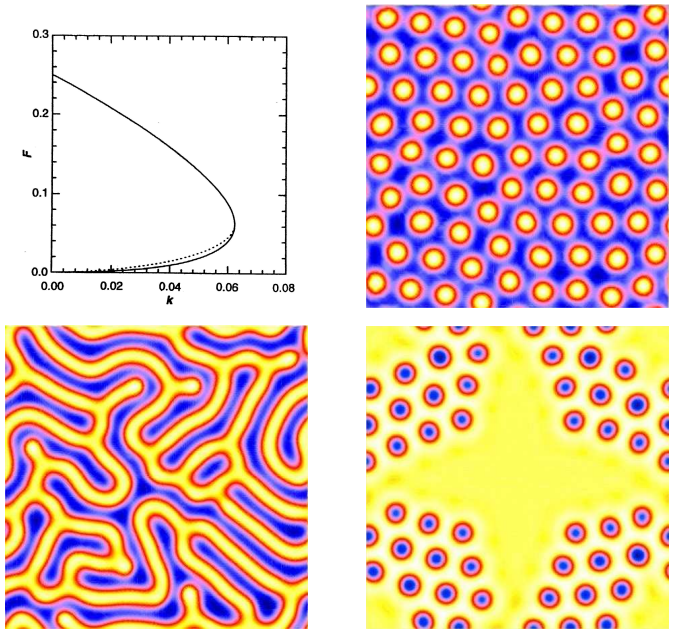


Figure 11: Phase diagram of the reaction kinetics (top left). Pattern for F=0.024, k=0.060 (top right), Patterns for F=0.05, k=0.062 (bottom left), and F=0.05, k=0.060 (bottom right).

Animal Coat Markings: Similar models were provided by J.D. Murray [72] in order to describe animal coat markings. A typical result is shown in Figure 12.

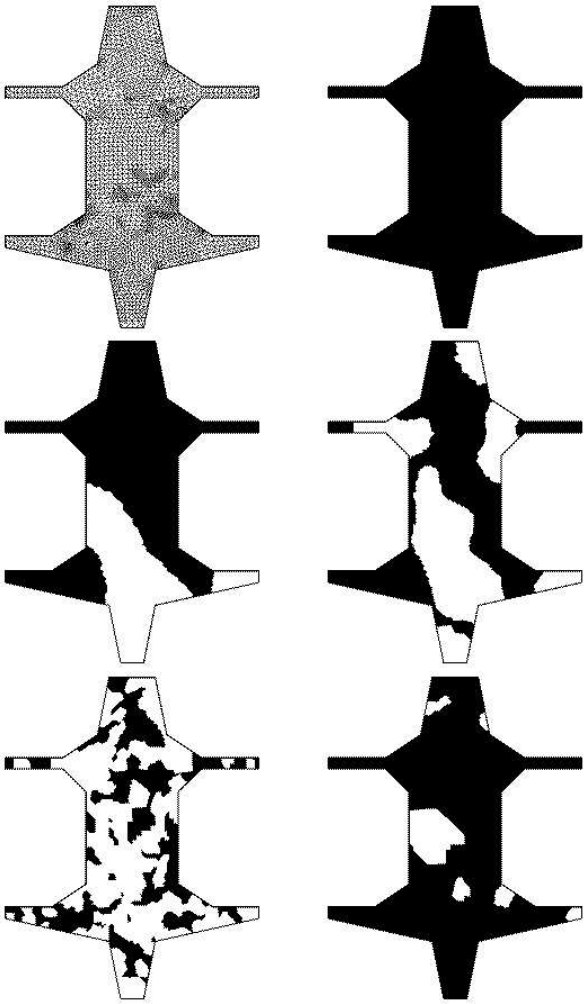


Figure 12: Animal coat markings

3.5 Thermo-Diffusive Flames

Combustion problems are known to range among the most demanding for spatial adaptivity when the thin flame front is to be resolved numerically. This is often required as the inner structure of the flame determines global properties such as the flame speed, the formation of cellular patterns or even

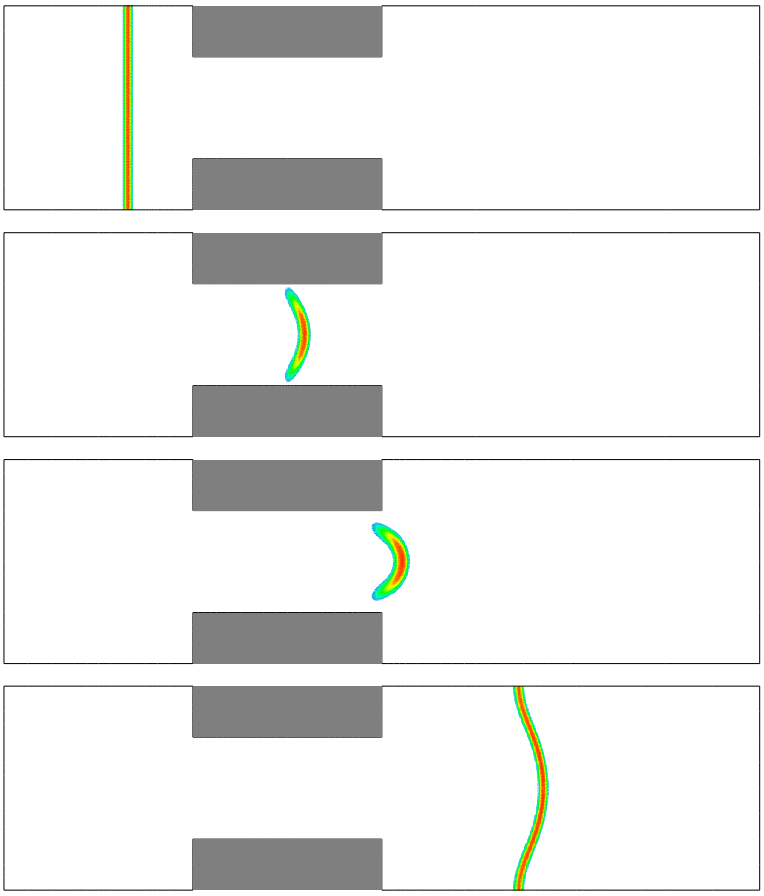


Figure 13: Flame through cooled grid, $Le = 1$, $k = 0.1$. Reaction rate ω at $t = 1$, 20, 40, 60.

more important the mass fraction of reaction products (e.g. NO_x formation). A large part of numerical studies in this field is devoted to the different instabilities of such flames. The observed phenomena include cellular patterns, spiral waves, and transition to chaotic behaviour.

In Figures 13 and 14 we show laminar flames moving through a cooled grid and in Figure 15 a reaction front in a non-uniformly packed solid.

Introducing the dimensionless temperature $\theta = (T - T_{unburnt})/(T_{burnt} - T_{unburnt})$, denoting by Y the species concentration, and assuming constant diffusion coefficients yields

$$\begin{aligned} \frac{\partial \theta}{\partial t} - \nabla^2 \theta &= \omega \\ \frac{\partial Y}{\partial t} - \frac{1}{Le} \nabla^2 Y &= -\omega \end{aligned}$$

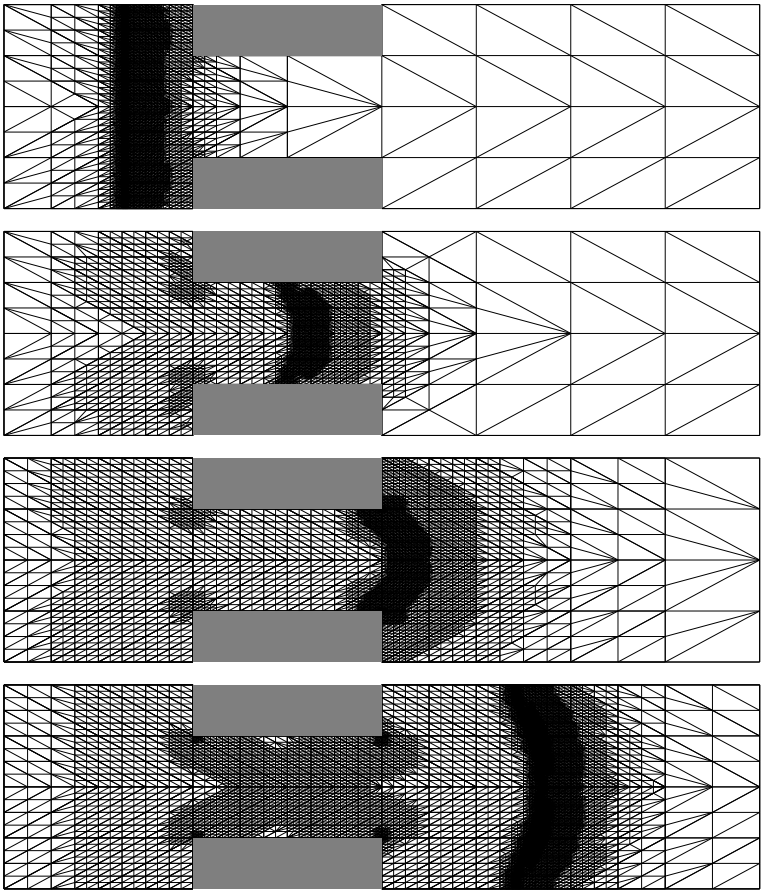


Figure 14: Flame through cooled grid, $Le = 1$, $k = 0.1$. Spatial discretization at $t = 1$, 20, 40, 60.

where the Lewis number Le is the ratio of diffusivity of heat and diffusivity of mass. We use a simple one-species reaction mechanism governed by an Arrhenius law

$$\omega = \frac{\beta^2}{2Le} Y e^{\frac{\beta(\theta-1)}{1+\alpha(\theta-1)}},$$

in which an approximation for large activation energy has been employed. For details of the problem see in [60].

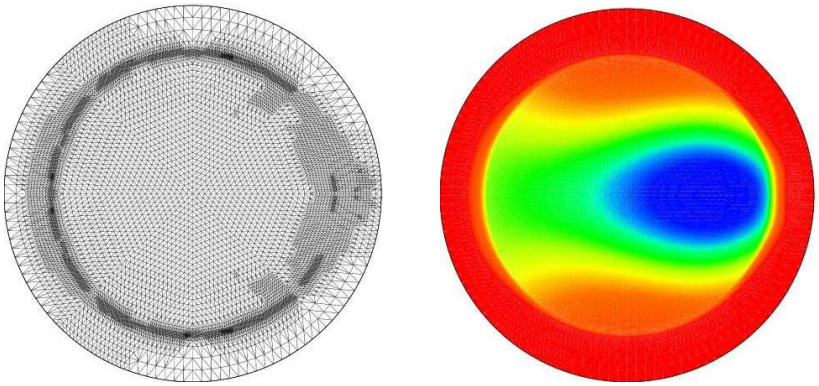


Figure 15: Non-uniformly packed solid. Concentration of the reactant and grid at time t = 0.07, where blue corresponds to the unburnt and red to the burnt state.

A characterictic of the example from solid-solid combustion is that convection is impossible and that the macroscopic diffusion for the species in solids is in general negligible with respect to heat conductivity. With the heat diffusion time scale as reference, the equations for a one step chemical alloying reaction read

$$\begin{aligned} \frac{\partial T}{\partial t} - \kappa \nabla^2 T &= Q\omega \\ \frac{\partial Y}{\partial t} &= -\omega \end{aligned}$$

where T is the temperature divided by the reference temperature, Y the concentration of the deficient reactant and Q a heat dissipation parameter. Concerning the reaction term quite a number of different models are employed in the literature. They generally contain an Arrhenius term for the temperature dependence and use a first order reaction, i.e.,

$$\omega = K_0 Y e^{-\frac{E}{T}},$$

E is a dimensionless activation energy. Besides these equations we provide appropriate boundary and initial values. Details can be found in the book of LANG [60].

Stability of Flame Balls: The profound understanding of premixed gas flames near extinction or stability limits is important for the design of efficient, clean-burning combustion engines and for the assessment of fire and explosion hazards in oil refineries, mine shafts, etc. Surprisingly, the near-limit behaviour of very simple flames is still not well-known. Since these phenomena are influenced by bouyant convection, typically experiments are performed in a micro-gravity environment. Under these conditions transport mechanisms such as radiation and small Lewis number effects, the ratio of thermal diffusivity to the mass diffusivity, come into the play, see the Figure 16.

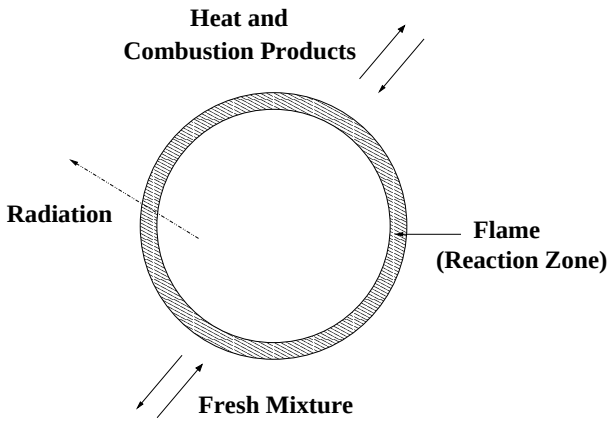


Figure 16: Configuration of a stationary flame ball. Diffusional fluxes of heat and combustion products (outwards) and of fresh mixture (inwards) together with radiative heat loss cause a zero mass-averaged velocity

Seemingly stable flame balls are one of the most exciting appearances which were accidentally discovered in drop-tower experiments by RONNEY (1990) and confirmed later in parabolic aircraft flights. First theoretical investigations on purely diffusion-controlled stationary spherical flames were done by ZELDOVICH (1944). 40 years later his flame balls were predicted to be unstable (1984). However, encouraged by the above new experimental discoveries, BUCKMASTER ET AL. (1990) have shown that for low Lewis numbers flame

balls can be stabilized including radiant heat loss which was not considered before.

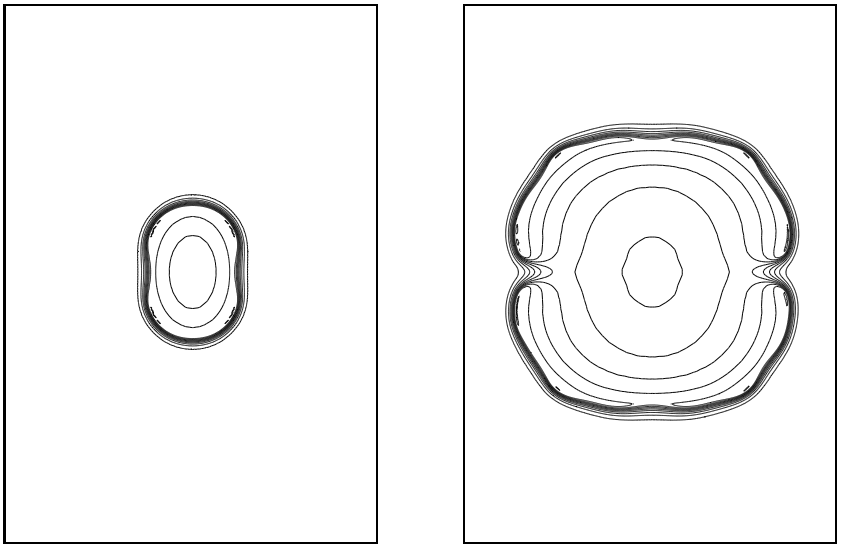


Figure 17: Two-dimensional flame ball with Le = 0.3, c = 0.01. Isothermals T = 0.1, 0.2,..., 1.0 at times t = 10 and 30.

The processes are governed by equations of the following structure:

$$\begin{aligned} \frac{\partial T}{\partial t} - \nabla^2 T &= w - s, \\ \frac{\partial Y}{\partial t} - \frac{1}{Le}\nabla^2 Y &= -w, \\ w &= \frac{\beta^2}{2Le} Y \exp\left(\frac{\beta(T-1)}{1+\alpha(T-1)}\right), \\ s &= c\frac{\bar{T}^4 - \bar{T}_u^4}{(\bar{T}_b - \bar{T}_u)^4}. \end{aligned}$$

Here, $T := ((\overline{T}) - (\overline{T})_u)/((\overline{T})_b - (\overline{T})_u)$ is the dimensionless temperature determined by the dimensional temperatures $\bar{T}$, $\bar{T}_u$, and $\bar{T}_b$, where the indices u and b refer to the unburnt and burnt state of an adiabetic plane flame, respectively. Y represents the mass fraction of the deficient component of the mixture. The chemical reaction rate w is modelled by an one-step Arrhenius term incorporating the dimensionless activation energy β, the Lewis number Le, and the heat dissipation parameter $\alpha := ((\overline{T})_b - (\overline{T})_u)/(\overline{T})_b$. Heat loss is

generated by a radiation term s modelled for the optically thin limit. Further explanation can be found in the book of Wouver, Saucez, and Schiesser [63].

Typically, instabilities occur which result in a local quenching of the flame as can be seen in the Figure 17. After a while the flame is splitted into two seperate smaller flames. Nevertheless, we found quasi-stationary flame ball configurations, fixing the heat loss by radiation and varying the initial radii for a circular flame.

3.6 Nonlinear Modelling of Heat Transfer in Regional Hyperthermia

Hyperthermia, i.e., heating tissue to 42^oC, is a method of cancer therapy. It is normally applied as an additive therapy to enhance the effect of conventional radio- or chemotherapy. The standard way to produce local heating in the human body is the use of electromagnetic waves. We are mainly interested in regional hyperthermia of deep-seated tumors. For this type of treatment usually a phased array of antennas surrounding the patient is used, see Figure 18.

Figure 18: Patient model (torso) and hyperthermia applicator. The patient is surrounded by eight antennas emitting radio waves. A water-filled bolus is placed between patient and antennas

The distribution of absorbed power within the patient's body can be steered by selecting the amplitudes and phases of the antennas' driving voltages. The space between the body and the antennas is filled by a so-called water bolus to avoid excessive heating of the skin.

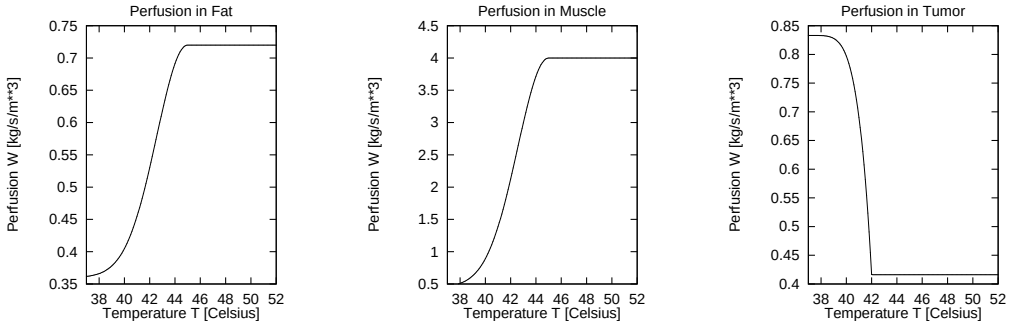


Figure 19: Nonlinear models of temperature-dependent blood perfusion for muscle tissue, fat tissue, and tumor.

The basis model used in our simulation is the instationary bio–heat–transfer–equation proposed by PENNES [78].

$$\rho c \frac{\partial T}{\partial t} = \nabla(\kappa \nabla T) - c_b W (T - T_b) + Q_e,$$

where ρ is the density of tissue, c and c_b are specific heat of tissue and blood, κ is the thermal conductivity of tissue; T_b is the blood temperature; W is the mass flow rate of blood per unit volume of tissue. The power Q_e deposited by an electric field E in a tissue with electric conductivity σ given by

$$Q_e = \frac{1}{2}\sigma |E|^2.$$

Besides the differential equation boundary condtions determine the temperature distribution. The heat exchange between body and water bolus can be described by the flux condition

$$\kappa \frac{\partial T}{\partial n} = \beta (T_{bol} - T)$$

where T_{bol} is the bolus temperature and β is the heat transfer coefficient. No heat loss is assumed in remaining regions. We use for our simulations $\beta = 45$ $[W/m^2/^oC]$ and $T_{bol} = 25$ $[^oC]$.

Studies that predict temperatures in tissue models usually assume a constant-rate blood perfusion within each tissue. However, several experiments have shown that the response of vasculature in tissues to heat stress is strongly temperature-dependent (SONG ET AL., 1984). So the main intention of this work was to develop new nonlinear heat transfer models in order to reflect this important observation.

We assume a temperature dependence of blood perfusion W in the tissues muscle, fat, and tumor (compare Figure 19):

$$W_{muscle} = \begin{cases} 0.45 + 3.55 \exp\left(-\frac{(T-45.0)^2}{12.0}\right), & T \leq 45.0 \\ 4.00, & T > 45.0 \end{cases}$$

$$W_{fat} = \begin{cases} 0.36 + 0.36 \exp\left(-\frac{(T-45.0)^2}{12.0}\right), & T \leq 45.0 \\ 0.72, & T > 45.0 \end{cases}$$

$$W_{tumor} = \begin{cases} 0.833 & T < 37.0 \\ 0.833 - (T-37.0)^{4.8}/5438, & 37.0 \leq T \leq 42.0 \\ 0.416, & T > 42.0 \end{cases}$$

There are significant qualitative differences between the temperature distribution predicted by the standard (linear) and the nonlinear heat transfer model. Generally, the self-regulation of healthy tissue is better reflected by the nonlinear model which is now used in practical computations.

See [29], [30], and [59] for more details.

3.7 Tumour Invasion

A tumour arises from a single cell which is genetically disturbed. A local tumour is growing but it doesn't grow arbitrarily. In fact, we get a balance between new and dying cells because the tumour cannot be sufficiently supplied with oxygen and other nutrients. This equilibrium can take months or years. However, some tumours are able to produce proteins initializing the growth of blood vessels. If these proteins come close to existing blood vessels then new blood vessels are generated growing in direction of the tumour and

penetrating it. This improves the supplement of the tumour with oxygen and nutrients. The tumour strengthens its growth.

The tumour starts to produce metastasis when meeting some blood vessels. We distinguish the following steps: 1. Tumour cells are seperated from the original tumour. 2. The seperated cells penetrate the neighbouring tissue and enter the circulation of blood and lymph being transported to other locations in the body. 3. Somewhere the tumour cells leave the circulation and penetrate healthy tissue. There they generate new tumours called metastasis.

Each of these steps is influenced by different factors, e.g., the presence of special proteins. We consider a taxis-diffusion-reaction model of tumour cell invasion described in ANDERSON ET AL.[4] and used in the considerations of GERISCH and VERWER [35]. It describes the cell propagation and the process of penetration the neighbouring tissue. The interaction between extracellular matrix (ECM) and matrix degradative enzymes (MDE) is responsible for that. ECM integrates regular cells into tissue. ECM is reduced by MDE when healing a wound or developing an embryo. The increase of metastasis is also determined by the reduction of ECM. The MDE necessary for that can be produced by the tumour itself.

The model describes the behaviour of three components: the density n of the tumour cells, the density c_1 of ECM, and the concentration c_2 of MDE.

We use the following equations in two space dimensions:

$$\begin{aligned}
\frac{\partial n}{\partial t} &= \nabla \cdot (\varepsilon \nabla n) - \nabla \cdot (n\gamma \nabla c_1), \\
\frac{\partial c_1}{\partial t} &= -\eta c_1 c_2, \\
\frac{\partial c_2}{\partial t} &= \nabla \cdot (d_2 \nabla c_2) + \alpha n - \beta c_2.
\end{aligned}$$

with the constant parameters ε, γ, η, d_2, α, and β. We have boundary conditions of NEUMANN type for the components n and c_2. There is no need for boundary values for c_1. For the initial values we refer to the publication of ANDERSON ET AL. [4].

In particular, we can simulate the fragmentation of an initial cell mass, see Figure 20.

We refer to the diploma thesis of SCHUMANN [84] for more details.

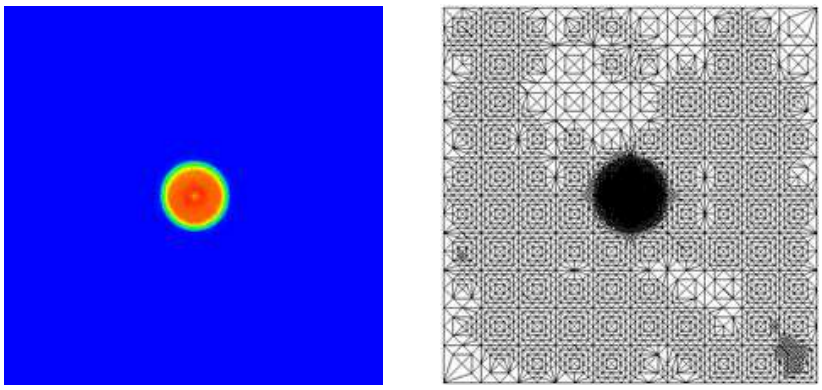


Figure 20: Fragmentation of initial cell mass and corresponding FEM mesh.

3.8 Linear Elastic Modelling of the Human Mandible

A detailed understanding of the mechanical behaviour of the human mandible has been an object of medical and biomedical research for a long time. Better knowledge of the stress and strain distribution, e.g., concerning standard biting situations, allows an advanced evaluation of the requirements for improved osteosynthesis or implant techniques. In the field of biomechanics, FEM-Simulation has become a well appreciated research tool for the prediction of regional stresses. The scope of this project is to demonstrate the impact of adaptive finite element techniques in the field of biomechanical simulation. Regarding to their reliability, computationally efficient adaptive procedures are nowadays entering into real-life applications and starting to become a standard feature of modern simulation tools. Because of its complex geometry and the complicated muscular interplay of the masticatory system, modelling and simulaion of the human mandible are challenging applications.

In general, simulation in structural mechanics requires at least a representation of the specimen's geometry, an appropriate material description, and a definition of the loading case. In our field, the inherent material is bone tissue, which is one of the strongest and stiffest tissues of the body. Bone itself is a highly complex composite material. Its mechanical properties are anisotropic, heterogeneous, and visco-elastic. At a macroscopic scale, two different kinds of bone can be distinguished in the mandible: cortical or compact bone is present in the outer part of bones, while trabecular, cancellous or spongious bone is situated at the inner, see Figure 21.

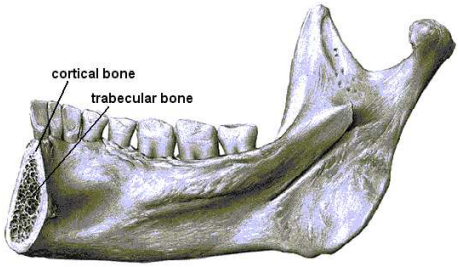


Figure 21: The bone structure of the human mandible.

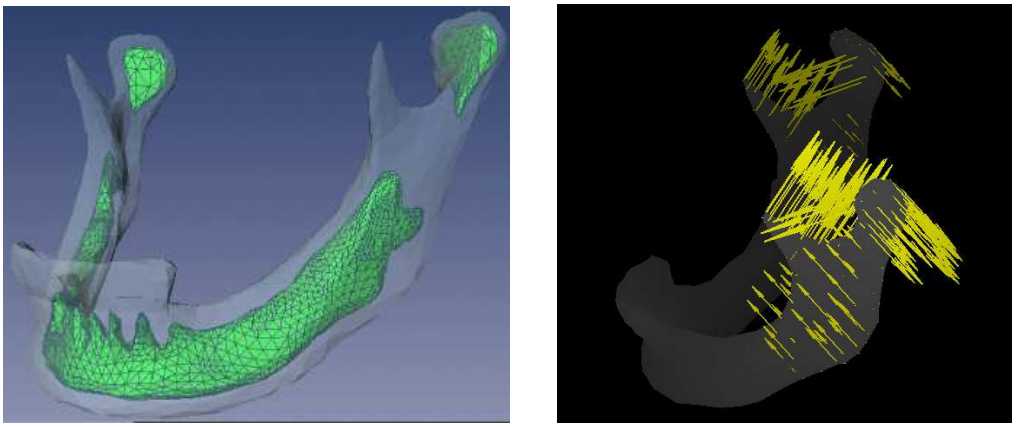


Figure 22: The separation of cortical and cancellous bone as realized in the simulations (left). Loading case (right).

Computed tomography data (CT) are the base of the jawbone simulation. By this, the individual geometry is quite well reproduced, also the separation of cortical and trabecular bone, see Figure 22 (left). In this project, we restrict ourselves to an isotropic, but inhomogeneous linear elastic material law due to Lamé. Figure 22 (right) gives a view on a loading case, here the lateral biting situation.

If we denote the displacement vector by $u = (u_1, u_2, u_3)$ and the strain tensor by $\mathcal{E}$ then we can write

$$-2\mu \ \operatorname{div} \mathcal{E}(u) - \lambda \ \operatorname{grad}(\operatorname{div} u) = f$$

supplied by appropriate boundary values.

This equation can be transformed to

$$-\nabla \cdot (\mathbf{D} \nabla u) = f$$

where

$$\mathbf{D} = \begin{pmatrix} \begin{pmatrix} \lambda+2\mu & 0 & 0 \\ 0 & \mu & 0 \\ 0 & 0 & \mu \end{pmatrix} & \begin{pmatrix} 0 & \lambda & 0 \\ \mu & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & \lambda \\ 0 & 0 & 0 \\ \mu & 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & \mu & 0 \\ \lambda & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} & \begin{pmatrix} \mu & 0 & 0 \\ 0 & \lambda+2\mu & 0 \\ 0 & 0 & \mu \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & \lambda \\ 0 & \mu & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 & \mu \\ 0 & 0 & 0 \\ \lambda & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & \mu \\ 0 & \lambda & 0 \end{pmatrix} & \begin{pmatrix} \mu & 0 & 0 \\ 0 & \mu & 0 \\ 0 & 0 & \lambda+2\mu \end{pmatrix} \end{pmatrix}$$

.

We use the relationships

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \qquad \mu = \frac{E}{2(1+\nu)}$$

between the elastic coefficients λ, μ, E (*Young's modulus*), and ν (*Poisson's ratio*).

For pre- and postprocessing including volumetric grid generation we use the visualization package AMIRA [91]. After semiautomatic segmentation of the CT data, the algorithm for generation of non-manifold surfaces gives a quite satisfying reconstruction of the individual geometry. After some coarsening, we get a mesh (see Figure 23) which can be used as initial grid (11,395 tetrahedra resp. 2,632 points) in the adaptive discretization process.

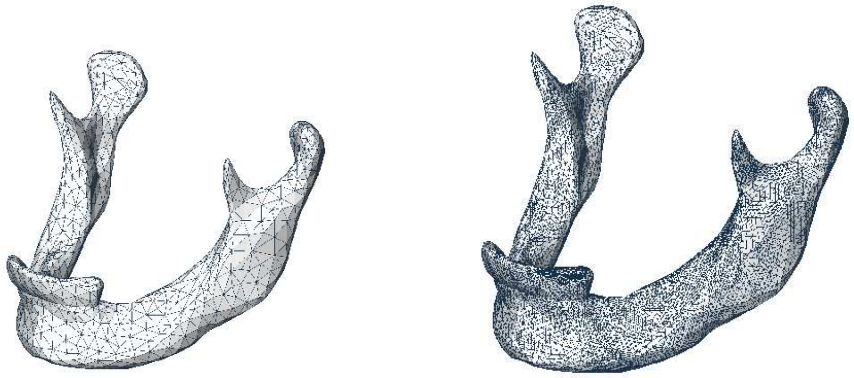


Figure 23: Initial grid for the adaptive finite element method (left). Grid after three steps of adaptive refinement (right).

According to the required accuracy, the volumetric grid is adaptively refined from level 0 up to level 3. The finest grid is shown in Figure 23. In Figure 24, we present the maximum absolute values of deformation (occuring in the processus coronoidus) for both adaptive and uniform refinement of the grid. The comparison makes it comprehensible that the adaptive method is much more efficient if high accuracy is required.

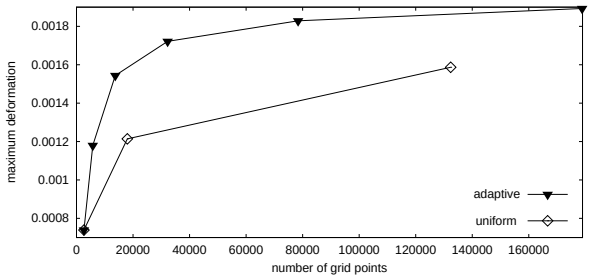


Figure 24: Adaptive versus uniform mesh refinement: comparative maximum deformation results.

In the following, the results after adaptive calculation of a common postpro-

cessing variable, the von Mises equivalent stress, is discussed. It represents the distortional part of the strain energy density for isotropic materials. Figure 25shows a comparison between the results from a calculation on the coarse (level 0) grid versus that from the finest (level 3) grid. In both calculations, the stress maximum occurs around the processus coronoidus of the working side whereas the condyles are nearly at the minimum level in spite of the conylar reaction forces. On the level 0, the observed stress maximum of 2.81 MPa is about 65 % less than the maximum stress of 4.34 MPa achieved on the level 4 calculation.

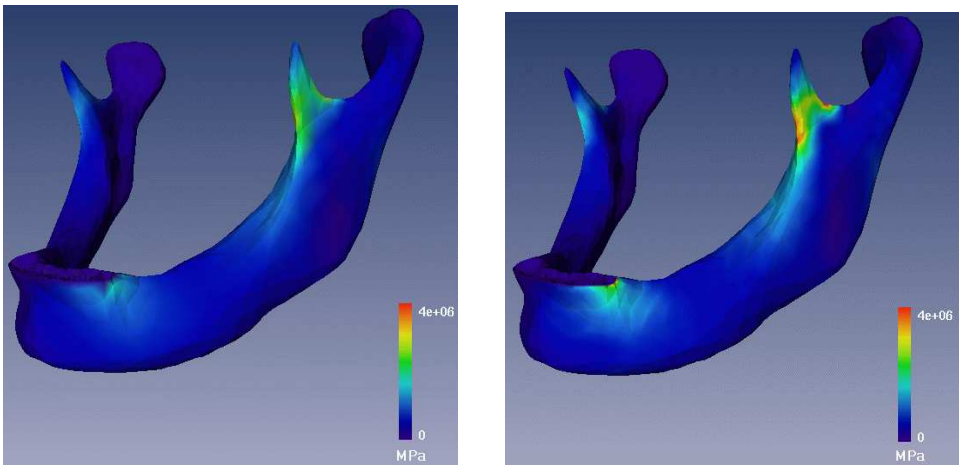


Figure 25: Von Mises equivalent stress on level 0, maximum: 2.81 MPa (left).Von Mises equivalent stress on level 4, maximum: 4.34 MPa (right).

3.9 Porous Media

Brine Transport in Porous Media. High–level radioactive waste is often disposed in salt domes. The safety assessment of such a repository requires the study of groundwater flow enriched with salt. The observed salt concentration can be very high with respect to seawater, leading to sharp and moving freshwater–saltwater fronts. In such a situation, the basic equations of groundwater flow and solute transport have to be modified (Hassanizadeh and Leijnse [41]). We use the physical model proposed by Trompert, Verwer, and Blom [89] for a non–isothermal, single–phase, two–component saturated flow. It consists of the brine flow equation, the

salt transport equation, and the temperature equation and reads

$$n\rho\,(\beta\,\partial_t p + \gamma\,\partial_t w + \alpha\,\partial_t T) + \nabla\cdot(\rho\mathbf{q}) = 0\,, \tag{19}$$

$$n\rho\,\partial_t w + \rho\mathbf{q}\cdot\nabla w + \nabla\cdot(\rho J^w) = 0\,, \tag{20}$$

$$(nc\rho + (1-n)\rho^s c^s)\partial_t T + \rho\,c\,\mathbf{q}\cdot\nabla T + \nabla\cdot J^T = 0 \tag{21}$$

supplemented with the state equations for the density ρ and the viscosity μ of the fluid

$$\begin{aligned}\rho &= \rho_0 \exp\left(\alpha(T-T_0) + \beta(p-p_0) + \gamma w\right)\,,\\ \mu &= \mu_0\,(1.0 + 1.85w - 4.0w^2)\,.\end{aligned}$$

Here, the pressure p, the salt mass fraction w, and the temperature T are the independent variables, which form a coupled system of nonlinear parabolic equations. n is the porosity, ρ^s the density of the solid, c^s the heat capacity of the solid, and ρ_0 the freshwater density.

The Darcy velocity $\mathbf{q}$ of the fluid is defined as

$$\mathbf{q} = -\frac{K}{\mu}\,(\nabla p - \rho\mathbf{g})$$

where K is the permeability tensor of the porous medium, which is supposed to be of the form $K = \mathrm{diag}(k)$, and $\mathbf{g}$ is the acceleration of gravity vector. The salt dispersion flux vector J^w and the heat flux vector J^T are defined as

$$\begin{aligned}J^w &= -\left((nd_m + \alpha_T|\mathbf{q}|)\,I + \frac{\alpha_L - \alpha_T}{|\mathbf{q}|}\mathbf{q}\mathbf{q}^T\right)\nabla w\,,\\ J^T &= -\left((\kappa + \lambda_T|\mathbf{q}|)\,I + \frac{\lambda_L - \lambda_T}{|\mathbf{q}|}\mathbf{q}\mathbf{q}^T\right)\nabla T\,,\end{aligned}$$

where $|\mathbf{q}| = \sqrt{\mathbf{q}^T\mathbf{q}}$. α_L, α_T denote the transversal resp. longitudinal dispersity, and λ_L, λ_T the transversal resp. longitudinal heat conductivity.

Writing the system of the three balance equations (19)–(21) in the form (8), we find for the 3×3 matrix B

$$B(p,w,T) = \begin{pmatrix} n\rho\beta & n\rho\gamma & n\rho\alpha \\ 0 & n\rho & 0 \\ 0 & 0 & nc\rho + (1-n)\rho^s c^s \end{pmatrix}.$$

Since the compressibility coefficient β is very small, the matrix B is nearly singular and, as known (Hairer and Wanner [40], VI.6), linearly implicit

time integrators suitable for differential algebraic systems of index 1 do not give precise results. This is mainly due to the fact that for $\beta=0$ the matrix B becomes singular and additional consistency conditions have to be satisfied to avoid order reduction. We have applied the Rosenbrock solver ROWDAIND2 [71], which handles both situations, $\beta=0$ and $\beta\neq 0$.

An additional feature of the model is that the salt transport equation (20) is usually dominated by the advection term. In practice, global Peclet numbers can range between 10^2 and 10^4, as reported in [89]. On the other hand, the temperature and the flow equation are of standard parabolic type with convection terms of moderate size.

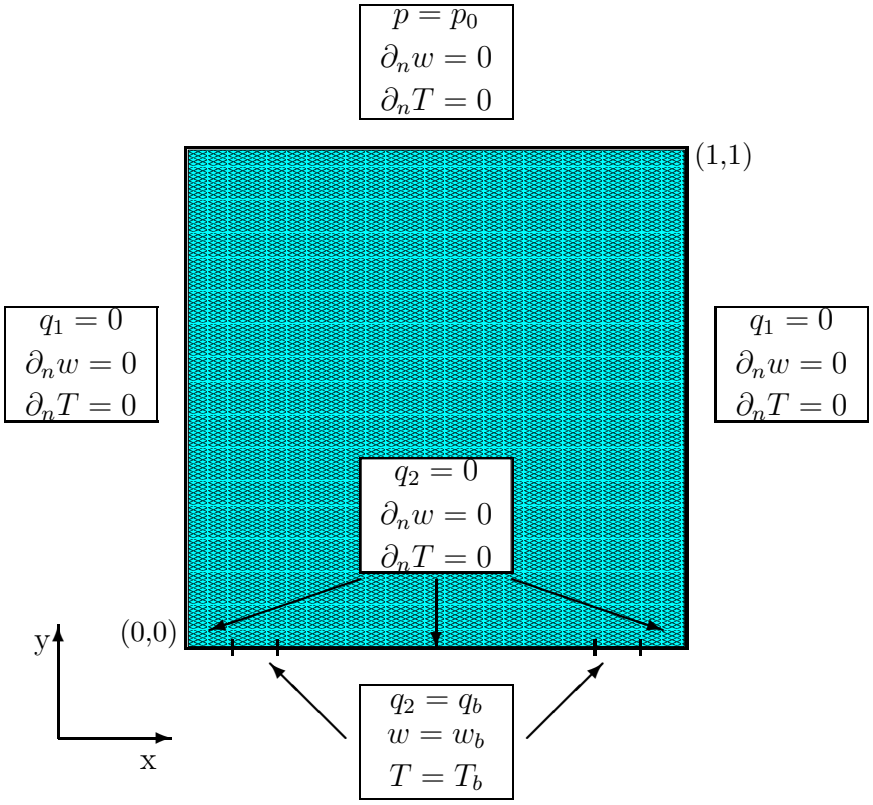


Figure 26: Two–dimensional brine transport. Computational domain and boundary conditions for $t > 0$. The two gates where warm brine is injected are located at $(x, y) : \frac{1}{11} \leq x \leq \frac{2}{11}$, $\frac{9}{11} \leq x \leq \frac{10}{11}$, $y=0$.

Two–dimensional warm brine injection. This problem was taken from [89].

We consider a (very) thin vertical column filled with a porous medium. This justifies the use of a two–dimensional flow domain $\Omega = \{(x, y) : 0 < x, y < 1\}$ representing a vertical cross–section. The acceleration of gravity vector points downward and takes the form $\mathbf{g} = (0, -g)^T$, where the gravity constant g is set to 9.81. The initial values at $t = 0$ are

$$p(x, y, 0) = p_0 + (1 - y)\rho_0 g, \quad w(x, y, 0) = 0\,, \text{ and } T(x, y, 0) = T_0\,.$$

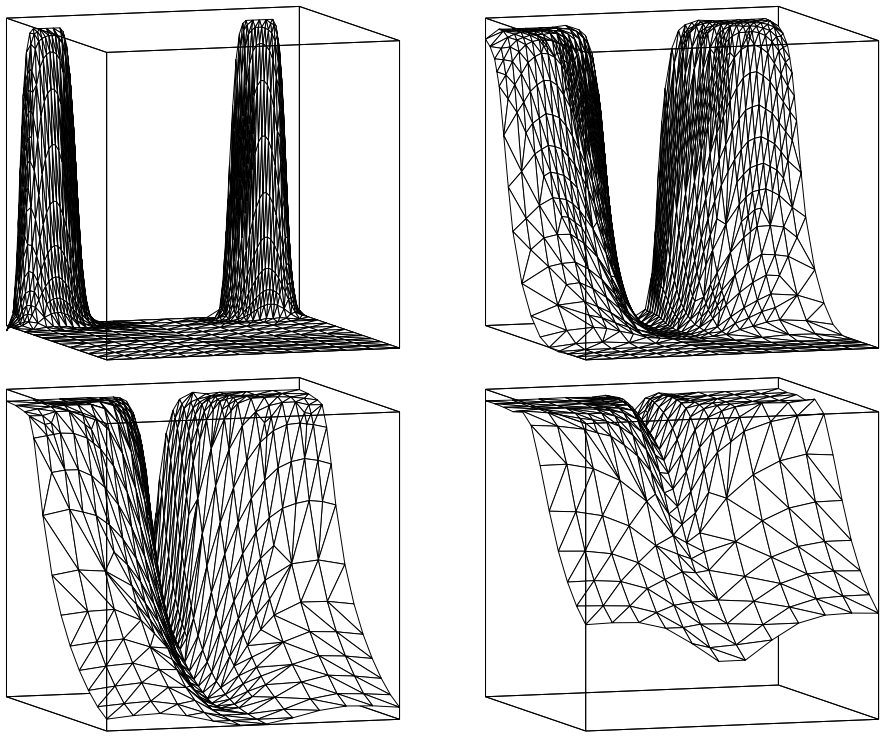


Figure 27: Two–dimensional brine transport. Distribution of salt concentration at $t = 500, 5000, 10000$, and 20000 with corresponding spatial grids.

The boundary conditions are described in Figure 26. We set $w_b = 0.25$, $T_b = 292.0$, and $q_b = 10^{-4}$. The remaining parameters used in the model are given in [63].

Warm brine is injected through two gates at the bottom. This gives rise to sharp fronts between salt and fresh water, which have to be resolved with fine meshes in the neighbourhood of the gates, see Figure 27. Later the solutions smooth out with time until the porous medium is filled completely with brine.

Our computational results are comparable to those obtained in [89] with a method of lines approach coupled with a local uniform grid refinement. In Figure 28 we show the time steps and the degrees of freedom chosen by the KARDOS solver to integrate over $t \in [0, 2 \cdot 10^4]$. The curves nicely reflect the high dynamics at the beginning in both, time and space, while larger time steps and coarser grids are selected in the final part of the simulation.

Three–dimensional pollution with salt water. Here, we consider Problem III of [9] and simulate a salt pollution of fresh water flowing from left to right through a tank $\Omega = \{(x, y, z) : 0 \leq x \leq 2.5, 0 \leq y \leq 0.5, 0 \leq z \leq 1.0\}$ filled with a porous medium. The flow is supposed to be isothermal ($\alpha = 0$) and incompressible ($\beta = 0$). Hence, the problem consists now of two PDEs with a singular 2×2 matrix $B(p, w)$ multiplying the vector of temporal derivatives. The acceleration of gravity vector takes the form $\mathbf{g} = (0, 0, -g)^T$.

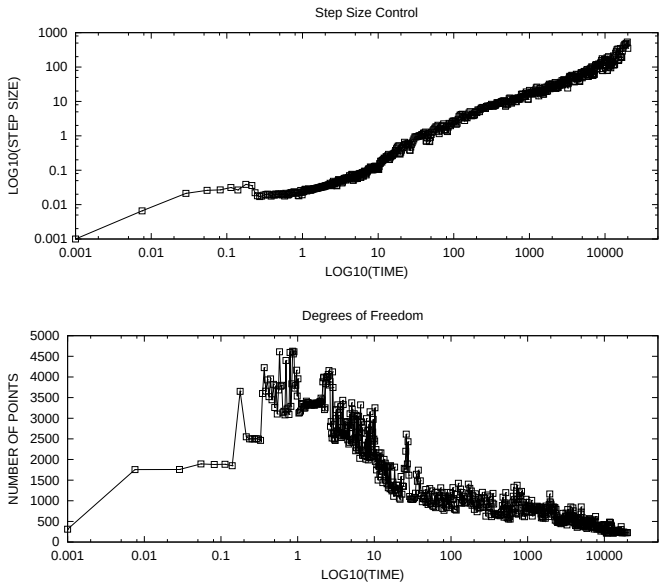


Figure 28: Two–dimensional brine transport. Evolution of time steps and number of spatial discretization points for $TOL_t = TOL_x = 0.005$.

The brine having a salt mass fraction $w_b = 0.0935$ is injected through a small slit $S = \{(x, y, 1) : 0.375 \leq x \leq 0.4375,\ 0.25 \leq y \leq 0.3125\}$ at the top of the tank. We note that the slit chosen here differs slightly from that used in [9].

The initial values at $t=0$ are taken as

$$p(x,y,z,0) = p_0 + (0.03 - 0.012x + 1.0 - z)\rho_0 g, \quad w(x,y,z,0) = 0\,,$$

and the boundary conditions are

$$\begin{aligned}
&p = p(x,y,z,0)\,,\ w = 0\,, && \text{on } x = 0\,,\\
&p = p(x,y,z,0)\,,\ \partial_n w = 0\,, && \text{on } x = 2.5\,,\\
&q_2 = 0\,,\ \partial_n w = 0\,, && \text{on } y = 0 \text{ and } y = 1\,,\\
&q_3 = 0\,,\ \partial_n w = 0\,, && \text{on } z = 0 \text{ and } \{z = 1\} \setminus S\,,\\
&\rho q_3 = -4.95 \cdot 10^{-2}\,,\ w = w_b = 0.0935\,, && \text{on } S\,.
\end{aligned}$$

The parameters used in the three–dimensional simulation are also given in [63]. Additionally, the state equation for the viscosity of the fluid is modified to

$$\mu = \mu_0(1.0 + 1.85w - 4.1w^2 + 44.5w^3)\,.$$

In Figure 29 we show the distribution of the salt concentration in the plane $y=0.28125$ after two and four hours. The pollutant is slowly transported by the flow while sinking to the bottom of the tank. The steepness of the solution is higher in the back of the pollution front, which causes fine meshes in this region. Despite the dominating convection terms no wiggles are visible, especially at the inlet. An interesting observation is the unexpected drift in front of the solution – a phenomenon which was also observed by Blom and Verwer [9].

Adiabatic Flow of a Homogeneous Gas. The Figure 30 shows a typical two-dimensional Barenblatt solution.

3.10 Sorption Technology

The common application for active thermal solar systems is the supply of buildings with hot water. The main obstacle for heating purposes is the gap between summer, when the major amount of energy can be collected, and winter, when the heating energy is required. This demands the application of a seasonal energy storage with small heat loss over a long time and well defined load and unload behavior. For that purpose a new technique has been developed, which is based on the heat of adsorption resp. desorption of stream on silicagel (Mittelbach and Henning, [76]). The load of water on the gel is a function of H_2O partial pressure and temperature.

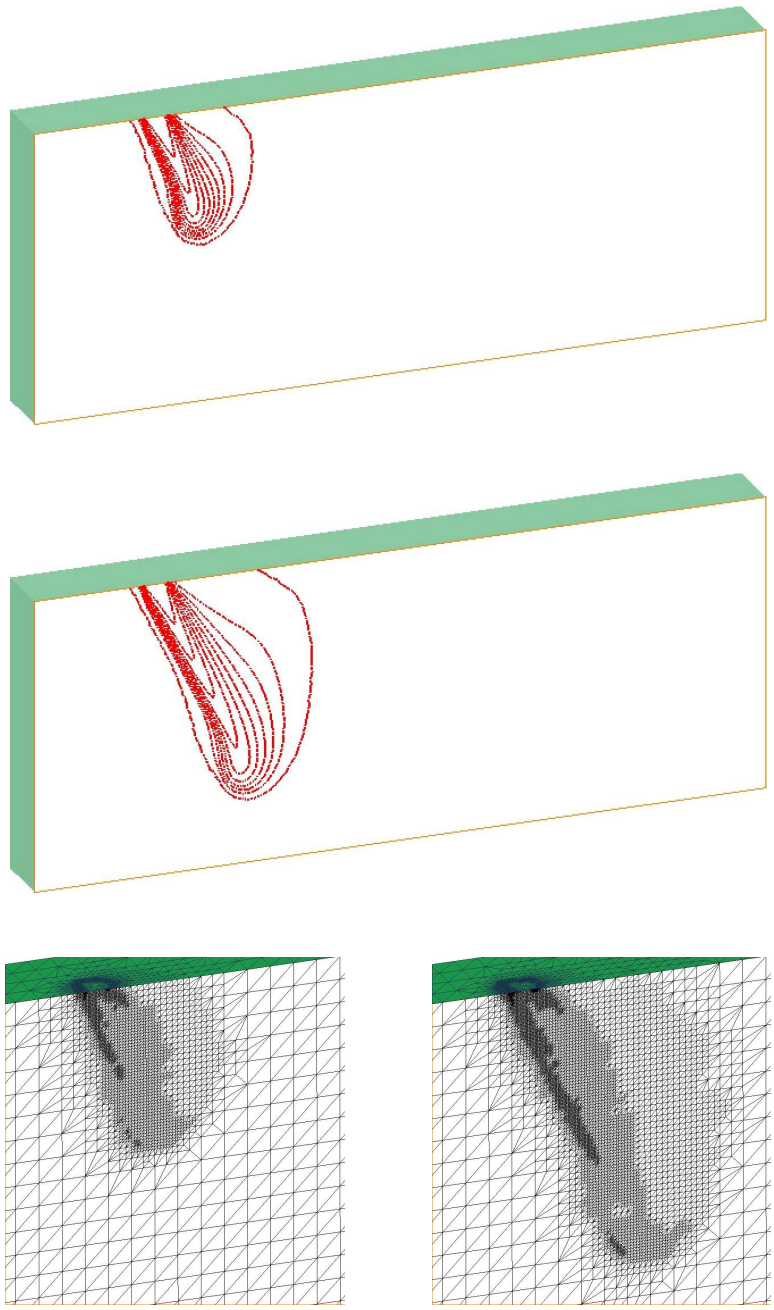


Figure 29: Three–dimensional brine transport. Level lines of the salt concentration $w = 0.0, 0.01, \ldots, 0.09$, in the plane $y = 0.28125$ after two hours (top) and four hours (middle), and corresponding spatial grids (bottom) in the neighborhood of the inlet.

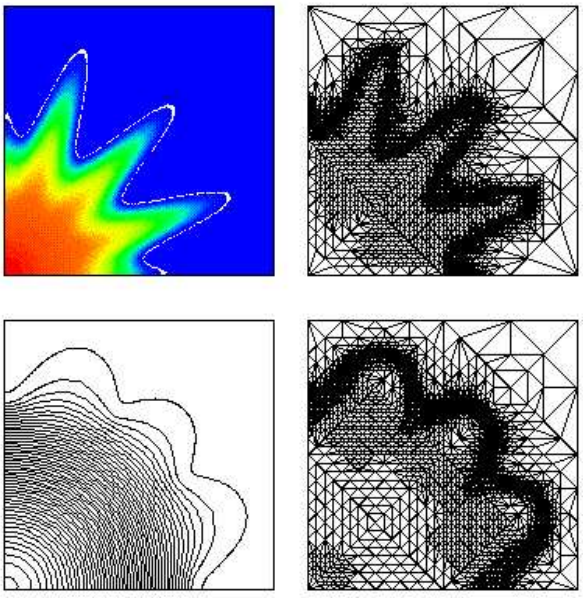


Figure 30: Initial (top) and final (bottom) solution at t=0.05.

For understanding and predicting the operation behavior of the adsorption storage the temperature field and vapor density field in the bed have to be calculated. First computations with KARDOS can be found in the report [85].

3.11 Combinable Catalytic Reactor System

In this project we develop robust and efficient numerical software for the simulation of a special type of a catalytic fixed-bed reactor, see Figure 31. Our project partners are constructing a set of combinable catalytic reactors of different size, appropriate connecting devices and measurement tools. Combining these moduls in various ways allows to investigate a chemical process of interest under quite different physical conditions. The final aim of the cooperation is to provide a software tool which allows an a priori simulation of a planned combination.

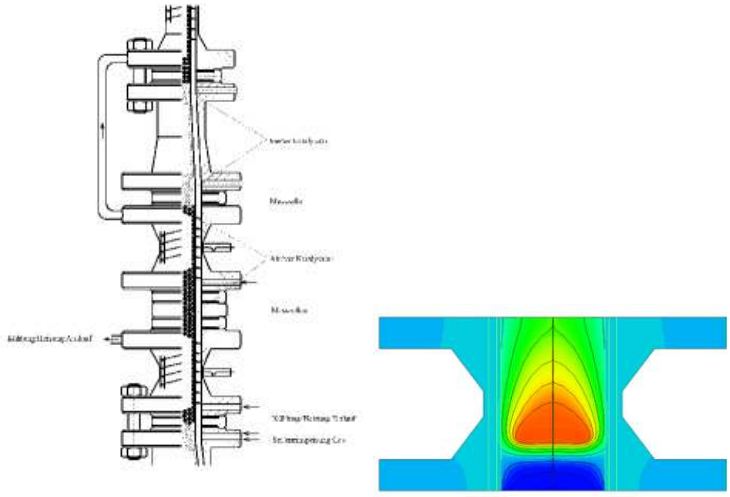


Figure 31: Combinable catalytic reactor (left) and stationary temperature distribution (right).

3.12 Incompressible Flows

The aim of this project is to extend the KARDOS-software to incompressible flow problems. It includes thermally coupled flows satisfying the thermodynamic assumptions for the Boussinesq approximation. The equations governing this flow are

$$\begin{aligned} \rho_0[\partial_t v + (v \cdot \nabla)v] - \mu\nabla^2 v + \nabla p &= \rho_0 g[1 - \beta(T - T_0)] + F_v, \\ \nabla \cdot v &= 0, \\ \rho_0 c_p[\partial_t T + (v \cdot \nabla)T] - \kappa\nabla^2 T &= F_T, \end{aligned} \tag{22}$$

where v describes the velocity field, p is the pressure, ρ_0 is the (constant) density of the fluid, μ is the dynamical viscosity, T is the temperature, κ is the thermal conductivity, g is the gravitational acceleration, c_p is the specific heat at constant pressure, F_v and F_T are force terms. The parameter β is the volume expansion coefficient and T_0 refers to a reference temperature state. A Galerkin/least-squares method is applied in space to prevent numerical instabilities forced by advection-dominated terms. First results obtained for various benchmark problems are very promising, showing that the adaptive algorithm implemented in KARDOS can also be a useful means to handle CFD problems, see [57].

The development of methods for CFD is not yet finished in KARDOS.

Therefore, we decided to offer the official version of KARDOS without CFD features.

Laminar Flow Around a Cylinder:

The flow past a cylinder is a widely solved problem. To make our computations comparable with the results of a benchmark [83], we skip the temperature and solve the conservation equations of mass and momentum. The fluid density is defined as $\rho_0 = 1.0kg/m^3$, and the dynamic viscosity is $\mu = 0.001m^2/s$. No force term F_v is considered. The computational domain has length $L = 2.2m$ and height $H = 0.41m$. The midpoint of the cylinder with diameter $D = 0.1m$ is placed at $(0.2m, 0.2m)$. The inflow condition at the left boundary is

$$v_x(0, y, t) = 4Vy(H - y)/H^2, \quad v_y = 0\,,$$

with a mean velocity $V = 0.3m/s$ yielding a Reynolds number $Re = 20$. We further use non-flux conditions at the right outflow boundary, and $v_x = v_y = 0$ otherwise. The flow becomes steady and two unsymmetric eddies develop behind the cylinder. We start with a very coarse approximation of the given geometry (81 points) to test our automatic mesh controlling. The resulting fine grid at the steady state contains 2785 points. The drag and lift coefficients as well as the pressure difference $\triangle p = p(0.15m, 0.2m) - p(0.25m, 0.2m)$ are in good agreement with the results given in [83].

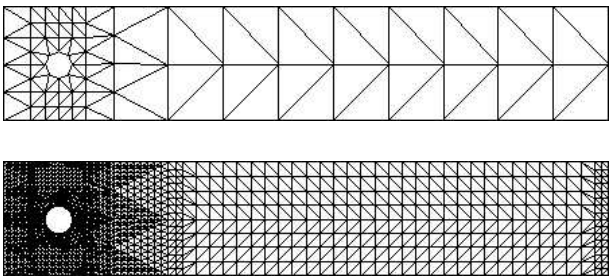


Figure 32: Initial and adapted spatial grid.

Thermo-Convective Poiseuille Flow: Introducing suitable reference values, the system (22) may be written for the so–called forced convection problem in dimensionless form as follows

$$\begin{aligned} \partial_t v + (v \cdot \nabla)v - \tfrac{1}{Re}\nabla^2 v + \nabla p &= -\tfrac{1}{Fr} T\hat{g}\,, \\ \nabla \cdot v &= 0\,, \\ \partial_t T + (v \cdot \nabla)T - \tfrac{1}{Pe}\nabla^2 T &= 0\,, \end{aligned}$$

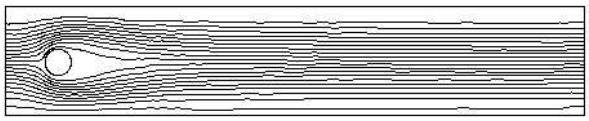


Figure 33: Streamlines.

where source terms have been omitted. Fr is the Froude number and the vector $\hat{g}$ in the momentum equation denotes now the normalized gravity acceleration vector. We consider a two–dimensional laminar flow in a horizontal channel $\Omega = [0, 10] \times [0, 1]$ suddenly heated from below with constant temperature $T = 1.0$. At the opposite wall we choose $T = 0.0$ and non–flux conditions for the temperature at the inlet and outlet. The boundary conditions for the velocity field are taken from the previous problem, whereas a parabolic inlet profile is prescribed by

$$v_x(0, y, t) = 6y(1 - y), \quad v_y = 0\,.$$

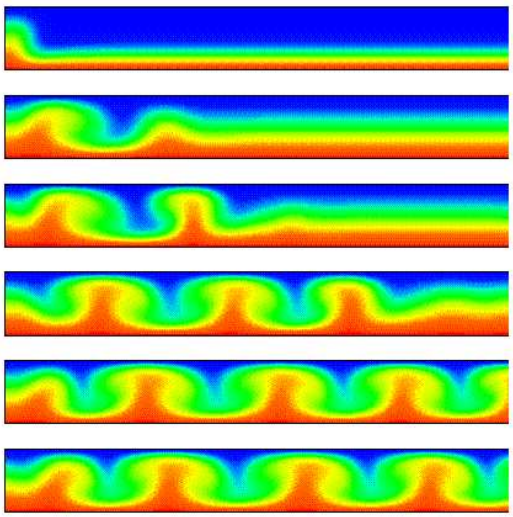


Figure 34: Evolution of temperature.

The dimensionless parameters have been taken as $Re = 10$, $Fr = 1/150$ and $Pe = 40/9$. The same setting was studied in [16]. Travelling transverse waves can be observed (see Figures 34, 35). We plot also the transient evolution of the temperature at the central point $(5.0, 0.5)$. Comparing our curve with

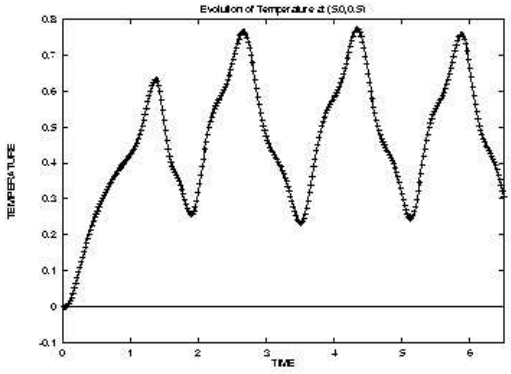


Figure 35: Evolution of temperature at central point.

that given in [16], we observe a smoother function due to the higher accuracy provided by the devised adaptive approach.

4 Installation Guidelines

Though the underlying algorithms of KARDOS for one, two, or three space dimensions agree on many points, we provide different programs for each case. That is caused by historical reasons and by the fact that the C language doesn't offer comfortable features to organise all in one code. Normally, the program will be offered as a compressed tar-file, e.g., kardos.tar.Z. For installation you have to perform the following steps independent from the space dimension your version is made for.

- `uncompress kardos.tar.Z`
- `tar -xf kardos.tar`
- `cd kardos`
  `ls`

  The moduls of KARDOS are assembled in different directories:

  - `kardos`: contains main program and some files with specific definitions)
  - `kaskSource`: grid and node management
  - `timeSource`: time integration routines
  - `mgSource`: routines for 1D and 2D graphics, graphic functions are based on X11
  - `problems`: directories with exemplary problem specifications
- `cd kardos/mgSource`
  `configure`
  `make`
- `cd ../kaskSource`
  `make`
- `cd ../timeSource`
  `make`
- `cd ../kardos`
  `make`

- ```
  cd ../problems/user
  setLink


  kardos
  do user1.ksk
  ```

Note: The makefiles are prepared for *SUN/Solaris* systems. For installation on another Unix platform you have to modify the makefile in ../kardos, in particular you have to choose the correct libraries and to describe their locations in the system folder. We give an example for a *Linux* system in the makefile "Makefile.Linux".

If there is no Fortran compiler called "f77" on your machine you have to call the available one in the makefile "Makefile" in the directory ../kaskSource.

Furthermore you should select the correct clock in the file "portab.c". On *Sun Solaris* systems the function gethrvtime() is used, which is not available on *Linux* machines. *Linux* offers the function "clock" instead of that.

If there occur any problems when installing KARDOS,
please contact
erdmann@zib.de or roitzsch@zib.de.

Example user1 (see user1.ksk) computes the solution of a simple transient Poisson equation (compare Section 5).
  ```

5 Define a New Problem

In this section we want to give all the information a user needs to check in his problem of type (1).

That means the user has to define:

- the coefficients B, D, and the right-hand side F which may depend on the coordinates, the time, the solution, and its gradient,
- the initial values at the starting time,
- the boundary values which may be of type DIRICHLET, CAUCHY, or NEUMANN,
- the geometry $\overline{\Omega}$,
- setting of parameters and the input files configuring the program for the new problem.

5.1 Coefficient Functions

The coefficients B, D, and the right-hand side F from equation (1) have always to be defined by functions in the source code. We recommend to do it in the file `user.c` that you can find in the directory `kardos/problems/user`. Note that you always have to recompile the code if you have changed something in the source code. Compare Section 4 for details.

In that directory we already prepared two examples which we will explain in the following. But first we shall give a description of the general case.

We can write equation (1) as set of n equations

$$\sum_{j=1}^{n} B_{ij} \frac{\partial u_j(x,t)}{\partial t} - \sum_{j=1}^{n} D_{ij} u_j \quad = \quad F_i(x,t,u,\nabla u), \quad i = 1(1)n. \tag{23}$$

That means we have to define the elements of the matrices $B = (B_{ij})$, $D = (D_{ij})$, and the components of the vector $F = (F_i)$, $i, j = 1(1)n$. Note that the elements D_{ij} are defined as follows

$$D_{ij} u_j \quad = \quad \nabla \cdot (\mathbf{P}^{ij} \nabla u_j) - \mathbf{Q}^{ij} \cdot \nabla u_j \tag{24}$$

with

$$\mathbf{P}^{ij} = (p_{kl}^{ij}), \quad k, l = 1(1)s,$$

and

$$\mathbf{Q}^{ij} = (Q_k^{ij}), \quad k = 1(1)s,$$

where $s \in \{1, 2, 3\}$ is the dimension of the space. $\mathbf{P}$ represents the diffusion term. The convection term $\mathbf{Q}$ can alternatively be considered as part of the right-hand side F.

For example, $\mathbf{P}^{ij}$ and $\mathbf{Q}^{ij}$ have in two space dimensions the shape

$$\mathbf{P}^{ij} = \begin{pmatrix} P_{xx}^{ij} & P_{xy}^{ij} \\ P_{yx}^{ij} & P_{yy}^{ij} \end{pmatrix},$$

resp.

$$\mathbf{Q}^{ij} = \begin{pmatrix} \beta_x^{ij} \\ \beta_y^{ij} \end{pmatrix}.$$

Using these definitions we also can write system (23) in the form

$$\begin{pmatrix} B_{11} & \cdots & B_{1n} \\ \vdots & \ddots & \vdots \\ B_{n1} & \cdots & B_{nn} \end{pmatrix} \begin{pmatrix} \frac{\partial u_1}{\partial t} \\ \vdots \\ \frac{\partial u_n}{\partial t} \end{pmatrix} - \nabla \cdot \left(\begin{pmatrix} P^{11} & \cdots & P^{1n} \\ \vdots & \ddots & \vdots \\ P^{n1} & \cdots & P^{nn} \end{pmatrix} \begin{pmatrix} \nabla u_1 \\ \vdots \\ \nabla u_n \end{pmatrix} \right)$$

$$+ \begin{pmatrix} Q^{11} & \cdots & Q^{1n} \\ \vdots & \ddots & \vdots \\ Q^{n1} & \cdots & Q^{nn} \end{pmatrix} \cdot \begin{pmatrix} \nabla u_1 \\ \vdots \\ \nabla u_n \end{pmatrix} \tag{25}$$

$$= \begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix}$$

To define the elements of B the user has to program two functions as interface. In 2D, we get the following form. In the 3D case you have to add the third coordinate `z` and a variable `uz` for the derivative $\partial u/\partial z$. In the 1D version we have not yet considered the dependencies of the gradient, i.e. the parameter `ux` is not implemented.

```
static int UserParabolic(real x,  real y, int classA, real t,
                         real *u, real *ux, real *uy, real **B)
{
   B[0][0]     = B11;
   B[0][1]     = B12;
     ...
     ...
   B[n-1][n-1] = Bnn;

   return true;
}

static int UserParabolicStruct(int **structB, int **dependsS,
                               int *dependsT, int *dependsU,
                               int *dependsGradU)
{
  structB[0][0]     = F_FILL;   dependsS[0][0] = false;
    ...
    ...
    ...
  structB[n-1][n-1] = F_FILL;   dependsS[n-1][n-1] = false;

  *dependsT = false;
  *dependsU = false;
  *dependsGradU = false;

  return true;
}
```

Here the values of B11,... correspond to the values B_{11},... in the equation (23). In the function `UserParabolicStruct` the user has the possibility to inform on which elements of B are equal 0 and which not. Is an element 0 then the corresponding element in the array `structB` can be set to `F_IGNORE` instead of `F_FILL`. In this case the corresponding value in the function `UserParabolic` must not be defined, it is ignored in all computations. A value `true` for an element in the matrix `dependsS` indicates that the corresponding element in B depends on the coordinates. If the value is `false` the element of B will be assumed to be constant. The variables `dependsT`, `dependsU`, and `dependsGradU` show if the values B, the solution, or the gradient of the solution depend on time, respectively. (Pay attention: these

three variables are not yet implemented in the 1D version!)

Note, that we start the numbering with index 0 instead of 1 as we did above. That's the difference between mathematical description and C programming. This convention is used also in all the other interface functions.

Analogously we can describe the interface for the matrices $\mathbf{P}^{ij}$ and the vectors $\mathbf{Q}^{ij}$. For the two-dimensional space it reads

```
static int UserLaplace(real x, real y, int classA, real t,
                       real *u, real *ux, real *uy,
                       real **matXX, real **matXY,
                       real **matYX, real **matYY)
{
  matXX[0][0] = P11_xx;
  matXY[0][0] = P11_xy;
  matYX[0][0] = P11_yx;
  matYY[0][0] = P11_yy;

  matXX[0][1] = P12_xx;
  matXY[0][1] = P12_xy;
  matYX[0][1] = P12_yx;
  matYY[0][1] = P12_yy;

    ...

  matXX[n-1][n-1] = Pnn_xx;
  matXY[n-1][n-1] = Pnn_xy;
  matYX[n-1][n-1] = Pnn_yx;
  matYY[n-1][n-1] = Pnn_yy;

  return true;
}

static int UserLaplaceStruct(int **structD, int **dependsS,
                             int *dependsT,
                             int *dependsU, int *dependsGradU)
{
  structD[0][0]     = F_FILL; dependsS[0][0] = false;
  structD[0][1]     = F_FILL; dependsS[0][1] = false;
```

```
    ...

  structD[n-1][n-1] = F_FILL; dependsS[n-1][n-1] = false;

  *dependsT = false;
  *dependsU = false;
  *dependsGradU = false;

  return true;
}

static int UserConvection(real x, real y,
                          int classA, real t, real *u,
                          real **matX, real **matY)
{
  matX[0][0] =  Q11_1;
  matY[0][0] =  Q11_2;

  matX[0][1] =  Q12_1;
  matY[0][1] =  Q12_2;

  ...

  matX[n-1][n-1] = Qnn_1;
  matY[n-1][n-1] = Qnn_2;

  return true;
}


static int UserConvectionStruct(int **structQ, int **dependsS,
                                int *dependsT, int *dependsU)
{
  structQ[0][0]     = F_FILL;     dependsS[0][0] = true;
  structQ[0][1]     = F_FILL;     dependsS[0][1] = true;

  ...

  structQ[n-1][n-1] = F_FILL;     dependsS[n-1][n-1] = true;
  structQ[n-1][n-1] = F_FILL;     dependsS[n-1][n-1] = true;
```

```
  *dependsT = false;
  *dependsU = false;

  return true;
}
```

The use of the parameters `dependsS`, `dependsT`, `dependsU`, and `dependsGradU` is the same as in the definition of the parabolic term B. If a matrix $\mathbf{P}^{ij}$ or a vector $\mathbf{Q}^{ij}$ is zero it must not be mentioned in the functions `UserLaplace` and `UserLaplaceStruct`, resp. `UserConvection` and `UserConvectionStruct`. Alternatively, the user can explicitely assign `F_IGNORE` (default!) to that element of `structD` or `structQ`.

The right-hand side (source term) of the equation is also coded in two functions, here shown for the 2d case.

```
static int UserSource(real x, real y, int classA, real t,
                      real *u, real *ux, real *uy, real *vec)
{
  vec[0]   =  F1;

    ...

  vec[n-1] =  Fn;

  return true;
}


static int UserSourceStruct(int *structF, int *dependsS,
                            int *dependsT, int *dependsU,
                            int *dependsGradU)
{
  structF[0]   = F_FILL;   dependsS[0] = false;

    ...

  structF[n-1] = F_FILL;   dependsS[0] = false;

  *dependsT = false;
```

```
  *dependsU = false;
  *dependsGradU = false;

  return true;
}
```

Dependencies of space, time, solution and its gradient are taken into account in the same manner as in the parabolic or laplacian terms. A component in `structF` is set to `F_IGNORE` if the corresponding component of F vanishes.

If the right-hand side F depends on the solution u, the user may specify the values of the Jacobian matrix. See the following example for n equations in two space dimensions.

```
static int UserJacobian(real x, real y, int classA,
                        real t,  real *u, real *ux,
                        real *uy, real **mat)
{
  mat[0][0]     = dF1_du1;
  mat[0][1]     = dF1_du2;
  ...
  mat[0][n-1]   = dF1_dun;

  ...


  mat[n-1][0]   = dFn_du1;
  mat[n-1][1]   = dFn_du2;
  ...
  mat[n-1][n-1] = dF1_dun;

  return true;
}

static int UserJacobianStruct(int **structJ,
                                    int **dependsXY)
{
  structJ[0][0]     = F_FILL;   dependsXY[0][0]   = true;
  structJ[0][1]     = F_FILL;   dependsXY[0][1]   = true;
  ...
```

```
  structJ[0][n-1]   = F_FILL;   dependsXY[0][n-1] = true;

  ...

  structJ[n-1][0]   = F_FILL;   dependsXY[n-1][0] = true;
  structJ[n-1][1]   = F_FILL;   dependsXY[n-1][1] = true;
  ...
  structJ[n-1][n-1] = F_FILL;   dependsXY[n-1][n-1] = true;

  return true;
}
```

The terms of shape `dFi_duj` stand for $\partial F_i/\partial u_j$. If these Jacobian functions are not provided by the user the program will compute the derivatives numerically.

5.2 Initial and Boundary Values

From equation (1) we can derive the boundary conditions in the following form

$$\begin{aligned} u_i &= \gamma_i, \quad i = 1(1)n, \quad \text{on } \Gamma_D, \\ \textstyle\sum_{j=1}^{n} \mathbf{P_{ij}} \nabla u_j \cdot \mathbf{n} &= \xi_i, \quad i = 1(1)n, \quad \text{on } \Gamma_C, \\ \textstyle\sum_{j=1}^{n} \mathbf{P_{ij}} \nabla u_j \cdot \mathbf{n} &= 0, \quad i = 1(1)n, \quad \text{on } \Gamma_N. \end{aligned} \tag{26}$$

The parts Γ_D, Γ_C, and Γ_N must be specified by the description of the domain, see subsection 5.4. Boundary values of DIRICHLET type (on Γ_D) have to be defined in a function, which is evaluated in a boundary point for each of the n equations.

```
static int UserDirichlet(real x, real y, int classA, real t,
                         real *u, int equation, real *bVal)
{
  switch  (equation)
  {
   case 0:
    bVal[0]   = B1-u[0];
    break;
```

```
   case 1:

   ...

   case n-1:
    bVal[n-1] = Bn-u[n-1];
    break;
  }

  return true;
}
```

Note, that the boundary values must be written in the implicit form $\gamma_i - u_i$. If the DIRICHLET boundary Γ_D is empty, then there is no need to define this function. The values `B1`,...,`Bn` correspond to γ_1,..., γ_n which may depend on the coordinates $x, \ldots,$ the time t, and the solution u.

Boundary values of type NEUMANN must not be specified.

However, for boundary values of type CAUCHY (on Γ_C) the user has to supply a function defining the value of ξ_i, where the parameter `equation` denotes the number i of the considered equation. The values of ξ may depend on the coordinates $x, \ldots,$ the time t, and the solution u.

```
static int UserCauchy(real x, real y, int classA, real t, real *u,
                      int equation, real *fVal)
{
  fVal[0] = xi(equation);

  return true;
}
```

The initial values are given by n (maybe constant) functions $u_{0,1}$,..., $u_{0,n}$. They will be specified in a function together with the derivatives $\partial u_{0,i}/\partial t$. Note that the derivatives have to be specified only when the coefficients B_{ij} in (23) depend on the solution or the gradient of the solution.

```
static int UserInitialFunc(real x, real y, int classA,
                           real *start, real *startUt)
{
  start[0]     =   U01;
```

```
  startUt[0]   =   Ut01;

    ...

  start[n-1]   =   U0n-1;
  startUt[n-1] =   Ut0n-1;

  return true;
}
```

5.3 Declare a Problem

Once the user has defined the functions specifying his problem, e.g.,
UserParabolic(...), UserParabolicStruct(...),
UserLaplace(...), UserLaplaceStruct(...),
UserConvection(...), UserConvectionStruct(...),
UserSource(...), UserSourceStruct(...),
UserJacobian(...), UserJacobianStruct(...),
UserDirichlet(...), UserCauchy(...), and UserInitial(...),
he can establish a new problem by calling the function `SetTimeProblem` in the function `SetUserProblems()` at the end of the file `user.c`.

```
SetUserProblems()
{
   ...

SetTimeProblem("user",userVarName,
                      UserParabolic,
                      UserParabolicStruct,
                      UserLaplace,
                      UserLaplaceStruct,
                      UserConvection,
                      UserConvectionStruct,
                      UserSource,
                      UserSourceStruct,
                      UserJacobian,
                      UserJacobianStruct,
                      UserInitialFunc,
                      UserCauchy,
                      UserDirichlet,
```

```
                        UserSolution));

   ...
}
```

Thus we get a new problem called "user" defined by all these functions as introduced before. A problem with this name is already prepared by the authors. The user can take this example and modify it in order to define his problem. We prefer to introduce new function names and call the function `SetTimeProblem` a second time. Thus we can save a set of problems which are distinguished by their names. `UserSolution` can be used if the true solution of the problem is known, e.g., when testing the code. If not, we set it to 0. The same is valid for other functions. The parameter `userVarName` is an array of names used to define names for the components of the solution, e.g. first component means 'temperature'.

We give two examples:

Example 1. We want to compute the solution of the simple heat transfer equation in 2D, given by

$$\begin{aligned} \frac{\partial T}{\partial t} - \nabla \cdot (k\nabla) &= F, \\ u &= 0 \quad \text{on } \Gamma_D, \\ k\frac{\partial T}{\partial n} &= \beta(T_0 - T) \quad \text{on } \Gamma_C. \end{aligned} \tag{27}$$

$k = k(x, y)$ may depend on the coordinates and $F = F(x, y, t)$ on time and space.

First we implement our coefficient functions, boundary and initial values.

```
static int Ex1Parabolic(real x, real y, int classA, real t,
                        real *u, real *ux, real *uy, real **mat)
{
  mat[0][0] =  1.0;

  return true;
}

static int Ex1ParabolicStruct(int **structM, int **dependsS,
                              int *dependsT,
```

```
                                int *dependsU, int *dependsGradU)
{
  structM[0][0] = F_FILL; dependsS[0][0] = false;

  *dependsT = false;
  *dependsU = false;
  *dependsGradU = false;

  return true;
}

static int Demo1Laplace(real x, real y, int classA, real t,
                        real *u, real *ux, real *uy,
                        real **matXX, real **matXY,
                        real **matYX, real **matYY)
  {
    matXX[0][0] = k(x,y);
    matXY[0][0] = 0.0;
    matYX[0][0] = 0.0;
    matYY[0][0] = k(x,y);

    return true;
  }

static int Ex1LaplaceStruct(int **structM, int **dependsS,
                            int *dependsT,
                            int *dependsU, int *dependsGradU)
{
  structM[0][0] = F_FILL; dependsS[0][0] = true;

  *dependsT = false;
  *dependsU = false;
  *dependsGradU = false;

  return true;
}

static int Ex1Source(real x, real y, int classA, real t,
                     real *u,
                     real *ux, real *uy, real *vec)
{
```

```
  vec[0] =  F(x,y,t);

  return true;
}

static int Ex1SourceStruct(int *structF, int *dependsS,
    int *dependsT, int *dependsU, int *dependsGradU)
{
  structF[0] = F_FILL;   dependsS[0] = true;

  *dependsT = true;
  *dependsU = false;
  *dependsGradU = false;

  return true;
}

static int Ex1InitialFunc(real x, real y, int classA,
                          real *start, real *startUt)
{
  start[0]   =   0.0;

  return true;
}

static int Ex1Dirichlet(real x, real y, int classA, real t,
                        real *u,
                        int equation, real *fVal)
{
  fVal[0] = 0.0-u[0];

  return true;
}

static int Ex1Cauchy(real x, real y, int classA, real t,
                     real *u, int equation, real *fVal)
{
  fVal[0] = beta*(T0 - u[0]);

  return true;
}
```

Now we set up the new problem denoted by "example1".

```
SetTimeProblem("example1",ex1VarName,
                          Ex1Parabolic,
                          Ex1ParabolicStruct,
                          Ex1Laplace,
                          Ex1LaplaceStruct,
                          0,
                          0,
                          Ex1Source,
                          Ex1SourceStruct,
                          0,
                          0,
                          Ex1InitialFunc,
                          Ex1Cauchy,
                          Ex1Dirichlet,
                          0));
```

We have no terms for the convection or the Jacobian nor for the true solution, so we set the corresponding parameters in the call of `SetTimeProblem` to 0. The parameter `ex1VarName` is used to set a name to the solution T, e.g.,

```
static char *ex1VarName[] = {"temperature"};
```

Example 2.

Here we present a system with two equations describing spot replication in $\Omega = [0,1] \times [0,1]$ in 2D, compare the example in Section 3.

$$\begin{aligned} \frac{\partial u}{\partial t} - \nabla(\mu_1 \nabla u) &= -uv^2 + F(1-u) \\ \frac{\partial v}{\partial t} - \nabla(\mu_2 \nabla v) &= uv^2 - (F+k)v \\ u = 1, v = 0 \qquad & \text{on } \Gamma_D = \partial\Omega \end{aligned}$$

and the initial condition in $[0.25, 0.75] \times [0.25, 0.75]$:

$$\begin{aligned} u &= 1.0 - 0.5 \sin^2(4.0\pi x) \sin^2(4.0\pi y), \\ v &= 0.25 \sin^2(4.0\pi x) \sin^2(4.0\pi y), \end{aligned}$$

elsewhere: $u = 1.0$, $v = 0.0$.

We get the following implementation of these equations

```
static int Ex2Parabolic(real x, real y, int classA, real t,
                        real *u, real *ux, real *uy,
                        real **mat)
{
  mat[0][0] = 1.0;
  mat[1][1] = 1.0;

  return true;
}

static int Ex2ParabolicStruct(int **structM, int **dependsS,
                              int *dependsT,
                              int *dependsU, int *dependsGradU)
{
  structM[0][0] = F_FILL;   dependsS[0][0] = false;
  structM[0][1] = F_IGNORE; dependsS[0][1] = false;
  structM[1][0] = F_IGNORE; dependsS[1][0] = false;
  structM[1][1] = F_FILL;   dependsS[1][1] = false;

  *dependsT     = false;
  *dependsU     = false;
  *dependsGradU = false;

  return true;
}

static int Ex2Laplace(real x, real y, int classA, real t,
                      real *u, real *ux, real *uy,
                      real **matXX, real **matXY,
                      real **matYX, real **matYY)
{
```

```
  matXX[0][0] = 2.0e-5/0.25;
  matXY[0][0] = 0.0;
  matYX[0][0] = 0.0;
  matYY[0][0] = 2.0e-5/0.25;

  matXX[1][1] = 1.0e-5/0.25;
  matXY[1][1] = 0.0;
  matYX[1][1] = 0.0;
  matYY[1][1] = 1.0e-5/0.25;

  return true;
}

static int Ex2LaplaceStruct(int **structM, int **dependsS,
                            int *dependsT,
                            int *dependsU, int *dependsGradU)
{
  structM[0][0] = F_FILL;   dependsS[0][0] = false;
  structM[0][1] = F_IGNORE; dependsS[0][1] = false;
  structM[1][0] = F_IGNORE; dependsS[1][0] = false;
  structM[1][1] = F_FILL;   dependsS[1][1] = false;

  *dependsT     = false;
  *dependsU     = false;
  *dependsGradU = false;

  return true;
}

static int Ex2Source(real x, real y, int classA, real t,
                     real *u, real *ux, real *uy,
                     real *vec)
{
  vec[0] =  -u[0]*u[1]*u[1] + 0.024*(1.0-u[0]);
  vec[1] =   u[0]*u[1]*u[1] - (0.024+0.06)*u[1];

  return true;
}

static int Ex2SourceStruct(int *structV, int *dependsS,
                           int *dependsT,
```

```
                            int *dependsU, int *dependsGradU)
{
  structV[0] = F_FILL;   dependsS[0] = true;
  structV[1] = F_FILL;   dependsS[1] = true;

  *dependsT     = false;
  *dependsU     = true;
  *dependsGradU = false;

  return true;
}

static int Ex2InitialFunc(real x, real y, int classA,
                          real *start, real *startUt)
{
  if ((0.25<=x)&&(x<=0.75)&&(0.25<=y)&&(y<=0.75))
    {
      start[0] = 1.0 - 0.5*pow(sin(4.0*REALPI*x),2.0)
                          *pow(sin(4.0*REALPI*y),2.0);
      start[1] = 0.25*pow(sin(4.0*REALPI*x),2.0)
                          *pow(sin(4.0*REALPI*y),2.0);
    }
  else
    {
      start[0] = 1.0;
      start[1] = 0.0;
    }

  return true;
}

static int Ex2Dirichlet(real x, real y, int classA, real t,
                        real *u, int equation, real *fVal)
{
   switch (equation)
      {
        case 0: fVal[0] = u[0] - 1.0;
                break;
        case 1: fVal[0] = u[1] - 0.0;
                break;
      }
```

```
   return true;
}
```

These functions are used to introduce the new example "example2" by the function SetTimeproblem .

```
SetTimeProblem("example2",ex2VarName,
                          Ex2Parabolic,
                          Ex2ParabolicStruct,
                          Ex2Laplace,
                          Ex2LaplaceStruct,
                          0,
                          0,
                          0,
                          0,
                          Ex2Source,
                          Ex2SourceStruct,
                          0,
                          0,
                          Ex2InitialFunc,
                          0,
                          Ex2Dirichlet,
                          0));
```

The array of strings `ex2VarName` contains the names of the two components of the solution , i.e.,

```
static
char *ex2VarName[] = {"concentration_0","concentration_1"};
```

Finally the user has to provide a geometrical description of the domain Ω including the specification of the boundaries as well as a classification of subdomains (e.g., in order to distinguish material properties).

5.4 Triangulation of Domain

KARDOS expects a triangulation of the domain $\overline{\Omega}$ in a file. The triangulation has to be given in the following format:

5.4.1 1D-geometry

A triangulation in 1D means a partiton of an interval $[a, b]$ and the definition of the two boundary points a and b. The data have to be given in the form (after a %–sign the rest of a line is comment):

```
name of triangulation
Dimension:(number of points,number of edges)noOfBoundaryTypes
% noOfBoundaryTypes: number of boundary types
% definition of boundary types
0:boundary type
...
...
l:boundary type
END
% it follows a list of points
0: coordinate of point 1,type of boundary
                         or 'I' for inner point
1: coordinate of point 2,type of boundary
                         or 'I' for inner point
...
...
...
2: coordinate of point n,type of boundary
                         or 'I' for inner point
END
% in this section we find the edges defined by their vertices
0:(number vertex 1, number of vertex 2)
1:(number vertex 1, number of vertex 2)
...
...
m:(number vertex 1, number of vertex 2)
END
```

As an example, we give the triangulation (name `unit`) of the unit interval $[0, 1]$. It comprises three points, two of them at the boundary, and two edges $[0, 0.5]$, $[0.5, 1]$. We have two types of boundary condition, DIRICHLET and CAUCHY, referenced by `B0` and `B1`, respectively.

```
unit
Dimension:(3,2)2
0:D
1:C
END
0:0.00,B0
1:0.5,I
2:1.00,B1
END
0:(0,1)
1:(1,2)
END
```

In the case of a system with n equations you have to define the boundary type for each of the equations. If $n = 4$, the example could look like

```
unit
Dimension:(3,2)2
0:DDCC
1:CDCD
END
0:0.00,B0
1:0.5,I
2:1.00,B1
END
0:(0,1)
1:(1,2)
END
```

This shows that different components of the solution may have different boundary types.

5.4.2 2D-geometry

Triangulations in 2D or in 3D have to be described by some *keyword sections* using a keyword followed by specific informations.

Necessary keyword sections:

- `no_of_points`
  Number of grid points in the triangulation.
- `no_of_boundary_edges` Number of edges on the boundary of the triangulation.
- `no_of_triangles`
  Number of triangles in the triangulation.
- `boundary_types`
  For each component of the solution you have to write one letter as description of its boundary condition, e.g.,

  ```
  boundary_types
  0:DDD
  1:DNC
  ```

  In this example we define two types of boundary conditions for a system of three equations, i.e. the solution has three components. B0 means that each component has Dirichlet boundary, B1 means that the first component is of DIRICHLET type, the second of NEUMANN type, and the third of CAUCHY type.
- `points`
  Coordinates of the grid points. The enumeration of the points by non-negative numbers is arbitrary. The point number can be referenced when defining other properties (boundary type, class,...) of the point, or the triangles or boundary edges. Example: definition of four grid points with numbers 5, 6, 7, and 8.

  ```
  Points
  5: 0.000000e+00,0.000000e+00
  6: 1.000000e+00,0.000000e+00
  7: 1.000000e+00,1.000000e+00
  8: 0.000000e+00,1.000000e+00
  ```

- `triangles`
  Triangles are defined by three of the point numbers used in the `points` section. The numbering of the triangles is arbitrary. The triangle numbers can be referenced when defining other properties of a triangle, e.g. a classification parameter.

  As example, we show the definition of two triangles:

```
triangles
0: 5,6,7
1: 5,7,8
```

- `boundary_edges`
  The edges on the boundary are given by a pair of point references (defined in the point list), and a specifier of the boundary types list.

  The definition of boundary conditions for four edges looks like

```
boundary_edges
(5,6):0
(6,7):0
(7,8):0
(8,5):0
```

- `boundary_points`
  Points on the boundary are defined by the reference of that point and the type of boundary condition.

  See for example the definition of the boundary conditions in the points 5, 6, 7,and 8:

```
boundary_points
5:0
6:0
7:0
8:0
END
```

Auxilary keyword sections:

- `name_of_triangulation`
  Name of the triangulation.

- `no_of_edges`
  Number of edges in the triangulation.

- `point_type_names`
  Each type name can be used to add an attribute to a point.

  Example: definition of three types.

  ```
  point_type_names
  property_1
  property_2
  property_3
  ```

- `edge_type_names`
  Each type name can be used to add an attribute to an edge.

  Example: definition of two types.

  ```
  edge_type_names
  property_7
  property_8
  ```

- `triangle_type_names`
  Each type type can be used to add an attribute to a triangle.

  Example: definition of three material types.

  ```
  triangle_type_names
  Zinc
  iron
  copper
  ```

- `point_type`
  Each point may get an attribute from the `point_type_names` list. This attribute can be used somewhere in the code.

  Example: definition of types in two points with the reference numbers 5 and 6.

  ```
  point_type
  5:property_1
  6:property_3
  ```

- `edge_type`
  Each edge may get an attribute from the `edge_type_names` list. This attribute can be used somewhere in the code.

  Example: definition of types in two edges defined by the points with the reference numbers 0, 1, and 2.

```
edge_type
(0,1):property_7
(1,2):property_8
```

- `triangle_type`
  Each triangle may get an attribute from the `triangle_type_names` list. This attribute can be used somewhere in the code.

  Example: definition of a material property in two triangles with the reference numbers 0 and 1.

```
triangle_type
0:ZiNC
1:copper
```

- `solution_at_point`
  For each point in the grid you can specify as many real numbers as you have nodes in that point.

  Example: Each of the points with reference numbers 5, 6, 7, 8 carries two node values.

```
solution_at_point
5:0.000000e+00,0.000000e+00
6:1.000000e+00,1.000000e+00
7:2.000000e+00,2.000000e+00
8:3.000000e+00,3.000000e+00
```

- `solution_at_edge`
  For each edge in the grid you can specify as many real numbers as you have nodes in that edge.

  Example: Each of the four edges (5,6), (6,7), (7,8), and (7,5) carries two node values.

```
solution_at_edge
(5,6):0.000000e+00,0.000000e+00
(6,7):1.000000e+00,1.000000e+00
(7,8):2.000000e+00,2.000000e+00
(8,5):3.000000e+00,3.000000e+00
(7,5):4.000000e+00,4.000000e+00
```

- `solution_at_triangle`
  For each triangle in the grid you can specify as many real numbers as you have nodes in that triangle.

  Example: Each of the two triangles carries one node value.

```
solution_at_triangle
0:0.000000e+00
1:1.000000e+00
```

- `no_of_circles`
  number of midpoints of circles which are used for providing circular edge lines. Each of these edges will be treated internally as circle line around the midpoint. Both a midpoint and the corresponding edges are to be specified after the keyword `circles`.

- `circle_midpoints`
  In this section we provide a possibility to modify the refinement of an edge. Normally it is refined generating two new edges, each of them connects one of the boundary points of the edge with its midpoint. Alternatively, we can determine other coordinates for the new point. This can we done in a very general way, but we only offer the following possibility: Connect the edge you want to refine with another point which has the same distance from each of the boundary points of the edge, the so called "circle_midpoint". Then, we take as new point (instead of the midpoint) the point on the circle arount the "circle_midpoint" which is also on the straight line between the midpoint and the "circle_midpoint'. This techniques can be used to introduce circular boundaries during the refinement process.

  Example: definition of one "circle_midpoint" with coordinates x, y.

```
circle_midpoints
0:x,y
```

Note that the index has to start with 0.

- `circle_edges`
  Corresponding to the points be defined in the `circle-midpoints` section we can define a set of edges which will become circular during the refinement process as decribed above.

  Example: definition of four edges getting circluar during the refinement process.

```
circle_edges
(5,6):0
(6,7):0
(7,8):0
(8,5):0
```

Sequence of keywords in an input data file:
In any case, the user has to specify
the number of points `no_of_points`,
the number of boundary edges `no_of_boundary_edges`,
the number of triangles `no_of_triangles`,
the number of boundary types `no_of_boundary_types`, and the list of boundary types `boundary_types`,
followed by the list of points `points`, triangles `triangles`, and boundary edges `boundary_edges`.

All the other keyword sections are not necessary. If they are not used the code sets default values, e.g. if the `point_class` keyword doesn't occur we suppose a value 0. If a material type is used (in the `point_class`, `edge_class`, or triangle_class sections) it must have been defined before by the sections `no_of_material_types` and `material_types`.

Note: A text after the character # in a keyword line is taken as comment.

A complete example:

```
name_of_triangulation
unit square

no_of_points # comment
4

no_of_edges
```

```
5

no_of_boundary_edges
4

no_of_triangles
2

boundary_types
0:DD
1:NN

point_type_names
1:Point_is_on_Circle
2:Point_is_on_Line

edge_type_names
Edge_is_on_Circle
Edge_is_on_Line

triangle_type_names
zinc
iron
copper

Points
5: 0.000000e+00,0.000000e+00
6: 1.000000e+00,0.000000e+00
7: 1.000000e+00,1.000000e+00
8: 0.000000e+00,1.000000e+00

boundary_points
5:0
6:0
7:0
8:0

point_type
5:Point_is_on_Circle
6:Point_is_on_Line
```

```
triangles
0: 5,6,7
1: 5,7,8

triangle_type
0:zinc
1:copper

boundary_edges
(5,6):0
(6,7):0
(7,8):0
(8,5):0

edge_type
(5,7):Edge_is_on_Line

circle_midpoints
0:0.5,0.5

circle_edges
(5,6):0
(6,7):0
(7,8):0
(8,5):0

solution_at_point
5:0.000000e+00,0.000000e+00
6:1.000000e+00,1.000000e+00
7:2.000000e+00,2.000000e+00
8:3.000000e+00,3.000000e+00
```

5.4.3 3D-geometry

Triangulations in 3D have to be described by some *keyword sections* using a keyword followed by specific informations. The structure is analogous to the 2D–case.

Necessary keyword sections:

- no_of_points

Number of grid points in the triangulation.

- no_of_tetrahedra
  Number of tetrahedra in the triangulation.

- no_of_boundary_triangles Number of triangles on the boundary of the triangulation.

- no_of_boundary_types
  Number of boundary types, e.g. NEUMANN, DIRICHLET, or CAUCHY.

- boundary_types
  For each component of the solution you have to write one letter as description of its boundary condition, `D` for DIRICHLET, `N` for NEUMANN, or `C` for CAUCHY. For example:

  ```
  boundary_types
  0:DDD
  1:DNC
  ```

  Here, we define two types of boundary conditions for a system of three equations (the solution vector has three components). Condition 0 means that each component has Dirichlet boundary, condition 1 means that the first component is of DIRICHLET type, the second of NEUMANN type, and the third of CAUCHY type.

- points
  Coordinates of the grid points. The enumeration of the points by non-negative numbers is arbitrary. The point number can be referenced when defining other properties (boundary type, class,...) of the point, or the triangles or boundary edges.

  Example: definition of four grid points with numbers 5,6,7, and 8.

  ```
  Points
  5: 0.000000e+00,0.000000e+00,0.000000e+00
  6: 1.000000e+00,0.000000e+00,1.000000e+00
  7: 1.000000e+00,1.000000e+00,0.000000e+00
  8: 0.000000e+00,1.000000e+00,1.000000e+00
  ```

- tetrahedra
  Triangles are defined by four point numbers as used in the `points` section. The numbering of the tetrahedra is arbitrary. The tetrahedra

numbers can be referenced when defining other properties of a tetrahedron, e.g. a classification parameter.

Example: definition of one tetrahedron.

```
tetrahedra
0: 5,6,7,8
```

- boundary_triangles
  The triangles on the boundary are given by three point references (defined in the point list), and a specifier of the boundary types list.

  Example: definition of boundary conditions for two triangles.

```
boundary_triangles
(5,6,7):0
(6,7,8):1
```

Auxilary keyword sections:

- name_of_triangulation
  name of the triangulation.
- no_of_edges
  number of edges in the triangulation.
- boundary_points
  Points on the boundary are defined by the reference of that point and the type of boundary condition. Example: definition of the boundary conditions in the points 5,6,7,and 8.

```
boundary_points
5:0
6:0
7:0
8:0
END
```

- point_type_names
  Each type name can be used to add an attribute to a point.

  Example: definition of three types of points.

```
point_type_names
1:property_1
2:property_2
3:property_3
```

- edge_type_names
  Each type name can be used to add an attribute to an edge.

  Example: definition of two types of edges.

```
edge_type_names
1:property_1
2:property_2
```

- triangle_type_names
  Each type name can be used to add an attribute to a triangle.

  Example: definition of three types of triangles.

```
triangle_type_names
1:type_1
2:type_2
3:type_3
```

- tetrahedron_type_names
  Each type name can be used to add an attribute to a tetrahedron, e.g., a material property.

  Example: definition of three types of tetrahedra.

```
tetrahedron_type_names
1:zinc
2:iron
3:copper
```

- point_type
  Each point may get an attribute which can be used somewhere in the code.

  Example: definition of a property in two points with the reference numbers 5 and 6.

```
point_type
5:property_1
6:property_2
END
```

- edge_type
  Each edge may get an attribute which can be used somewhere in the code.

  Example: definition of a property in two edges.

```
edge_type
(5,6):property_1
(7,8):property_2
END
```

- triangle_type
  Each triangle may get an attribute which can be used somewhere in the code.

  Example: definition of properties in two triangles.

```
triangle_type
(5,6,7):type_1
(6,7,8):type_2
END
```

- tetrahedron_type
  Each tetrahedron may get an attribute which can be used somewhere in the code.

  Example: definition of material properties in two tetrahedra with the reference numbers 0 and 1.

```
triangle_type
0:zinc
1:copper
END
```

- solution_at_point
  For each point in the grid you can specify as many real numbers as you have nodes in that point.

  Example: Each of the points with reference numbers 5, 6, 7, 8 carries two node values.

```
solution_at_point
5:0.000000e+00,0.000000e+00
6:1.000000e+00,1.000000e+00
7:2.000000e+00,2.000000e+00
8:3.000000e+00,3.000000e+00
```

- solution_at_edge
  For each edge in the grid you can specify as many real numbers as you have nodes in that edge.

  Example: Each of the four edges (5,6), (6,7), (7,8), and (7,5) carries two node values.

```
solution_at_edge
(5,6):0.000000e+00,0.000000e+00
(6,7):1.000000e+00,1.000000e+00
(7,8):2.000000e+00,2.000000e+00
(8,5):3.000000e+00,3.000000e+00
(7,5):4.000000e+00,4.000000e+00
```

- solution_at_triangle
  For each triangle in the grid you can specify as many real numbers as you have nodes in that triangle.

  Example: Each of the two triangles carries one node value.

```
solution_at_triangle
(5,6,7):0.000000e+00
(6,7,8):1.000000e+00
```

- solution_at_tetrahedron
  For each tetrahedron you can specify as many real numbers as you have nodes in that tetrahedron.

  Example: Each of the two tetrahedra carries one node value.

```
solution_at_tetrahedron
0:0.000000e+00
1:1.000000e+00
```

Sequence of keywords in an input data file:
In any case, the user has to specify the number of points `no_of_points`, the number of boundary triangles `no_of_boundary_triangles`, the number of tetrahedra `no_of_tetrahedra`, the number of boundary types `no_of_boundary_types`, and the list of boundary types `boundary_types`, followed by the list of points `points`, tetrahedra `tetrahedra`, and boundary triangles `boundary_triangles`.

All the other keywords are not necessary. If they are not used the code sets default values, e.g. if the `point_type` keyword doesn't occur we suppose a value `undefined`. If a type name is used (in the `point_type`, `edge_type`, `triangle_type`, or `tedrahedron_type` sections) it must have been defined before by the sections `point_type_names`, `edge_type_names`, `triangle_type_names`, or `tetrahedron_type_names`. The code stores the type names in suitable string arrays, each type name at the position which is set in the type name sections. This position is also stored in the data structure (component `classA`) of the corresponding object (point, edge,...). The parameter `classA` in the definition of the coefficient functions (compare the first subsections of Section 5) is exactly this component in the data structure. There it can be used to distinguish different properties (e.g., materials) of the objects.

Note: A text after the character # in a keyword line is taken as comment.

A complete example:

```
name_of_triangulation
tet8

no_of_points
10

no_of_tetrahedra
8

no_of_boundary_triangles
16

boundary_types
0:DD
1:CC

point_type_names
1:property_1
2:property_2
```

```
edge_type_names
1:property_3
2:property_4

triangle_type_names
1:property_5
2:property_6

tetrahedron_type_names
1:zinc
2:iron

points
0: 0.000000e+00, 0.000000e+00, 0.000000e+00
1: 0.000000e+00, 0.000000e+00, 1.000000e+00
2: 0.000000e+00, 1.000000e+00, 0.000000e+00
3: 1.000000e+00, 0.000000e+00, 0.000000e+00
4: 0.000000e+00, 5.000000e-01, 5.000000e-01
5: 5.000000e-01, 0.000000e+00, 5.000000e-01
6: 5.000000e-01, 5.000000e-01, 0.000000e+00
7: 0.000000e+00, 0.000000e+00, 5.000000e-01
8: 5.000000e-01, 0.000000e+00, 0.000000e+00
9: 0.000000e+00, 5.000000e-01, 0.000000e+00

boundary_points
0: 0
1: 0
2: 0
3: 0
4: 0
5: 1
6: 1
7: 1
8: 1
9: 0


tetrahedra
0: 9,    8,    7,    0
1: 9,    4,    6,    2
```

```
2: 7,    5,    4,    1
3: 8,    6,    5,    3
4: 5,    8,    9,    6
5: 9,    5,    6,    4
6: 9,    7,    8,    5
7: 5,    4,    9,    7


point_type
0:property_1
1:property_1
2:property_2
3:property_2
4:property_1
5:property_2
6:property_2
7:property_2
8:property_2


edge_type
(4,6):property_3
(5,1):property_4


triangle_type
(4,6,2):property_5
(5,1,4):property_6


tetrahedron_type
0:zinc
1:iron
2:iron
3:iron
4:iron
5:zinc
6:zinc
7:zinc

boundary_triangles
```

```
(    4,    6,    2),0
(    5,    4,    1),0
(    6,    5,    3),0
(    6,    4,    5),0
(    7,    8,    0),0
(    5,    7,    1),0
(    8,    5,    3),0
(    8,    7,    5),0
(    9,    8,    0),0
(    6,    9,    2),0
(    8,    6,    3),0
(    8,    9,    6),0
(    7,    9,    0),0
(    4,    7,    1),0
(    9,    4,    2),1
(    9,    7,    4),1


solution_at_point
0: 2.0,1.0
1: 2.0,1.0
2: 2.0,1.0
3: 2.0,1.0
4: 0.0,1.0
5: 0.0,1.0
6: 0.0,1.0
7: 0.0,1.0
8: 0.0,1.0
9: 0.0,1.0
```

5.5 Number of Equations

First of all the user has to announce the number of equations in his problem. This has to happen in the source file `kardos.c` in the directory `kardos/kardos`. There you can find the line

```
noOfEquations = ...;
```

For instance, you may set

```
noOfEquations = 1;
```

if you have a scalar equation as in many heat transfer problems.

5.6 Starting the Code

After recompiling you can start the program in the directory where you have defined your problem, normally in `kardos/problems/user`. To avoid copying the executable into this directory or typing long path specification you should set links. For that purpose just type `setLink`. This script file sets the links to the executable and to the file `kardos/kardos/kardos.def` and `kardos/kardos/kask.color` which include some presetting information.

Having been started by typing '`kardos`', the program will write out the prompter '`Kardos:`'. Now you are in the command mode and can communicate with the code by supplying a command from the list of commands, see Section 6.

Normally, you will use commands in the following sequence:

- `read` *filename*
  reads the file with the name *filename*, e.g., user.geo, including the triangulation of the domain (see Subsection 5.4). The initial grid is constructed.

  Example: `read user.geo`

- `timeproblem` *problem name*
  selects the problem with the name *problem name*, i.e. the coefficients, the boundary and initial values of your equation (1).

  Example: `timeproblem user`

- `seltimeinteg` *time integrator*
  selects a method for time integration. Possible methods are *ros1*, *ros2*, *ros3*, *ros3p*, *rodas3*, *rowda3ind2*, *rowda3*, *rodas4*, *rodas4p*.

  Example: `seltimeinteg ros2`

- `selestimate` *error estimator*
  selects a method for estimation of the spatial errors in each time step. Possible estimators are *hb* (hierarcical bases) and *cv* (curvature).

  Example: `selestimate hb`

- `selrefine` *refinement strategy*
  selects a strategy for refining the mesh by evaluating the local errors provided by the error estimator. The following methods are available: *all*, *maxvalue_e*, *meanvalue_e*, and *extrapol_e*.

  Example: `selrefine maxvalue_e`

- `seliterate` *iteration method*
  selects an iterative solver for the linear algebraic systems which have to be solved in each time step. Available are: BICGStab-method (*pbicgstab*), cg-method (*pcg*), and the gmres-solver (*pgmres*).

  Example: `seliterate pbicgstab`

- `selprecond` *preconditioner*
  selects a preconditioner for the iterative solver. Available are: ILU, SSOR, block-SSOR, or NONE for no preconditioning.

  Example: `selprecond ilu`

- `setpartime` sets parameters for the time integration. The main important are

  - *tstart*: initial time. Default value: 0.0.
  - *tend*: time up to which a calculation has to be computed.
  - *maxsteps*: maximum number of time steps.
  - *timetol*: maximum tolerance for error in time discretization.
  - *spacetol*: maximum tolerance for error in space discretization.
  - *globtol*: maximum tolerance for global discretization error.
  - *timestep*: step size in the next time step.
  - *direct*: enables direct instead of iterative solving for linear system. The direct solver is selected by the `seldirect` command.

- `timestepping`
  starts the solution algorithm. Some output informs on the progress of the process.

- `quit`
  quits the program.

The user can write all the commands for his run of KARDOS in a file. One line can contain one command. We call such a file a *command file*, and prefer

filenames with the extension `ksk`, e.g., `user.ksk`. If this file is given to the program by the `do-command`, all the included commands will be executed before asking for a new command.

Example: `do user.ksk`

Only the commands `read`, `timeproblem`, and `timestepping` are mandatory. The other commands are used to set parameters controlling and optimizing the numerical algorithm. Default values are provided.

A typical command file may have the following form:

```
read user.geo
timeproblem user
setpartime tstart 0.0 tend 10.0 timestep 0.1 globtol 0.01
timestepping
```

In the Section 6 we provide a list of all important commands and give some explanations about their algorithmic background.

6 Commands and Parameters

In Section 5 we described most of the work necessary to get a new problem into the code. Now we present two features which allow to control the solving process and to supply problem specific parameters (e.g. material properties, physical constants) interactively.

6.1 Command Language Interface

The command language interface to the KARDOS program consists of a set of commands with parameters. The input source can be stdin or text files.

This command language has a simple syntactic structure. A command is limited by a newline or a command delimiting character. Parameters are seperated by white space (e.g. blanks or tabs), exceptions are strings which are quoted by string delimiting characters.

The rest of an input line is ignored after a comment character (predefined '%'). Capitel letters are treated as small letters.

More details about the command language can be found in [25].

Some of the informations of this subsection have already been mentioned in the end of Section 5.

Having been started by typing '`kardos`', the program will write out the prompter '`Kardos:`'. Now you are in the command mode and can communicate with the code by supplying a command from the list of commands. We know one exception: If the user supplies a file called `kardos.ksk` containing a set of commands it will be executed just after the program has been started. This allows batch processing. If no `quit` is included, more commands are requested from standard input.

A complete command file should comprise commands to

- select the problem (e.g. `timeproblem`),
- read the geometric input data (e.g. `read`),
- set parameters describing the solution method (e.g. `seltimeinteg`),
- request output processing (e.g. `window`, `graphic`).

Alphabetical list of commands:

- `amiramesh`
  prints the current triangulation and the solution into a file *AmiraMesh.geo* using the format of the visualization software *amira*™. Only for three-dimensional grids!

- `checktri`
  checks the consistency and completeness of the triangulation, e.g. missing boundary conditions are reported.

- `do` *filename*
  reads a file *filename* which may include a sequence of commands. Each of the commands will be executed.

- `error` *parameter*
  computes the true error of the computed FEM solution if the exact solution is known and specified by the user, see Section 5. *parameter* may be one of the values *max*, *l2*, or *h1* and selects the norm in which the error is computed. This command makes sense only after having computed the FEM solution.

  Example: `error l2`

  By this command the error is computed in the L_2-norm.

- `graphic`
  selects graphic parameters. This command selects for the current graphical output stream the requests for the actual drawing (see the command `show` or the parameter `automatic` of the `window` command). Only for 1- and 2-dimensional applications!

  The main important parameters are

  - *triangulation*: draw the triangulation,
  - *boundary*: draw the boundary of the triangulation,
  - *solution*: draw isolines of a component of the solution vector,
  - *temperature*: draw the solution as color-washed graphic,
  - *levels*: define number of levels in an isoline plot, default is 10,
  - *areas*: draw elements (1D: edges, 2D: triangles) in different colours,
  - *clear*: delete all preceding parameter settings.

- `help` *name*
  prints some information about the command *name*.

  Example: `help do`

- `help`
  prints a list of all commands.

- `infgraphic`
  informs on graphic parameters.

- `infpar`
  prints a list of all user defined parameter lists. See next subsection for some explanations of the dynamical parameter handling.

- `infpar` *parameter name*
  informs on the current values of the user defined parameters in the parameter list called *parameter name*.

- `inftimeinteg`
  informs on the parameters of the time integration process. These parameters are set by the commands `setpartime` and `seltimeinteg`.

  Example:

```
Kardos: inftimeinteg
Current time integrator is 'rowda3'
Current parameters are:
  scaling                0   (0= off, 1= on, scaled error norm used)
  direct                 0   (0= off, 1= on, direct solver used)
  maxsteps            1000   (maximal time steps)
  maxreductions         10   (maximal reductions per time step)
  stdcontrol             0   (0= off, 1= on, standard controller used)
  tstart          0.00e+00   (starting time)
  tend            0.00e+00   (final time)
  timestep        1.00e-04   (proposed time step)
  maxtimestep     1.00e+20   (maximal allowed time step)
  timetol        -1.00e+00   (desired tolerance for time integrator)
  spacetol       -1.00e+00   (desired tolerance for spatial solver)
  globtol         0.00e+00   (desired global tolerance)
  timetolfac      5.00e-01   (timetol/globtol)
  spacetolfac     5.00e-01   (spacetol/globtol)
```

- `inftimeproblem`
  informs on the structure of the selected parabolic problem.

- `inftri`
  informs on the current triangulation.

  Example:

```
Kardos: inftri
Tri: current triangulation 'Square' from 'Grids/flame.geo'
       noOfPoints          :   79
       noOfEdges           :  190
       noOfTriangles       :  112
       noOfInitPoints      :   79
       noOfInitEdges       :  190
       noOfInitTriangles   :  112
       refLevel            :    0
       maxDepth            :    0
Kardos:
```

- `infwindow`
  informs on parameters of the window driver.

- `quit`
  quits the program.

- `read` *filename*
  reads the file with the name *filename*, e.g., user.geo, including the triangulation of the domain (see Subsection 5.4). The initial grid is constructed.

  Example: `read user.geo`

- `seldirect` *direct solver*
  selects a method for direct solution of the linear algebraic systems, default: ma28. As *direct solver* we supply

  - *fullchol*: Cholesky algorithm for general symmetric, positive definit problems,
  - *envchol*: Cholesky algorithm for sparse symmetric, positive definit problems using particular storage (envelope) scheme for matrix and renumbering of nodes (Reverse Cuthill/McKee),
  - *ma28*: MA28 method from the Harwell library, for unsymmetric problems.

- `seldraw` *variable*
  selects the variable (component of the solution vector) to be drawn.

- `selestimate` *error estimator*
  selects a method for estimation of the spatial errors in each time step.

Possible estimators are *hb* (hierarchical bases) and *cv* (curvature). The user may also select *none* when he wants to switch off error estimation.

Example: `selestimate hb`

- `seliterate` *iteration method*
  selects an iterative solver for the linear algebraic systems to be solved in each time step. Available are: BICGStab-method (*pbicgstab*), cg-method (*pcg*), and the gmres-solver (*pgmres*).

  Example: `seliterate pbicgstab`

- `selprecond` *preconditioner*
  selects a preconditioner for the iterative solver. Available are: ILU, SSOR, block-SSOR, or NONE for no preconditioning.

  Example: `selprecond ilu`

- `selrefine` *refinement strategy*
  selects a strategy for refining the mesh by evaluating the local errors provided by the error estimator. The following methods are available: *all*, *maxvalue_e*, *meanvalue_e*, and *extrapol_e*.

  Example: `selrefine maxvalue_e`

- `seltimeinteg` *time integrator*
  selects a method for time discretization. Possible time integrators are *ros1*, *ros2*, *ros3*, *ros3p*, *rodas3*, *rowda3ind2*, *rowda3*, *rodas4*, *rodas4p*.

  Example: `seltimeinteg ros2`

- `setpar`
  changes values in the parameter list defined by the first parameter, followed by pairs of names and values.

  Example: `setpar ssor omega 0.5`
  Here we have the parameter list `ssor` which includes at least one parameter, i.e. `omega`. By this command the value of `omega` is set to `0.5`. See for details in the next subsection.

- `setpardirect` The direct solver implemented in KARDOS has three parameters: *dropfac*, *licnfac*, and *lirnfac* with the following meaning

  - *dropfac* is an real variable. If it is set to a positive value, then any non-zero whose modulus is less than *dropfac* will be dropped from the factorization. The factorization will then require less storage but will be inaccurate. Default value: 0.0.

- *licnfac* is an integer variable, proportional to length of array for column indices, default=1, has to be increased if direct solver calls for more memory.
- *lirnfac* proportional to length of array for row indices, default=1, has to be increased if direct solver calls for more memory.

Example: `setpardirect licnfac 2`
This command allows double as much memory for storing column indices as allowed by default.

- `setparerror` *norm*
enables the computation of the true error of the FEM solution after each time step. This option makes sense only if the exact solution is known and specified by the user, see Section 5. *norm* may be one of the values *max*, *l2*, or *h1* and selects the norm in which the error is computed.

- `setpariter` *parameters*
First the user has to select a method for solving the linear algebraic system by the `seliterate` command. Then he can define *parameters* determining details of the iteration process:

  - *itertol*: relative precision of solution of linear system,
  - *itermaxsteps*: maximal number of iteration steps,
  - *krylovdim*: maximal dimension of Krylov space. Only for iteration method *gmres*.

  Example: `setpariter itertol 1.0e-5 itermaxsteps 200`

- `setparrefine` *parameters*
By this command the user can influence the adaptive refinement process. The error estimator (selected by the command `selestimate`) provides informations about the distribution of local errors. The refinement strategy (selected by the command `selrefine`) evaluates the local errors and determines where to refine the grid. The *parameters* supplied by the `setparrefine` command provide another possibility to affect the refinement process:

  - *maxpoints*: maximal number of points in grid. If this number is reached, no further refinenement is allowed.

- *refrate*: determines the rate of increasing points from one level of refinement to the next, e.g. *refrate=0.3* means that the number of points has to increase by at least 30 %.
- *maxdepth*: maximal number of refinement depth. If this depth is reached, no further refinement is allowed.

Example: `setparrefine refrate 0.3`

- `setpartime` *parameters*
  sets parameters for the time integration. The main important are
  - *tstart*: initial time. Default value: 0.0.
  - *tend*: time up to which a calculation has to be computed.
  - *maxsteps*: maximum number of time steps.
  - *timetol*: maximum tolerance for error in time discretization.
  - *spacetol*: maximum tolerance for error in space discretization.
  - *globtol*: maximum tolerance for global discretization error.
  - *timestep*: step size in the next time step.
  - *direct*: enables direct instead of iterative solving for linear system. The direct solver is selected by the `seldirect` command.

  Example: `setpartime tend 3600.0 globprec 1.0e-2`
  or
  Example: `setpartime direct 1`

- `setscaling` *parameters*
  sets the parameters *atol* and *rtol* which determine how accurate each of the components of the solution vector is computed. *rtol* measures the relative precision of a component. If the modulus of a component is less than *atol* then there are no further efforts to compute this component more accurate.

  To say it precisely: The relative discretization error e_h of the approximate solution $u = (u_i)_{i=1,n}$ on the triangulation τ_h is given by

$$\|e_h\| = \sum_{i=1}^{n} \frac{\|e_{h,i}\|}{ATOL_i + \|u_i\| RTOL_i}$$

  where $e_{h,i}$ is the error of the i-th component.

  Example:
  `setscaling atol 1.0e-12 1.0e-12 1.0e-12 rtol 1.0 1.0 1.0`

- `show`
  draws a picture. The number of a graphical port can be used as parameter of the command. This is useful if there are more than one windows for graphical output.

  Example: `show 1`

- `timeproblem` *problem name*
  selects the problem with the name *problem name*, i.e. the coefficients, the boundary and initial values of your equation (1).

  Example: `timeproblem user`

- `solveinform` *parameter*
  selects different levels of information by the code. By default, the user only gets short messages about the progress of the program. More details can be ordered by setting some of the parameters *iterinfo*, *estiinfo*, *refinfo*, or *solveinfo* to 1:

  - *iterinfo*: information about the solution of the linear systems
  - *estiinfo*: information about the error estimation
  - *refinfo*: information about the adaptive refinement process
  - *solveinfo*: same effect as setting the three preceding parameters to 1.

  Example:

  ```
  solveinform iterinfo 1 estiinfo 1
  ```

- `timestepping`
  starts the solution algorithm. Some output informs on the progress of the process.

- `write` *filename*
  writes the current triangulation into the file with the name *filename*, e.g., user.geo. The standard geometry format is used as described in Section 5. The output of the write command can be reread by the read command.

  Example: `write user.geo`

- `window` *parameters*
  open (or update) a graphical window. The *file* parameter defines the output file name for *postscript*. The *name* parameter defines the window headline for *screen*. *automatic* can be used to call drawing routines at certain events, e.g. when a time step is finished.

  Example: `window new automatic postscript file "picture.ps"`

  In this example, a new port for postscript output is opened. The port is connected to a file with the name picture.ps. After each time step a new picture is generated. What kind of graphics is drawn has to be defined by the `graphic` command.

6.2 Dynamical Parameter Handling

The parameter module includes routines to handle named parameter lists. A parameter list itself contains a list of parameter values of fixed size. A list of parameter names and a list of named values may be maintained. All the technical details about dynamical parameter handling can be read in [25].

From the user's point of view, it is most important to know the routine

```
(char*) NewParamList(char *buf, char *listName, int noOfParams,
                     int valueSize, char **names, int type,
                     int noOfValNames, char **valNames, char vals,
                     int(*UserParamChanged)(char*,char*,int),
                     int(*UserListChanged)(char*))
```

This function uses `buf` as storage for a parameter list or, if `buf==nil` allocates new storage. `listName` is checked for double definitions. The result of the function is the address to an array of `noOfParams` blocks of `valueSize` bytes of memory.

Parameter values may be named to allow a user–friendly input via the `setpar` command. A list of name-value pairs with length `noOfValNames` is defined by `valNames` and `vals`.

If the user wants to be notified on changes of parameters or the complete list a user routine `UserParamChanged` or respectively `UserListChanged` may be supplied. When using the function `NewParamList` the include file `params.h` has to be added in top of the file.

Example:

Here we introduce a parameter list `coefficients` with two real parameters called `alpha` and `beta`.

```
#include "params.h"

...

static real *userData = nil;
static char *coefficientNames[] = {"alpha","beta"};

if (userData==nil)
  {
    userData = (real*)NewParamList(nil,"coefficients",
                2,sizeof(real),
                coefficientNames,T_REAL,0,(char **)nil,
                (ptr)nil,(int (*)(char*,char*,int))nil,
                (int (*)(char*))nil);

    userData[0] =   1.0;
    userData[1] =  20.0;
  }
```

It is recommended to do these definitions directly before defining the problem by `SetTimeProblem`. The statements

```
    userData[0] =   1.0;
    userData[1] =  20.0;
```

define some initial values for these parameters. The parameter `userData[0]` is associated with the name `alpha`, and the parameter `userData[1]` is associated with the name `beta`. These names can be used when changing these parameters during runtime by the `setpar` command, e.g.

```
Kardos: infpar coefficients
        coefficients       alpha  1.0000e+00   beta  2.0000e+1
Kardos: setpar coefficients alpha 0.5 beta 10.0
Kardos: infpar coefficients
        coefficients       alpha  5.0000e-01   beta  1.0000e+1
```

Once we have defined such a parameter list, we can use it somewhere in the code, e.g. when we define the function for the diffusion in a system of two equations (compare Example 2 in Section 5).

```
static int Ex2Laplace(real x, real y, int classA, real t,
                      real *u, real *ux, real *uy,
                      real **matXX, real **matXY,
                      real **matYX, real **matYY)
{
  matXX[0][0] = userData[0];
  matXY[0][0] = 0.0;
  matYX[0][0] = 0.0;
  matYY[0][0] = userData[0];

  matXX[1][1] = userData[1];
  matXY[1][1] = 0.0;
  matYX[1][1] = 0.0;
  matYY[1][1] = userData[1];

  return true;
}
```

References

[1] `ftp://ftp.zib.de/pub/kaskade`

[2] `http://www.zib.de/SciSoft/kaskade`

[3] `http://www.zib.de/SciSoft/kardos`

[4] A.R.A. Anderson, M.A.J. Chaplain, E.L. Newman, R.J.C. Steele, A.M. Thompson: Mathematical modelling of tumor invasion and metastasis, J. of Theor. Medicine, 2000.

[5] R.E. Bank: PLTMG: A Software Package for Solving Elliptic Partial Differential Equations - User's Guide 8.0, SIAM, 1998.

[6] R.E. Bank, R.K. Smith: A Posteriori Error Estimates Based on Hierarchical Bases, SIAM J.Numer. Anal., 30 (1993), 921-935.

[7] R. Beck, B. Erdmann, R. Roitzsch: KASKADE 3.0 - An object-oriented adaptive finite element code, TR-95-04, Konrad-Zuse-Zentrum Berlin, 1995.

[8] R. Beck, B. Erdmann, R. Roitzsch: An Object-Oriented Adaptive Finite Element Code and its Application in Hyperthermia Treatment Planning. In: Erlend Arge, Are Magnus Bruaset und Hans Petter Langtangen (eds.): Modern Software Tools for Scientific Computing. Birkhäuser, 1997.

[9] J.G. Blom, J.G. Verwer: VLUGR3: A Vectorizable Adaptive Grid Solver for PDEs in 3D, I. Algorithmic Aspects and Applications, Appl. Numer.Math. **16** (1994), 129–156.

[10] F. Bornemann: An Adaptive Multilevel Approach to Parabolic Equations I. General Theory and 1D-Implementation, IMPACT Comput. Sci. Engrg. 2, 279-317(1990).

[11] F. Bornemann: An Adaptive Multilevel Approach to Parabolic Equations II. Variable Order Time Discretization Based on a Multiplicative Error Correction, IMPACT Comput. Sci. Engrg. 3, 93-122 (1991).

[12] F. Bornemann, B. Erdmann, R. Roitzsch: KASKADE - Numerical Experiments, TR-91-01, Konrad-Zuse-Zentrum Berlin, 1991.

[13] F. Bornemann: An Adaptive Multilevel Approach to Parabolic Equations III. 2D Error Estimation and Multilevel Preconditioning, IMPACT Comput. Sci. Engrg. 4, 1-45 (1992).

[14] F. Bornemann, B. Erdmann, R. Kornhuber: Adaptive multilevel-methods in three space dimensions, Int. J. Num. Meth.in Eng, Vol. 36, (1993), 3187-3203.

[15] F. Bornemann, B. Erdmann, R. Kornhuber: A Posteriori Error Estimates for Elliptic Problems in Two and Three Space Dimensions, SIAM J. Numer. Anal. 33, 1188-1204 (1996).

[16] R. Codina: A Finite Element Formulation for Viscous Incompressible Flows, Monografia No. 16, Enero 1993, Centro Internacional de Metodos Numericos en Ingenieria, Barcelona, Espana.

[17] D. Braess, P. Deuflhard, K. Lipnikov: A Subspace Cascadic Multigrid Method for Mortar Elements. Preprint SC 99-07, Konrad-Zuse-Zentrum Berlin.

[18] K. Dekker, J.G. Verwer: Stability of Runge–Kutta Methods for Stiff Nonlinear Differential Equations, North-Holland Elsevier Science Publishers,1984.

[19] P. Deuflhard, P. Leinen, H. Yserentant: Concepts of an Adaptive Hierarchical Finite Element Code. IMPACT Comp. Sci. Eng. 1 (1989), 3-35.

[20] P. Deuflhard, Recent Progress in Extrapolation Methods for Ordinary Differential Equations, SIAM Rev. **27** (1985), 505–535.

[21] P. Deuflhard, F. Bornemann: Numerische Mathematik II, Integration Gewöhnlicher Differentialgleichungen, De Gruyter Lehrbuch, Berlin, New York, 1994.

[22] P. Deuflhard, K. Lipnikov: Domain Decomposition with Subdomain CCG for Material Jump Elliptic Problems. In: East-West J. Numer. Math., Vol. 6, No. 2, 81-100 (1998).

[23] P. Deuflhard: Uniqueness Theorems for Stiff ODE Initial Value Problems, in: D.F. Griffiths and G.A. Watson (eds.), Numerical Analysis 1989, Proceedings of the 13th Dundee Conference, Pitman Research Notes in Mathematics Series 228, Longman Scientific and Technical (1990), 74–87.

[24] P. Deuflhard, J. Lang, U. Nowak: Adaptive Algorithms in Dynamical Process Simulation, in: H. Neunzert (ed.), Progress in Industrial Mathematics at ECMI 94, 122-137 (Wiley-Teubner 1996).

[25] B. Erdmann, J. Lang, R. Roitzsch: KASKADE–Manual, Technical Report TR 93-05, Konrad–Zuse–Zentrum Berlin (ZIB), 1993.

[26] B. Erdmann, M. Frei, R.H.W. Hoppe, R. Kornhuber, U. Wiest(1993), Adaptive finite element methods for variational inequalities, East-West J. Numer. Math. 1 (1993), 165-197.

[27] B. Erdmann, J Lang, R. Roitzsch, KASKADE Manual, Version 2.0, TR-93-05, Konrad-Zuse-Zentrum Berlin, 1993.

[28] B. Erdmann, R.H.W. Hoppe, R. Kornhuber, Adaptive multilevel-methods for obstacle problems in three space dimensions, in W. Hackbusch, & G. Wittum (eds.), 'Adaptive Methods - Algorithms, Theory and Applications', Vieweg (1994), 120-141.

[29] B. Erdmann, J. Lang, M. Seebass: Adaptive Solutions of Nonlinear Parabolic Equations with Application to Hyperthermia Treatments, in: Graham de Vahl Davis and Eddie Leonardi (eds.), CHT'97: Advances in Computational Heat Transfer, 103-110, (Cesme, 1997; Begell House Inc., New York 1998).

[30] B. Erdmann, J. Lang, M. Seebass: Optimization of Temperature Distributions for Regional Hyperthermia Based on a Nonlinear Heat Transfer Model, in: K. Diller (ed.), Biotransport: Heat and Mass Transfer in Living Systems, Annals of the New York Academy of Sciences, Vol. 858, 36-46, 1998.

[31] B. Erdmann, J. Lang, R. Roitzsch: Adaptive Linearly Implicit Methods for Instationary Nonlinear Problems, in: Finite Element Methods: Three-dimensional Problems, GAKUTO International Series: Mathematical Sciences and Applications, Vol. 15 (2001), 66-75.

[32] B. Erdmann, C. Kober, J. Lang, P. Deuflhard, H.-F. Zeilhofer, R. Sader: Efficient and Reliable Finite Element Methods for Simulation of the Human Mandible, to appear in Proceedings of Medicine Meets Mathematics, Hartgewebe-Modellierung, Kloster Banz/Staffelstein, April 2001, Report ZIB 01-14 (2001), Konrad-Zuse-Zentrum Berlin.

[33] L.P. Franca, S.L. Frey: Stabilized Finite Element Methods, Comput. Methods Appl. Mech. Engrg. **99** (1992), 209–233.

[34] J. Fröhlich, J. Lang, R. Roitzsch: Selfadaptive Finite Element Computations with Smooth Time Controller and Anisotropic Refinement, in: J.A. Desideri, P.Le. Tallec, E. Onate, J. Periaux, E. Stein (eds.), Numerical Methods in Engineering '96, 523-527 (John Wiley & Sons, New York 1996).

[35] A. Gerisch, J.G. Verwer: Operator splitting and approximate factorization for taxis-diffusion-reaction models, Appl. Numer. Math., Volume 42 (2002), 159-176.

[36] J. Fröhlich, J. Lang: Twodimensional Cascadic Finite Element Computations of Combustion Problems, Comp. Meth. Appl. Mech. Engrg. 158 (1998), 255-267.

[37] K. Gustafsson: Control–Theoretic Techniques for Stepsize Selection in Implicit Runge-Kutta Methods, ACM Trans. Math. Software **20** (1994), 496–517.

[38] K. Gustafsson, M. Lundh, G. Söderlind: A PI Stepsize Control for the Numerical Solution of Ordinary Differential Equations. BIT **28** (1988), 270–287.

[39] E. Hairer, S.P. Nørsett, G. Wanner: Solving Ordinary Differential Equations I, Nonstiff Problems, Springer–Verlag, Berlin, Heidelberg, New York, 1987.

[40] E. Hairer, G. Wanner: Solving Ordinary Differential Equations II, Stiff and Differential-Algebraic Problems, Second Revised Edition, Springer–Verlag, Berlin, Heidelberg, New York, 1996.

[41] S.M. Hassanizadeh, T. Leijnse: On the Modeling of Brine Transport in Porous Media, Water Resources Research **24** (1988), 321–330.

[42] R. Kornhuber, R. Roitzsch: Adaptive Finite-Elemente-Methoden fr konvektionsdominierte Randwertprobleme bei partiellen Differentialgleichungen, in 'Proc. 4. TECFLAM - Seminar', Stuttgart (1988), 103-116.

[43] R. Kornhuber, R. Roitzsch: On adaptive grid refinement in the presence of internal or boundary layers', IMPACT Comput. Sci. Engrg. 2 (1990), 40-72.

[44] R. Kornhuber, R. Roitzsch: Self adaptive computation of the breakdown voltage of planar pn-junctions with multistep field plates, in W. Fichtner & D. Hemmer (eds.), 'Proc. 4th International Conference on Simulation

of Semiconductor Devices and Processes', Hartung-Gorre (1991), 535-543.

[45] R. Kornhuber, R. Roitzsch: Self adaptive finite element simulation of bipolar, strongly reverse biased pn-junctions', Comm. Numer. Meth. Engrg. 9 (1993), 243-250.

[46] J. Lang, A. Walter: A Finite Element Method Adaptive in Space and Time for Nonlinear Reaction-Diffusion-Systems, Impact Comput. Sci. Engrg. 4 (1992), 269-314.

[47] J. Lang: Raum- und zeitadaptive Finite-Elemente-Methoden für Reaktions-Diffusionsgleichungen, in: Modellierung Technischer Flammen, Proc. 8.TECFLAM-Seminar, 115-123 (Darmstadt 1992).

[48] J. Lang, A. Walter: An Adaptive Rothe-Method for Nonlinear Reaction-Diffusion-Systems, Appl. Numer. Math. 13 (1993), 135-146.

[49] J. Lang, A. Walter: An Adaptive Discontinuous Finite Element Method for the Transport Equation, J. Comp. Phys. 117 (1995), 28-34.

[50] J. Lang, J. Fröhlich: Selfadaptive Finite Element Computations of Combustion Problems, in: R.W.Lewis, P.Durbetaki (eds.), Numerical Methods in Thermal Problems Vol. IX, 761-769 (Pineridge Press, Swansea 1995).

[51] J. Lang: Two-Dimensional Fully Adaptive Solutions of Reaction-Diffusion Equations, Appl. Numer. Math. 18 (1995), 223-240.

[52] J. Lang: High-Resolution Self-Adaptive Computations on Chemical Reaction-Diffusion Problems with Internal Boundaries, Chem. Engrg. Sci. 51 (1996), 1055-1070.

[53] J. Lang: Numerical Solution of Reaction-Diffusion Equations, in: F.Keil, W.Mackens, H.Voss, J.Werther (eds.), Scientific Computing in Chemical Engineering, 136-141 (Springer 1996).

[54] J. Lang, B. Erdmann, R. Roitzsch: Three-Dimensional Fully Adaptive Solution of Thermo-Diffusive Flame Propagation Problems, in: R.W. Lewis, J.T.Cross (eds.), Numerical Methods in Thermal Problems, 857-862 (Pineridge Press, Swansea, UK 1997).

[55] J. Lang, B. Erdmann, R. Roitzsch: Adaptive Time-Space Discretization for Combustion Problems, in: A. Sydow (ed.), 15th IMACS World

Congress on Scientific Computation, Modelling and Applied Mathematics, Vol. 2 (Numerical Mathematics), 149-155 (Wissenschaft und Technik Verlag, Berlin, 1997).

[56] J. Lang: Adaptive FEM for Reaction-Diffusion Equations, Appl. Numer. Math. 26 (1998), 105-116.

[57] J. Lang: Adaptive Incompressible Flow Computations with Linearly Implicit Time Discretization and Stabilized Finite Elements, in: K.D. Papailiou, D. Tsahalis, J. Periaux, C. Hirsch, M. Pandolfi (eds.), Computational Fluid Dynamics '98, 200-204 (John Wiley & Sons, New York 1998).

[58] J. Lang, W. Merz: Dynamic Mesh Design Control in Semiconductor Device Simulation, in: V.B. Bajic (ed.), Advances in Systems, Signals, Control and Computers, Vol. 2, 82-86 (Academy of Nonlinear Science, Durban, South Africa, 1998).

[59] J. Lang, B. Erdmann, M. Seebass: Impact of Nonlinear Heat Transfer on Temperature Distribution in Regional Hyperthermia, IEEE Transaction on Biomedical Engineering 46 (1999), 1129-1138.

[60] J. Lang: Adaptive Multilevel Solution of Nonlinear Parabolic PDE Systems. Theory, Algorithm, and Applications, Lecture Notes in Computational Science and Engineering, Vol. 16, 2000, Springer.

[61] J. Lang: Adaptive Multilevel Solutions of Nonlinear Parabolic PDE Systems, in: P. Neittaanmki, T. Tiihonen, P. Tarvainen (eds.), Numerical Mathematics and Advanced Applications, 141-145 (World Scientific, Singapore, New Jersey, London, Hong Kong 2000).

[62] J. Lang: Adaptive Linearly Implicit Methods in Dynamical Process Simulation, CD-ROM Proceedings of the European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS'00, Barcelona, 2000).

[63] J. Lang, B. Erdmann: Adaptive Linearly Implicit Methods for Heat and Mass Transfer, in: A.V. Wouver, P. Saucez, W.E. Schiesser (eds.), Adaptive Metod of Lines, 295-316 (CRC Press, 2000).

[64] J. Lang, J. Verwer: ROS3P - an Accurate Third-Order Rosenbrock Solver Designed for Parabolic Problems, BIT 41 (2001) 730-737.

[65] J. Lang, W. Merz: Two-Dimensional Adaptive Simulation of Dopant Diffusion in Silicon, Computing and Visualization in Science 3 (2001), 169-176.

[66] J. Lang, B. Erdmann: Three-Dimensional Adaptive Computation of Brine Transport in Porous Media, in: G. de Vahl Davis and E. Leonardi (eds.), CHT'01: Advances in Computational Heat Transfer, 1001-1008 (Begell House Inc., New York 2001).

[67] J. Lang, B. Erdmann: Three-Dimensional Adaptive Computation of Brine Transport in Porous Media, (enlarged version), Numerical Heat Transfer: Applications, Vol. 42, No. 1 (2002), 107-119, Taylor & Francis.

[68] M.J. Lourenco, S.C.S. Rosa, C.A. Nieto de Castro, C. Albuquerque, B. Erdmann, J. Lang, R. Roitzsch: Simulation of the Transient Heating in an Unsymmetrical Coated Hot-Strip Sensor with a Self-Adaptive Finite Element Method, in: M.S. Kim and S.T. Ro (eds.), Proc. 5th Asian Thermophysical Properties Conference, Seoul, 1998, Vol. 1, 91-94 (Seoul National University, 1998).

[69] M.J. Lourenco, S.C.S. Rosa, C.A. Nieto de Castro, C. Albuquerque, B. Erdmann, J. Lang, R. Roitzsch: Simulation of the Transient Heating in an Unsymmetrical Coated Hot-Strip Sensor with a Self-Adaptive Finite Element Method, Int. J. Thermophysics 21 (2000) 377-384.

[70] G. Lube, D. Weiss: Stabilized Finite Element Methods for Singularly Perturbed Parabolic Problems, Appl. Numer. Math. **17** (1995), 431–459.

[71] Ch. Lubich, M. Roche: Rosenbrock Methods for Differential–Algebraic Systems with Solution–Dependent Singular Matrix Multiplying the Derivative, Comput., 43 (1990), 325–342.

[72] J.D. Murray: A Pre-pattern Formation Mechanism for Animal Coat Markings, J. theor. Biol., 88 (1981), 161-198.

[73] M. Roche: Runge–Kutta and Rosenbrock Methods for Differential–Algebraic Equations and Stiff ODEs, PhD thesis, Université de Genève, 1988.

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

[75] W. Merz, J. Lang: Analysis and Simulation of Two-Dimensional Dopant Diffusion in Silicon, in: K.-H.Hoffmann (ed.), Smart Materials, Proceedings of the First Caesarium, Springer-Verlag, 2001.

[76] W. Mittelbach, H.-M. Henning: Seasonal Heat Storage Using Adsorption Processes, in IEA Workshop Advanced Solar Thermal Storage Systems, Helsinki, 1997.

[77] J.E. Pearson: Complex Patterns in a Simple System, Science, Vol. 261 (1993), 189-192.

[78] H.H. Pennes: Analysis of tissue and arterial blood temperatures in the resting human forearm, J. Appl. Phys. **1** (1948), 1299–1306.

[79] R. Roitzsch, R. Kornhuber: BOXES, a programm to generate triangulations from a rectangular domain description, Technical Report TR 90-9 (1990), Konrad-Zuse-Zentrum Berlin.

[80] R. Roitzsch, B. Erdmann, J. Lang: The Benefits of Modularization: from KASKADE to KARDOS, Report, SC-98-15 (1998), Konrad-Zuse-Zentrum Berlin.

[81] H.H. Rosenbrock: Some General Implicit Processes for the Numerical Solution of Differential Equations, Computer J. (1963), 329–331.

[82] W. Ruppel, Entwicklung von Simulationsverfahren für die Reaktionstechnik, manuscript, BASF research, 1993.

[83] M. Schäfer, S. Turek: Benchmark Computations of Laminar Flow Around a Cylinder, Preprint 96–03 (SFB 359), IWR Heidelberg, 1996.

[84] D. Schumann: Eine anwendungsbezogene Einfhrung in die adaptive Finite Elemente Methode, Diplomarbeit, Tech. Fachhochschule Berlin, 2001.

[85] F. Seewald, A. Pollei, M. Kraume, W. Mittelbach, J. Lang: Numerical Calculation of the Heat Transfer in an Adsorption Energy Storage with KARDOS, Report SC-99-04 (1999), Konrad-Zuse-Zentrum Berlin.

[86] G. Steinebach: Order–Reduction of ROW–methods for DAEs and Method of Lines Applications, Preprint 1741, Technische Hochschule Darmstadt, Germany, 1995.

[87] K. Strehmel, R. Weiner: Linear–implizite Runge–Kutta–Methoden und ihre Anwendungen, Teubner Texte zur Mathematik 127, Teubner Stuttgart, Leipzig, 1992.

[88] L. Tobiska, R. Verfürth: Analysis of a Streamline Diffusion Finite Element Method for the Stokes and Navier–Stokes Equation, SIAM J. Numer. Anal. **33** (1996), 107–127.

[89] R.A. Trompert, J.G. Verwer, J.G. Blom: Computing Brine Transport in Porous Media with an Adaptive–Grid Method, Int. J. Numer. Meth. Fluids **16** (1993), 43–63.

[90] H.A. van der Vorst: BI–CGSTAB: A fast and smoothly converging variant of BI–CG for the solution of nonsymmetric linear systems, SIAM J. Sci. Stat. **13** (1992), 631–644.

[91] D. Stalling, M. Zöckler, H.-C. Hege: AMIRA – Advanced Visualization, Data Analysis and Geometry Reconstruction. `http://amira.zib.de`.

Appendix
Implementation of Examples of Use

In the Section 3 we made the reader familiar to a lot of applications which we treated with KARDOS. In this appendix we complete these examples by showing how we brought them into the code.

Determination of Thermal Conductivity

```
static int SensorParabolic(real x, real y, int classA, real t,
                           real *u, real *ux, real *uy,  real **C)
{
   real p = 0.0;

   switch (classA)
   {
      case 1: p = 2852850.0; break;
      case 2: p = 3280394.0; break;
      case 3: p = 3418205.0; break;
      case 4: p = 4164469.0; break;
   }

   C[0][0] = p;

   return true;
}


static int SensorParabolicStruct(int **structC, int **dependsS,
                                 int *dependsT, int *dependsU,
                                 int *dependsGradU)
{
  structC[0][0]   = F_FILL;
  dependsS[0][0] = false;

  *dependsT     = false;
  *dependsU     = false;
  *dependsGradU = false;

  return true;
```

```
}


static int SensorLaplace(real x, real y, int classA, real t,
                         real *u, real *ux, real *uy,
                         real **matXX, real **matXY,
                         real **matYX, real **matYY)
{
   real e = 0.0;

   switch (classA)
   {
      case 1: e = 72.0;  break;
      case 2: e = 25.5;  break;
      case 3: e = 32.3;  break;
      case 4: e = 0.606; break;
   }

   matXX[0][0] = e;
   matXY[0][0] = 0.0;
   matYX[0][0] = 0.0;
   matYY[0][0] = e;

   return true;
}


static int SensorLaplaceStruct(int **structD,
                               int **dependsS, int *dependsT,
                               int *dependsU,  int *dependsGradU)
{
  structD[0][0] = F_FILL;    dependsS[0][0] = false;

  *dependsT     = false;
  *dependsU     = false;
  *dependsGradU = false;

  return true;
}

static int SensorSource(real x, real y, int classA, real t,
```

```
                        real *u, real *ux, real *uy, real *vec)
{
    real s = 0.0;

    switch (classA)
    {
      case 1: s = 170.45e+9; break;
      case 2: s = 0.0; break;
      case 3: s = 0.0; break;
      case 4: s = 0.0; break;
    }

    vec[0] = s;
    return true;
}


static int SensorSourceStruct(int *structF, int *dependsS,
                              int *dependsT, int *dependsU,
                              int *dependsGradU)
{
  structF[0]   = F_FILL;   dependsS[0]  = false;

  *dependsT     = false;
  *dependsU     = false;
  *dependsGradU = false;

  return true;
}


static int SensorInitialFunc(real x, real y, int classA,
                             real *start, real *startUt)
{
  start[0]   = 0.0;

  return true;
}


static int SensorDirichlet(real x, real y, int classA, real t,
```

```
                          real *u, int equation, real *bVal)
{
  bVal[0] = u[0];

  return true;
}

static char *tempVarName = {``temperature''};

int SetSensorProblems()
{
  if (!SetTimeProblem("sensor",tempVarName,
                              SensorParabolic,
                              SensorParabolicStruct,
                              SensorLaplace,
                              SensorLaplaceStruct,
                              0,
                              0,
                              0,
                              0,
                              SensorSource,
                              SensorSourceStruct,
                              0,
                              0,
                              SensorInitialFunc,
                              0,
                              SensorDirichlet,
                              0)) return false;

  return true;
}
```

Pattern Formation

```
static
char *tempVarName[] = {"concentration0","concentration1"};

static int GrayScottParabolic(real x, real y, int classA, real t,
                              real *u, real *ux, real *uy,
                              real **mat)
{
  mat[0][0] = 1.0;
  mat[1][1] = 1.0;

  return true;
}

static int GrayScottParabolicStruct(int **structM, int **dependsXY,
                                    int *dependsT, int *dependsU,
                                    int *dependsGradU)
{
  structM[0][0] = F_FILL;   dependsXY[0][0] = false;
  structM[0][1] = F_IGNORE; dependsXY[0][1] = false;
  structM[1][0] = F_IGNORE; dependsXY[1][0] = false;
  structM[1][1] = F_FILL;   dependsXY[1][1] = false;

  *dependsT = false;
  *dependsU = false;
  *dependsGradU = false;

  return true;
}

static int GrayScottLaplace(real x, real y, int classA, real t,
                            real *u, real *ux, real *uy,
                            real **matXX, real **matXY,
                            real **matYX, real **matYY)
{
  matXX[0][0] = 2.0e-5/0.25;
  matXY[0][0] = 0.0;
  matYX[0][0] = 0.0;
  matYY[0][0] = 2.0e-5/0.25;
```

```
  matXX[1][1] = 1.0e-5/0.25;
  matXY[1][1] = 0.0;
  matYX[1][1] = 0.0;
  matYY[1][1] = 1.0e-5/0.25;

  return true;
}

static int GrayScottLaplaceStruct(int **structM, int **dependsXY,
                                  int *dependsT, int *dependsU,
                                  int *dependsGradU)
{
  structM[0][0] = F_FILL;   dependsXY[0][0] = false;
  structM[0][1] = F_IGNORE; dependsXY[0][1] = false;
  structM[1][0] = F_IGNORE; dependsXY[1][0] = false;
  structM[1][1] = F_FILL;   dependsXY[1][1] = false;

  *dependsT = false;
  *dependsU = false;
  *dependsGradU = false;

  return true;
}

static int GrayScottSource(real x, real y, int classA, real t,
                           real *u, real *ux, real *uy,
                           real *vec)
{
  vec[0] =  -u[0]*u[1]*u[1] + 0.024*(1.0-u[0]);
  vec[1] =   u[0]*u[1]*u[1] - (0.024+0.06)*u[1];

  return true;
}

static int GrayScottSourceStruct(int *structV, int *dependsXY,
                                 int *dependsT, int *dependsU,
                                 int *dependsGradU)
{
  structV[0] = F_FILL;   dependsXY[0] = true;
  structV[1] = F_FILL;   dependsXY[1] = true;
```

```
  *dependsT = false;
  *dependsU = true;
  *dependsGradU = true;

  return true;
}

static int GrayScottInitialFunc(real x, real y, int classA,
                                real *start, real *dummy)
{
  if ((0.25<=x)&&(x<=0.75)&&(0.25<=y)&&(y<=0.75))
    {
      start[0] = 1.0
               - 0.5*pow(sin(4.0*REALPI*x),2.0)
                    *pow(sin(4.0*REALPI*y),2.0);
      start[1] = 0.25*pow(sin(4.0*REALPI*x),2.0)
                     *pow(sin(4.0*REALPI*y),2.0);
    }
  else
    {
      start[0] = 1.0;
      start[1] = 0.0;
    }

  return true;
}


static int GrayScottDirichlet(real x, real y, int classA, real t,
                              real *u, int variable, real *fVal)
{
  switch (variable)
   {
    case 0: fVal[0] = u[0] - 1.0;
            break;
    case 1: fVal[0] = u[1] - 0.0;
            break;
   }
  return true;
}
```

```
int SetGrayScottProblem()
{
  if (!SetTimeProblem("gray_scott",tempVarName,
                                    GrayScottParabolic,
                                    GrayScottParabolicStruct,
                                    GrayScottLaplace,
                                    GrayScottLaplaceStruct,
                                    0,
                                    0,
                                    GrayScottSource,
                                    GrayScottSourceStruct,
                                    0,
                                    0,
                                    GrayScottInitialFunc,
                                    0,
                                    GrayScottDirichlet,
                                    0)) return false;

  return true;
}
```

Thermo-Diffusive Flames

```
static char *flameVarNames[] = {"temperature","concentration"};

static int ThinFlameParabolic(real x, real y, int classA, real t,
                              real *u, real *ux, real *uy,
                              real **mat)
{
  mat[0][0] = 1.0;
  mat[1][1] = 1.0;

  return true;
}

static int ThinFlameParabolicStruct(int **structM, int **dependsXY,
                                    int *dependsT, int *dependsU,
                                    int *dependsGradU)
{
  structM[0][0] = F_FILL;   dependsXY[0][0] = false;
  structM[0][1] = F_IGNORE; dependsXY[0][1] = false;
  structM[1][0] = F_IGNORE; dependsXY[1][0] = false;
  structM[1][1] = F_FILL;   dependsXY[1][1] = false;

  *dependsT = false;
  *dependsU = false;
  *dependsGradU = false;

  return true;
}

static int ThinFlameLaplace(real x, real y, int classA, real t,
                            real *u, real *ux, real *uy,
                            real **matXX, real **matXY,
                            real **matYX, real **matYY)
{
  matXX[0][0] = 1.0;
  matXY[0][0] = 0.0;
  matYX[0][0] = 0.0;
  matYY[0][0] = 1.0;

  matXX[1][1] = 1.0/flameCoeff[0];
```

```
  matXY[1][1] = 0.0;
  matYX[1][1] = 0.0;
  matYY[1][1] = 1.0/flameCoeff[0];

  return true;
}

static int ThinFlameLaplaceStruct(int **structM, int **dependsXY,
                                  int *dependsT, int *dependsU,
                                  int *dependsGradU)
{
  structM[0][0] = F_FILL;   dependsXY[0][0] = false;
  structM[0][1] = F_IGNORE; dependsXY[0][1] = false;
  structM[1][0] = F_IGNORE; dependsXY[1][0] = false;
  structM[1][1] = F_FILL;   dependsXY[1][1] = false;

  *dependsT = false;
  *dependsU = false;
  *dependsGradU = false;

  return true;
}

static int ThinFlameSource(real x, real y, int classA, real t,
                           real *u, real *ux, real *uy, real *vec)
{
  real help1, help2;

  help1 = 1.0-u[0];
  help2 = flameCoeff[1]*flameCoeff[1]*u[1]*
          exp(-flameCoeff[1]*help1/(1.0-flameCoeff[2]*help1))
          *HALF/flameCoeff[0];

  vec[0] =  help2;
  vec[1] = -help2;

  return true;
}

static int ThinFlameSourceStruct(int *structV, int *dependsXY,
                                 int *dependsT, int *dependsU,
```

```
                                  int *dependsGradU)
{
  structV[0] = F_FILL;   dependsXY[0] = true;
  structV[1] = F_FILL;   dependsXY[1] = true;

  *dependsT = false;
  *dependsU = true;
  *dependsGradU = false;

  return true;
}

static int ThinFlameJacobian(real x, real y, int classA,
                             real t,  real *u, real *ux,
                             real *uy, real **mat)
{
  real help0, help2, val0, val1;

  help0 = 1.0-u[0];
  help2 = 1.0-flameCoeff[2]*help0;
  val0  = flameCoeff[1]*flameCoeff[1]*flameCoeff[1]*u[1]*
            exp(-flameCoeff[1]*help0/help2)*HALF
            /(flameCoeff[0]*help2*help2);
  val1  = flameCoeff[1]*flameCoeff[1]
          *exp(-flameCoeff[1]*help0/(1.0-flameCoeff[2]*help0))
          *HALF/flameCoeff[0];

  mat[0][0] =  val0;
  mat[0][1] =  val1;
  mat[1][0] = -val0;
  mat[1][1] = -val1;

  return true;
}

static int ThinFlameJacobianStruct(int **structM,
                                   int **dependsXY)
{
  structM[0][0] = F_FILL;   dependsXY[0][0] = true;
  structM[0][1] = F_FILL;   dependsXY[0][1] = true;
  structM[1][0] = F_FILL;   dependsXY[1][0] = true;
```

```
  structM[1][1] = F_FILL;   dependsXY[1][1] = true;

  return true;
}

static int HandFlameInitialFunc(real x, real y, int classA,
                                real *start, real *dummy)
{
  real x0 = flameCoeff[5]*0.15,
       length = flameCoeff[5];

  if ((0.0<=x)&&(x<=x0))
    {
      start[0] = 1.0;
      start[1] = 0.0;
    }
  else if ((x0<x)&&(x<=length))
    {
      start[0] = exp(-x+x0);
      start[1] = 1.0-exp(flameCoeff[0]*(-x+x0));
    }
  else return false;

  return true;
}

static int HandFlameDirichlet(real x, real y, int classA,
                              real t, real *u,
                              int variable, real *fVal)
{
  switch (variable)
    {
      case 0: if (x==0.0) fVal[0] = u[0] - 1.0;
              else if (x==flameCoeff[5]) fVal[0] = u[0];
              else ZIBStdOut("error in HandFlameDirichlet\n");
              break;
      case 1: if (x==0.0) fVal[0] = u[1];
              else if (x==flameCoeff[5]) fVal[0] = u[1] - 1.0;
              else ZIBStdOut("error in HandFlameDirichlet\n");
              break;
    }
```

```
  return true;
}

static int HandFlameCauchy(real x, real y, int classA, real t,
                           real *u, int equation, real *fVal)
{
  switch (equation)
    {
       case 0: if (  (x!=0.0)
                   &&(x!=flameCoeff[5])&&(y!=flameCoeff[6])
                   &&(y!=-flameCoeff[6]))
                  fVal[0] = -flameCoeff[3]*u[0];
               else ZIBStdOut("error in HandFlameCauchy\n");
               break;
       case 1: ZIBStdOut("error in HandFlameCauchy\n");
               break;
    }

  return true;
}


int SetFlameProblems()
{
   if (!SetTimeProblem("handflame",flameVarNames,
                                   ThinFlameParabolic,
                                   ThinFlameParabolicStruct,
                                   ThinFlameLaplace,
                                   ThinFlameLaplaceStruct,
                                   0,
                                   0,
                                   ThinFlameSource,
                                   ThinFlameSourceStruct,
                                   ThinFlameJacobian,
                                   ThinFlameJacobianStruct,
                                   HandFlameInitialFunc,
                                   HandFlameCauchy,
                                   HandFlameDirichlet,
                                   0)) return false;
```

```
    return true;
}
```

```
static char *hyperVarName[] = {"temperature"};

static real temp_bolus = 25.0;
static real temp_air = 25.0;
static real transportAir  = 15.0;
static real transportWater  = 45.0;


static int HyperParabolic(real x, real y, real z, int classA,
                          real t, real *u, real *ux, real *uy,
  real *uz, real **mat)
{
  mat[0][0]        =  rc_tissue[classA]*scaleTime;

  return true;
}

static int HyperParabolicStruct(int **structM, int **dependsS,
                                int *dependsT, int *dependsU,
                                int *dependsGradU)
{
  structM[0][0] = F_FILL; dependsS[0][0] = false;

  *dependsT = false;
  *dependsU = false;
  *dependsGradU = false;

  return true;
}

static int HyperLaplace(real x, real y, real z, int classA,
                        real t, real *u,
                        real *ux, real *uy, real *uz,
                        real **matXX, real **matXY, real **matXZ,
                        real **matYX, real **matYY, real **matYZ,
                        real **matZX, real **matZY, real **matZZ)
{
  matXX[0][0] =  kappa_tissue[classA];
  matXY[0][0] =  0.0;
```

```
  matXZ[0][0] =  0.0;
  matYX[0][0] =  0.0;
  matYY[0][0] =  kappa_tissue[classA];
  matYZ[0][0] =  0.0;
  matZX[0][0] =  0.0;
  matZY[0][0] =  0.0;
  matZZ[0][0] =  kappa_tissue[classA];

  return true;
}

static int HyperLaplaceStruct(int **structM, int **dependsS,
                              int *dependsT,int *dependsU,
                              int *dependsGradU)
{
  structM[0][0] = F_FILL; dependsS[0][0] = false;

  *dependsT = false;
  *dependsU = false;
  *dependsGradU = false;

  return true;
}

static int HyperSource(real x, real y, real z, int classA,
                       real t, real *u, real *ux, real *uy,
                       real *uz, real *vec)
{
  /*  SAR-scaling: 0.181, 0.11342, 0.284   */
  real sarScale = hyper_data[15];

  vec[0] = sarScale*SAR(x,y,z);

  return true;
}

static int HyperSourceStruct(int *structV, int *dependsS,
                             int *dependsT, int *dependsU,
                             int *dependsGradU)
{
  structV[0] = F_FILL;   dependsS[0] = true;
```

```
  *dependsT = true;
  *dependsU = true;
  *dependsGradU = false;

  return true;
}

static int HyperInitial(real x, real y, real z, int classA,
                        real *start, real *startUt)
{
  start[0]   =   hyper_data[14];

  return true;
}

static int HyperCauchy(real x, real y, real z, int classA,
                       real t, real *u, int equation,
                       real *fVal)
{
  if (classA==3)
    fVal[0] = transportWater*(temp_bolus - u[0]);
  else
    if (classA==5 || classA==2)
      fVal[0] = transportAir*(temp_air - u[0]);
    else
      printf("\n undefined boundary condition");

  return true;
}

static int HyperDirichlet(real x, real y, real z, int classA,
                          real t, real *u, int equation,
                          real *fVal)
{
  fVal[0]= 37.0-u[0];

  return true;
}
```

```
int SetHyperProblems()
{
  if (!SetTimeProblem("hyper",hyperVarName,
      HyperParabolic,
      HyperParabolicStruct,
      HyperLaplace,
      HyperLaplaceStruct,
      0,
      0,
      HyperSource,
      HyperSourceStruct,
      0,
      0,
      HyperInitial,
      HyperCauchy,
      HyperDirichlet,
      0)) return false;

    return true;
}
```

Tumour Invasion

```
#define CELL_KAPPA        0.7
#define CELL_ALPHA       10.0
#define CELL_BETA         4.0
#define CELL_GAMMA        1.0
#define CELL_LAMBDA       1.0
#define CELL_CSTAR        0.2
#define CELL_MU         100.0
#define CELL_EPSILON      0.001


static
char *cellVarNames[] = {"density of blood cells","concentration taf"};
static
char *timVarNames[] =  {"density of tumor cells","ecm","mde"};



/*#################################################################
  #
  #              Example: Tumor angiogenesis model
  #
  #                   u[0]: density
  #                   u[1]: taf
  #
  #################################################################
*/

static int CellParabolic(real x, real y, int classA, real t,
                         real *u, real *ux, real *uy,
                         real **mat)
{
  mat[0][0] = 1.0;
  mat[1][1] = 1.0;

  return true;
}

static int CellParabolicStruct(int **structM, int **dependsXY,
                               int *dependsT, int *dependsU,
```

```
                                int *dependsGradU)
{
  structM[0][0] = F_FILL;   dependsXY[0][0] = false;
  structM[0][1] = F_IGNORE; dependsXY[0][1] = false;
  structM[1][0] = F_IGNORE; dependsXY[1][0] = false;
  structM[1][1] = F_FILL;   dependsXY[1][1] = false;

  *dependsT = false;
  *dependsU = false;
  *dependsGradU = false;

  return true;
}

static int CellLaplace(real x, real y, int classA, real t,
                       real *u, real *ux, real *uy,
                       real **matXX, real **matXY,
                       real **matYX, real **matYY)
{
  matXX[0][0] = CELL_EPSILON;
  matXY[0][0] = 0.0;
  matYX[0][0] = 0.0;
  matYY[0][0] = CELL_EPSILON;

  matXX[0][1] = -u[0]*CELL_KAPPA;
  matXY[0][1] = 0.0;
  matYX[0][1] = 0.0;
  matYY[0][1] = -u[0]*CELL_KAPPA;

  matXX[1][1] = 1.0;
  matXY[1][1] = 0.0;
  matYX[1][1] = 0.0;
  matYY[1][1] = 1.0;

  return true;
}

static int CellLaplaceStruct(int **structM, int **dependsXY,
                             int *dependsT, int *dependsU,
                             int *dependsGradU)
{
```

```
  structM[0][0] = F_FILL;   dependsXY[0][0] = false;
  structM[0][1] = F_FILL;   dependsXY[0][1] = true;
  structM[1][0] = F_IGNORE; dependsXY[1][0] = false;
  structM[1][1] = F_FILL;   dependsXY[1][1] = false;

  *dependsT = false;
  *dependsU = true;
  *dependsGradU = false;

  return true;
}

static int CellSource(real x, real y, int classA, real t,
                      real *u, real *ux, real *uy, real *vec)
{
  vec[0] =  CELL_MU*u[0]*(1.0-u[0])*MAXIMUM(0.0,u[1]-CELL_CSTAR)
           -CELL_BETA*u[0];
  vec[1] = -CELL_LAMBDA*u[1]
           -CELL_ALPHA*u[0]*u[1]/(CELL_GAMMA+u[1]);

  return true;
}

static int CellSourceStruct(int *structV, int *dependsXY,
                            int *dependsT, int *dependsU,
                            int *dependsGradU)
{
  structV[0] = F_FILL;   dependsXY[0] = true;
  structV[1] = F_FILL;   dependsXY[1] = true;

  *dependsT = false;
  *dependsU = false;
  *dependsGradU = false;

  return true;
}

static int CellInitialFunc(real x, real y, int classA, real *start,
                           real *dummy)
{
  if ( (x >= 0.95) && ( (y>=0.2 && y<=0.27)
```

```
                  ||   (y>=0.5 && y<=0.57) ||
                       (y>=0.7 && y<=0.77) ))
    {
      start[0]=1.0;
    }
  else
    {
      start[0] = 0.0;
    }


  start[1] = cos(REALPI/2.0*x)*(2.0*(1.0-x)+2.0+
             cos (2.0*REALPI*(0.5-y)))/5.0
            *exp((-1.0)*(1.0-cos(REALPI/2.0*x)));

  return true;
}

static int CellDirichlet(real x, real y, int classA, real t,
                         real *u, int equation, real *fVal)
{
  real u0[2], ut0[2];

  CellInitialFunc(x, y, classA, u0, ut0);

  switch (equation)
    {
      case 0: *fVal = u0[0]-u[0];
              break;
      case 1: *fVal = u0[1]-u[1];
              break;
    }

  return true;
}


/*
  ######################################################################
  #
  #        Example: Tumor invasion model
```

```
  #
  #         u[0]: tumor cell density
  #         u[1]: density of the extracellular matrix, ECM
  #         u[2]: density of the matrix degradative enzymes, MDE
  #
  ################################################################
*/


static int TIMParabolic(real x, real y, int classA, real t, real *u,
                        real *ux, real *uy,  real **mat)
{
  mat[0][0] = 1.0;
  mat[1][1] = 1.0;
  mat[2][2] = 1.0;

  return true;
}

static int TIMParabolicStruct(int **structM, int **dependsXY,
                              int *dependsT, int *dependsU,
                              int *dependsGradU)
{
  structM[0][0] = F_FILL;   dependsXY[0][0] = false;
  structM[1][1] = F_FILL;   dependsXY[1][1] = false;
  structM[2][2] = F_FILL;   dependsXY[2][2] = false;

  *dependsT = false;
  *dependsU = false;
  *dependsGradU = false;

  return true;
}

static int TIMLaplace(real x, real y, int classA, real t,
                      real *u, real *ux, real *uy,
                      real **matXX, real **matXY,
                      real **matYX, real **matYY)
{
  matXX[0][0] = 0.1*CELL_EPSILON;
  matXY[0][0] = 0.0;
```

```
  matYX[0][0] = 0.0;
  matYY[0][0] = 0.1*CELL_EPSILON;

  matXX[0][1] = -u[0]*0.005;
  matXY[0][1] = 0.0;
  matYX[0][1] = 0.0;
  matYY[0][1] = -u[0]*0.005;

  matXX[2][2] = 0.001;
  matXY[2][2] = 0.0;
  matYX[2][2] = 0.0;
  matYY[2][2] = 0.001;

  return true;
}

static int TIMLaplaceStruct(int **structM, int **dependsXY,
                            int *dependsT, int *dependsU,
                            int *dependsGradU)
{
  structM[0][0] = F_FILL;   dependsXY[0][0] = false;
  structM[0][1] = F_FILL;   dependsXY[0][1] = true;
  structM[2][2] = F_FILL;   dependsXY[2][2] = false;

  *dependsT = false;
  *dependsU = true;
  *dependsGradU = false;

  return true;
}

static int TIMSource(real x, real y, int classA, real t,
                     real *u, real *ux, real *uy, real *vec)
{
  vec[0] =  0.0;
  vec[1] = -10.0*u[1]*u[2];
  vec[2] =  0.1*u[0];

  return true;
}
```

```
static int TIMSourceStruct(int *structV, int *dependsXY,
                           int *dependsT, int *dependsU,
                           int *dependsGradU)
{
  structV[1] = F_FILL;     dependsXY[1] = true;
  structV[2] = F_FILL;     dependsXY[2] = true;

  *dependsT = false;
  *dependsU = true;
  *dependsGradU = false;

  return true;
}

static int TIMInitialFunc(real x, real y, int classA, real *start,
                          real *dummy)
{
  real qdist = (x-0.5)*(x-0.5)+(y-0.5)*(y-0.5),
       dist  = sqrt(qdist);

  start[0] = exp(-qdist/0.0025);
  start[1] =  0.5+0.5
              *sin(10.0*x*(dist+0.1)*REALPI/(y+1.0))
              *sin(10.0*x*(dist+0.1)*REALPI*y)
              *sin(10.0*(1.0-x)*(dist+0.1)*REALPI/(y+1.0))
              *sin(10.0*(x-1.0)*(dist+0.1)*REALPI*(y-1.0));
  start[2] = exp(-qdist/0.0025);

  return true;
}

static int TIMDirichlet(real x, real y, int classA, real t,
                        real *u, int equation, real *fVal)
{
  switch (equation)
    {
      case 0: *fVal = -u[0];
              break;
    }

  return true;
```

```
}

static int TIMSolution(real x, real y, int classA, real t,
                       real *u, real *ux, real *uy)
{
  u[0]=0.0;
  u[1]=0.0;
  u[2]=0.0;

  ux[0]=0.0;
  ux[1]=0.0;
  ux[2]=0.0;

  uy[0]=0.0;
  uy[1]=0.0;
  uy[2]=0.0;

  return true;
}


int SetCellProblems()
{
  if (!SetTimeProblem("cell",cellVarNames,
                       CellParabolic,
                       CellParabolicStruct,
                       CellLaplace,
                       CellLaplaceStruct,
                       0,
                       0,
                       CellSource,
                       CellSourceStruct,
                       0,
                       0,
                       CellInitialFunc,
                       0,
                       CellDirichlet,
                       0))  return false;

  if (!SetTimeProblem("tim",timVarNames,
                       TIMParabolic,
```

```
                        TIMParabolicStruct,
                        TIMLaplace,
                        TIMLaplaceStruct,
                        0,
                        0,
                        TIMSource,
                        TIMSourceStruct,
                        0,
                        0,
                        TIMInitialFunc,
                        0,
                        TIMDirichlet,
                        TIMSolution)) return false;
  return true;
}
```

```
static char *elastVarName[] = {"u","v","w"};


static int ElastParabolic(real x, real y, real z, int classA,
                          real t, real *u, real *ux, real *uy,
                          real *uz, real **mat)
{
  mat[0][0] = 1.0;
  mat[1][1] = 1.0;
  mat[2][2] = 1.0;

  return true;
}

static int ElastParabolicStruct(int **structM, int **dependsS,
                                int *dependsT, int *dependsU,
                                int *dependsGradU)
{
  structM[0][0] = F_FILL;   dependsS[0][0] = false;
  structM[0][1] = F_IGNORE; dependsS[0][1] = false;
  structM[0][2] = F_IGNORE; dependsS[0][2] = false;
  structM[1][0] = F_IGNORE; dependsS[1][0] = false;
  structM[1][1] = F_FILL;   dependsS[1][1] = false;
  structM[1][2] = F_IGNORE; dependsS[1][2] = false;
  structM[2][0] = F_IGNORE; dependsS[2][0] = false;
  structM[2][1] = F_IGNORE; dependsS[2][1] = false;
  structM[2][2] = F_FILL;   dependsS[2][2] = false;

  *dependsT     = false;
  *dependsU     = false;
  *dependsGradU = false;

  return true;
}


static int ElastLaplace(real x, real y,  real z, int classA,
                        real t, real *u,
                        real *ux, real *uy, real *uz,
```

```
                        real **matXX, real **matXY, real **matXZ,
                        real **matYX, real **matYY, real **matYZ,
                        real **matZX, real **matZY, real **matZZ)
{
  real lambda, my;

  lambda = tissue[classA].lambda;
  my     = tissue[classA].mu;

  matXX[0][0] = lambda + 2.0*my;
  matXY[0][0] = 0.0;
  matXZ[0][0] = 0.0;
  matYX[0][0] = 0.0;
  matYY[0][0] = my;
  matYZ[0][0] = 0.0;
  matZX[0][0] = 0.0;
  matZY[0][0] = 0.0;
  matZZ[0][0] = my;

  matXX[0][1] = 0.0;
  matXY[0][1] = lambda;
  matXZ[0][1] = 0.0;
  matYX[0][1] = my;
  matYY[0][1] = 0.0;
  matYZ[0][1] = 0.0;
  matZX[0][1] = 0.0;
  matZY[0][1] = 0.0;
  matZZ[0][1] = 0.0;

  matXX[0][2] = 0.0;
  matXY[0][2] = 0.0;
  matXZ[0][2] = lambda;
  matYX[0][2] = 0.0;
  matYY[0][2] = 0.0;
  matYZ[0][2] = 0.0;
  matZX[0][2] = my;
  matZY[0][2] = 0.0;
  matZZ[0][2] = 0.0;

  matXX[1][0] = 0.0;
  matXY[1][0] = my;
```

```
matXZ[1][0] = 0.0;
matYX[1][0] = lambda;
matYY[1][0] = 0.0;
matYZ[1][0] = 0.0;
matZX[1][0] = 0.0;
matZY[1][0] = 0.0;
matZZ[1][0] = 0.0;

matXX[1][1] = my;
matXY[1][1] = 0.0;
matXZ[1][1] = 0.0;
matYX[1][1] = 0.0;
matYY[1][1] = lambda+2.0*my;
matYZ[1][1] = 0.0;
matZX[1][1] = 0.0;
matZY[1][1] = 0.0;
matZZ[1][1] = my;

matXX[1][2] = 0.0;
matXY[1][2] = 0.0;
matXZ[1][2] = 0.0;
matYX[1][2] = 0.0;
matYY[1][2] = 0.0;
matYZ[1][2] = lambda;
matZX[1][2] = 0.0;
matZY[1][2] = my;
matZZ[1][2] = 0.0;

matXX[2][0] = 0.0;
matXY[2][0] = 0.0;
matXZ[2][0] = my;
matYX[2][0] = 0.0;
matYY[2][0] = 0.0;
matYZ[2][0] = 0.0;
matZX[2][0] = lambda;
matZY[2][0] = 0.0;
matZZ[2][0] = 0.0;

matXX[2][1] = 0.0;
matXY[2][1] = 0.0;
matXZ[2][1] = 0.0;
```

```
  matYX[2][1] = 0.0;
  matYY[2][1] = 0.0;
  matYZ[2][1] = my;
  matZX[2][1] = 0.0;
  matZY[2][1] = lambda;
  matZZ[2][1] = 0.0;

  matXX[2][2] = my;
  matXY[2][2] = 0.0;
  matXZ[2][2] = 0.0;
  matYX[2][2] = 0.0;
  matYY[2][2] = my;
  matYZ[2][2] = 0.0;
  matZX[2][2] = 0.0;
  matZY[2][2] = 0.0;
  matZZ[2][2] = lambda+2.0*my;

  return true;
}

static int ElastLaplaceStruct(int **structM, int **dependsS,
                              int *dependsT, int *dependsU,
                              int *dependsGradU)
{
  structM[0][0] = F_FILL;   dependsS[0][0] = true;
  structM[0][1] = F_FILL;   dependsS[0][1] = true;
  structM[0][2] = F_FILL;   dependsS[0][2] = true;
  structM[1][0] = F_FILL;   dependsS[1][0] = true;
  structM[1][1] = F_FILL;   dependsS[1][1] = true;
  structM[1][2] = F_FILL;   dependsS[1][2] = true;
  structM[2][0] = F_FILL;   dependsS[2][0] = true;
  structM[2][1] = F_FILL;   dependsS[2][1] = true;
  structM[2][2] = F_FILL;   dependsS[2][2] = true;

  *dependsT     = false;
  *dependsU     = false;
  *dependsGradU = false;

  return true;
}
```

```
static int ElastSource(real x, real y, real z, int classA,
                       real t, real *u, real *ux, real *uy,
                       real *uz, real *vec)
{
  vec[0] = 0.0;
  vec[1] = 0.0;
  vec[2] = 0.0;

  return true;
}

static int ElastSourceStruct(int *structV, int *dependsS,
                             int *dependsT, int *dependsU,
                             int *dependsGradU)
{
  structV[0] = F_IGNORE;   dependsS[0] = false;
  structV[1] = F_IGNORE;   dependsS[1] = false;
  structV[2] = F_IGNORE;   dependsS[2] = false;

  *dependsT     = false;
  *dependsU     = false;
  *dependsGradU = false;

  return true;
}


static int ElastDirichlet(real x, real y, real z, int classA,
                          real t, real *u, int variable,
                          real *fVal)
{
  switch (variable)
    {
    case 0: fVal[0] =  0.0 - u[0];
      break;
    case 1: fVal[0] =  0.0 - u[1];
      break;
    case 2: fVal[0] =  0.0 - u[2];
      break;
    }
```

```
  return true;
}


static int ElastInitialFunc(real x, real y,  real z,
                            int classA, real *start,
                            real *dummy)
{
   start[0] = 0.0;
   start[1] = 0.0;
   start[2] = 0.0;

   return true;
}

static int ElastCauchy(real x, real y, real z, int id, real t,
                       real *u, int equation, real *fVal)
{
  switch (equation)
    {
    case 0:
      fVal[0] = force[id].fx;
      break;
    case 1:
      fVal[0] = force[id].fy;
      break;
    case 2:
      fVal[0] = force[id].fz;
      break;
    }

  return true;
}


int SetElastProblems()
{
  if (!SetTimeProblem("elast",elastVarName,
      ElastParabolic,
      ElastParabolicStruct,
```

```
ElastLaplace,
ElastLaplaceStruct,
0,
0,
ElastSource,
ElastSourceStruct,
0,
0,
ElastInitialFunc,
ElastCauchy,
ElastDirichlet,
0)) return false;
```

```
static char *brineVarName[] = {"p","w"};

static real Rho(real p, real w)
{
  real rho0 = BRINE_rho0;

  return rho0*exp(BRINE_beta*p+BRINE_gamma*w);
}

static real Mu(real w)
  {
     return BRINE_mu0*(1.0+1.85*w-4.1*w*w+44.5*w*w*w);
  }

static int BrineParabolic(real x, real y, real z, int classA,
                          real t, real *u,
                          real *ux, real *uy, real *uz,
                          real **mat)
  {
     real rho = Rho(u[0],u[1]);

     mat[0][0] = BRINE_n*rho*BRINE_beta;
     mat[0][1] = BRINE_n*rho*BRINE_gamma;

     mat[1][1] = BRINE_n*rho;

     return true;
  }

static int BrineParabolicStruct(int **structM, int **dependsS,
                                int *dependsT,
                                int *dependsU,
                                int *dependsGradU)
{
  structM[0][0] = F_FILL;     dependsS[0][0] = true;
  structM[0][1] = F_FILL;     dependsS[0][1] = true;

  structM[1][0] = F_IGNORE;   dependsS[1][0] = false;
  structM[1][1] = F_FILL;     dependsS[1][1] = true;
```

```
  *dependsT = false;
  *dependsU = true;
  *dependsGradU = false;

  return true;
}

static int BrineLaplace(real x, real y, real z, int classA,
                        real t, real *u,
real *ux, real *uy, real *uz,
real **matXX, real **matXY, real **matXZ,
real **matYX, real **matYY, real **matYZ,
real **matZX, real **matZY, real **matZZ)
{
  real rho = Rho(u[0],u[1]),
    mu = Mu(u[1]),
    q1, q2, q3, absQ, pX, pY, pZ,
    g1 = BRINE_gx,
    g2 = BRINE_gy,
    g3 = BRINE_gz,
    k = BRINE_k;

  if (classA==1) k = BRINE_k;
  else if (classA==2) k = BRINE_k2;

  GiveGradVar(x,y,z, 0,&pX,&pY,&pZ);
  q1 = -(k/mu)*(pX-rho*g1);
  q2 = -(k/mu)*(pY-rho*g2);
  q3 = -(k/mu)*(pZ-rho*g3);
  absQ = sqrt(q1*q1+q2*q2+q3*q3);
  if (absQ==0.0) { absQ=1.0;}

  matXX[0][0] = rho*k/mu;
  matXY[0][0] = 0.0;
  matXZ[0][0] = 0.0;
  matYX[0][0] = 0.0;
  matYY[0][0] = rho*k/mu;
  matYZ[0][0] = 0.0;
  matZX[0][0] = 0.0;
  matZY[0][0] = 0.0;
```

```
  matZZ[0][0] = rho*k/mu;

  matXX[1][1] = rho*(BRINE_n*BRINE_Dmol+BRINE_alphaT*absQ
     +(BRINE_alphaL-BRINE_alphaT)*q1*q1/absQ);
  matXY[1][1] = rho*(BRINE_alphaL-BRINE_alphaT)*q1*q2/absQ;
  matXZ[1][1] = rho*(BRINE_alphaL-BRINE_alphaT)*q1*q3/absQ;
  matYX[1][1] = rho*(BRINE_alphaL-BRINE_alphaT)*q2*q1/absQ;
  matYY[1][1] = rho*(BRINE_n*BRINE_Dmol+BRINE_alphaT*absQ
     +(BRINE_alphaL-BRINE_alphaT)*q2*q2/absQ);
  matYZ[1][1] = rho*(BRINE_alphaL-BRINE_alphaT)*q2*q3/absQ;
  matZX[1][1] = rho*(BRINE_alphaL-BRINE_alphaT)*q3*q1/absQ;
  matZY[1][1] = rho*(BRINE_alphaL-BRINE_alphaT)*q3*q2/absQ;
  matZZ[1][1] = rho*(BRINE_n*BRINE_Dmol+BRINE_alphaT*absQ
     +(BRINE_alphaL-BRINE_alphaT)*q3*q3/absQ);

  return true;
}

static int BrineLaplaceStruct(int **structM, int **dependsS,
                              int *dependsT, int *dependsU,
                              int *dependsGradU)
{
  structM[0][0] = F_FILL;     dependsS[0][0] = true;
  structM[0][1] = F_IGNORE;   dependsS[0][1] = false;

  structM[1][0] = F_IGNORE;   dependsS[1][0] = false;
  structM[1][1] = F_FILL;     dependsS[1][1] = true;

  *dependsT = false;
  *dependsU = true;
  *dependsGradU = false;

  return true;
}

static int BrineConvection(real x, real y, real z,
                           int classA, real t, real *u,
                           real **matX, real **matY,
                           real **matZ)
{
  real rho = Rho(u[0],u[1]),
```

```
    mu = Mu(u[1]),
    q1, q2, q3, gradX[2], gradY[2], gradZ[2],
    g1 = BRINE_gx,
    g2 = BRINE_gy,
    g3 = BRINE_gz,
    k = BRINE_k;

  if (classA==1) k = BRINE_k;
  else if (classA==2) k = BRINE_k2;

  GiveGrad(x,y,z,gradX,gradY,gradZ);

  q1 = -(k/mu)*(gradX[0]-rho*g1);
  q2 = -(k/mu)*(gradY[0]-rho*g2);
  q3 = -(k/mu)*(gradZ[0]-rho*g3);

  matX[0][0] =  2.0*(k/mu)*rho*rho*BRINE_beta*g1;
  matY[0][0] =  2.0*(k/mu)*rho*rho*BRINE_beta*g2;
  matZ[0][0] =  2.0*(k/mu)*rho*rho*BRINE_beta*g3;
  matX[0][1] =  2.0*(k/mu)*rho*rho*BRINE_gamma*g1
    -(k/mu/mu)*rho*rho*BRINE_mu0*(1.85-8.2*u[1]
    +133.5*u[1]*u[1])*g1;
  matY[0][1] =  2.0*(k/mu)*rho*rho*BRINE_gamma*g2
    -(k/mu/mu)*rho*rho*BRINE_mu0*(1.85-8.2*u[1]
    +133.5*u[1]*u[1])*g2;
  matZ[0][1] =  2.0*(k/mu)*rho*rho*BRINE_gamma*g3
    -(k/mu/mu)*rho*rho*BRINE_mu0*(1.85-8.2*u[1]
    +133.5*u[1]*u[1])*g3;

  matX[1][1] =  rho*q1;
  matY[1][1] =  rho*q2;
  matZ[1][1] =  rho*q3;

  matX[1][0] =  -rho*(k/mu)*gradX[1];
  matY[1][0] =  -rho*(k/mu)*gradY[1];
  matZ[1][0] =  -rho*(k/mu)*gradZ[1];

  return true;
}
```

```
static int BrineConvectionStruct(int **structM,
                                 int **dependsS,
                                 int *dependsT,
                                 int *dependsU)
{
  structM[0][0] = F_FILL;     dependsS[0][0] = true;
  structM[0][1] = F_FILL;     dependsS[0][1] = true;

  structM[1][0] = F_FILL;     dependsS[1][0] = true;
  structM[1][1] = F_FILL;     dependsS[1][1] = true;

  *dependsT = false;
  *dependsU = true;

  return true;
}


static int BrineSource(real x, real y, real z, int classA,
                       real t, real *u,
                       real *ux, real *uy, real *uz,
                       real *vec)
{
  real rho = Rho(u[0],u[1]),
    mu = Mu(u[1]),
    gradX[2], gradY[2], gradZ[2],
    k = BRINE_k;

  if (classA==1) k = BRINE_k;
  else if (classA==2) k = BRINE_k2;

  GiveGrad(x,y,z,gradX,gradY,gradZ);

  vec[1] = -rho*(k/mu)*
          (gradX[0]*gradX[1]+gradY[0]*gradY[1]
                            +gradZ[0]*gradZ[1]);

  return true;
}

static int BrineSourceStruct(int *structV, int *dependsS,
```

```
                              int *dependsT, int *dependsU,
                              int *dependsGradU)
{
  structV[0]    = F_IGNORE;   dependsS[0] = false;
  structV[1]    = F_FILL;     dependsS[1] = true;

  *dependsT = false;
  *dependsU = true;
  *dependsGradU = false;

  return true;
}

static int BrineInitialFunc(real x, real y, real z,
                            int classA, real *start,
                            real *startUt)
{
  start[0] = (0.03-0.012*x+1.0-z)*BRINE_rho0*BRINE_g;
  start[1] = 0.0;

  return true;
}

static int BrineDirichlet(real x, real y, real z,
                          int classA, real t,
                          real *u, int equ, real *fVal)
{
  real tRamp = 10.0;

  if (equ==0)
    {
      fVal[0] = u[0]-(0.03-0.012*x+1.0-z)*BRINE_rho0*BRINE_g;
      return true;
    }

  if (equ==1)
    {
      if ( (classA==6) || (classA==7) )
        {
          if (t<=tRamp)
             fVal[0] = u[1]-t*BRINE_w0/tRamp;
```

```
          else
             fVal[0] = u[1]-BRINE_w0;
}
      else if (classA==5)
{
  fVal[0] = u[1];
}
      else
      printf("\n weder classA==5 noch classA==6");

      return true;
    }

  return true;
}


static int BrineCauchy(real x, real y, real z, int classA,
                       real t, real *u, int equ,
                       real *fVal)
{
  real rho = Rho(u[0],u[1]),
       mu = Mu(u[1]),
       g3 = BRINE_gz, k = BRINE_k;

  if ( classA==2 )      fVal[0] = -k/mu*g3*rho*rho;
  else if ( classA==3 ) fVal[0] =  k/mu*g3*rho*rho;
  else if ( classA==6 ) fVal[0] =  k/mu*g3*rho*rho+BRINE_qc;
  else if ( classA==7 ) fVal[0] = -k/mu*g3*rho*rho+BRINE_qc;

  return true;
}


int SetBrineProblem()
{
  if (!SetTimeProblem("brine-isothermal",brineVarName,
      BrineParabolic,
      BrineParabolicStruct,
      BrineLaplace,
      BrineLaplaceStruct,
```

```
      BrineConvection,
      BrineConvectionStruct,
      BrineSource,
      BrineSourceStruct,
      (int(*)(real,real,real,int,real,real*,
                            real*,real*,real*,real**))nil,
      (int(*)(int**,int**))nil,
      BrineInitialFunc,
      BrineCauchy,
      BrineDirichlet,
      BrineSol)) return false;

  return true;
}
```