Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

RAINER ROITZSCH[1] BODO ERDMANN JENS LANG

The Benefits of Modularization: from KASKADE to KARDOS

[1]submitted to Proceedings of the 14th GAMM-Seminar Kiel on 'Concepts of Numerical Software', January 23rd to 25th, 1998

Preprint SC 98–15 (May 1998)

The Benefits of Modularization: from KASKADE to KARDOS

Rainer Roitzsch Bodo Erdmann Jens Lang

Abstract

KARDOS solves nonlinear evolution problems in 1, 2, and 3D. An adaptive multilevel finite element algorithm is used to solve the spatial problems arising from linearly implicit discretization methods in time. Local refinement and derefinement techniques are used to handle the development of the mesh over time.

The software engineering techniques used to implement the modules of the KASKADE toolbox are reviewed and their application to the extended problem class is described. A notification system and dynamic construction of records are discussed and their values for the implementation of a mesh transfer operation are shown. The need for low-level and high–level interface elements of a module is discussed for the assembling procedure of KARDOS. At the end we will summarize our experiences.

1 Introduction

Our "definition" of modularity in the programming context will remain vague and general: Breaking a program into parts. The art of programming is to do this "breaking into parts" in a way which gives you modules of moderate size, with natural functionality (reflecting the problem area), and a minimal interface. The modularization should not sacrifice efficient computation, add too much complexity, or introduce new (programming) problem areas. Hopefully, the user of a module needs only to know the interface, not the implementation. Modules should allow reuse without touching and the adaption to programming languages like Fortran, C, and C++.

It is obviously the programming environment (social and computational) and the developers personal heritage (character and taste) which guides the "modularizator" to his modularization. The selection of the implementation language (here C) gives him a set of techniques to define the interfaces between the modules. All his previous programming practice will influence his preferences. One of the authors introduces his previous LISP experience and his "living" with the Macintosh OS. The techniques to extend the description of module interfaces, which we present in this paper, have a dynamic object oriented flavor.

The KASKADE toolbox [4, 5] developed at the Konrad–Zuse–Center can be used to write algorithms to solve linear systems of elliptic partial differential equations by an adaptive refinement process. It uses the data structures of Peter Leinen to handle adaptive refinable meshes (red/green refinement, R. Bank 98). The mesh and the data arising by the discretization are stored in local data objects to get distributed arrays and sparse matrices. An overview of the modules will be given in Section 2.

The KARDOS [8] application (or toolbox extension) of KASKADE implements some new requirements on the handling of meshes and finite element vectors and matrices. KARDOS solves time–dependent, nonlinear partial differential equations

$$\begin{aligned}
&H(x,t,u)u_t - \nabla\cdot(D(x,t,u)\nabla u) = F(x,t,u,\nabla u)\\
&x \in \Omega \subset \mathbb{R}^n,\ t>0,\ n=1,2,3\\
&u = (u^1,\ldots,u^d)
\end{aligned} \tag{1}$$

with suitable boundary conditions. Some application areas are combustion problems [7, 9], pattern formation, regional hyperthermia [6], dopand diffusion in semiconductor manufacturing [10], incompressible flows, and porous media.

For the time discretization a one–step method of Rosenbrock type is used. At each new time the solution from the previous time is required to solve elliptic problems to get the intermediate stage values. The implementation of the necessary mesh transfer operation is given in Section 3.

The need to use low–level interface elements to implement efficient algorithms is shown in Section 4. The high–level interface for assembling the linear system of equations was not well suited to implement the KARDOS solver. Thus, the low–level (previously internal) interface of the KASKADE toolbox has to be made available.

2 KASKADE Modules

Figure 1 shows a simplified overview of the module structure of the KASKADE toolbox. The conventional finite element modules use the runtime interface modules and each other in a hierarchical order from left to right.

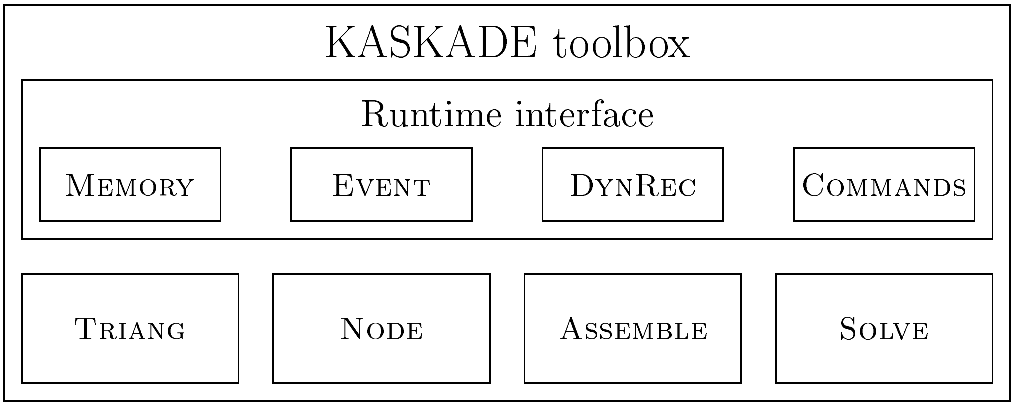

Figure 1: KASKADE modules

2.1 Runtime Interface Modules

These modules were designed to supply functionality to the C programming language and library.

The first implementation of KASKADE showed the deficiency of the (implementation dependent) C library routine `malloc`. Intensive use of this runtime routine can be very inefficient, depending on the quality of the local implementation. This introduces an unintended system dependency. The Memory module consists of routines to allocate/deallocate masses of short storage units which arise from the adaptive triangulation handling. Extra code to help debugging and catch pointer errors as early as possible are included. Data about the used memory are at any time available. This proved to be essential for the development of KARDOS because memory book–keeping errors are fatal for long time simulations.

The Event module manages events to create a notification system. Events can be defined by modules to enhance the module interface. Let us look at an example: The interface to the Solve module includes an event `NewSol` which can be used to register routines. The Graphic modules register the routine `DrawSol` to be computed when a new solution is available. The event is raised every time that a new solution is computed and the Event module will call all registered routines automatically, see Figure 2. Independently other routines can be registered for the same event.

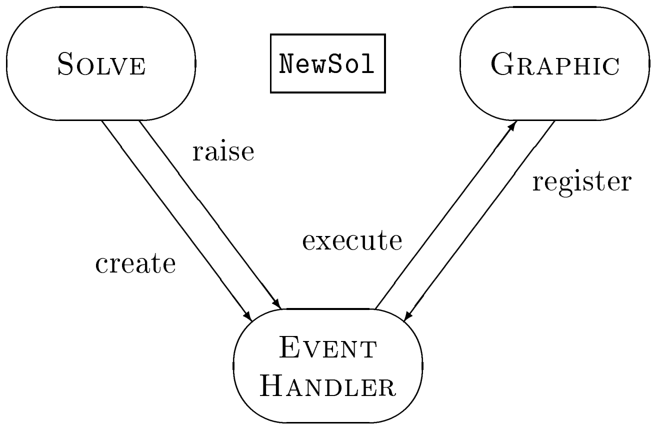

Figure 2: Automatic execution of a routine at event `NewSol`

The EVENT interface contains routines to create and delete events. These events can be used to raise events in two variants, one to process the registered routines immediately and one to put the event in a queue for processing in an event loop. At registration the user may control at which point in the sequence of registered routines the new routine is inserted. The EVENT module includes debugging code to trace event processing and inform on existing events.

The DYNREC module manages dynamic structures. The idea is a simple one: The user can request a slice of a given byte array for his use. DYNREC searches for the first free slice (maybe with some alignment condition), reserves the slice and returns the index of the first byte of the slice.

An example of utilizing this idea is the management of user storage at the triangulation. At each geometric entity (points, edges, triangles, tetrahedra) a byte array (of fixed length) is stored. A prototype for the array is handled by DYNREC and the user has access to it. For example, a user who wants to store a pointer at triangles will request 4 bytes (somehow aligned) on this prototype byte array and gets an index to a free slice. Macros are available to access this storage for all existing triangles. The DYNREC module contains debugging code like a trace and an information facility.

The EVENT and DYNREC modules support the clean separation of interfaces thus minimizing the interfaces. Different users of an event or a dynamic record need not coordinate their usage which is completely hidden from each other.

All these runtime modules include an interface to the command language module COMMANDS. The user can add commands and get some help to process parameters. Furthermore, a set of commands is available to view and change internal parameters which are registered. The complete module can be substituted by the well–known Tcl/Tk scripting language [11].

2.2 Finite Element Modules

The interface of the TRIANG module consists of a set of procedures to create, delete, or select triangulations, see Figure 3. The access to the geometric entities is handled by applying (user) functions on sets of points, edges, triangles, tetrahedra. The refinement/derefinement process is invoked by a call of open/close routines and a marking process in between.

routine	
`CrTri`	create triangulation
`SelTri`	select triangulation
`CloseTri`	delete triangulation
`ApplyP`	apply user routine to points
`ApplyE`	apply user routine to edges
`ApplyT`	apply user routine to triangles
`ApplyTD`	apply user routine to tetrahedra
`OpenRef`	prepare triangulation for refinement
`CloseRef`	refine triangulation
`RefTr, RefTd`	mark triangle (tetrahedron) for refinement
`OpenDel`	prepare triangulation for derefinement
`CloseDel`	derefining triangulation
`DelTr, DelTd`	mark triangle (tetrahedron) for deletion

Figure 3: Some routines of the interface to the TRIANG module

This functional interface is enhanced by events which allow the user to write callback procedures, see Figure 4, and by dynamic records to store data at the basic geometric objects, see Figure 5. These callback procedures are managed by the EVENT module, the dynamic records by the DYNREC module.

This interface proved to be rich enough to implement a mechanism to connect the triangles of two triangulations in a way to find for a triangle of one triangulation the corresponding triangle of the other triangulation in a very direct way, see Section MESH TRANSFER.

event	
`TriSelectOld`	before triangulation selection
`TriSelectNew`	after triangulation selection
`NewTriangle`	new triangle generated
`ReturnTriangle`	triangle to delete
`RefineTriangle`	triangle refined
`DeleteTriangle`	triangle derefined

Figure 4: Some events of the interface to the TRIANGULATION module

event	
`accPoint`	at points
`accEdge`	at edges
`accTriangle`	at triangles

Figure 5: Dynamic records managed for the TRIANGULATION module

The NODE module handles assignment of node storage (including the node number for each degree of freedom) at the triangulation. It should work for arbitrary (but fixed over the triangulation) finite elements by just using the number of nodes at points, edges, triangles, or tetrahedra. The module is responsible to identify common nodes correctly, for example to ensure the right sequence of nodes on an edge. Systems of equations are handled here too. The implementation of this module depends heavily on the runtime modules. The memory assigned to a node is distributed over the triangulation as slices of the dynamic records at points, edges, triangles, and tetrahedra.

The ASSEMBLE module contains the interface to define the coefficients of the elliptic problem or a local assembly routine, assembling the stiffness matrix and right–hand sides. Standard sets of shape functions, sets of integration points are included.

The SOLVE module defines the framework for the adaptive cycle: solving the linear system, error estimation, and refinement. Methods can be registered to the module which execute them in a well defined environment. The selection and information on these registered methods are supported in the command language.

3 Mesh Transfer

A time integration scheme for equation (1) generates a sequence of elliptic problems which can be solved by the KASKADE toolbox. In the context of this paper it is sufficient to look at the most simple equation

$$u_t - \triangle u = 0 \tag{2}$$

and the simple time integration scheme (backward Euler)

$$(I - \tau\triangle)u(t_{n+1}) = u(t_n) \tag{3}$$

to see the necessity to implement an efficient mesh transfer operator. Figure 6 shows a diagram of computational steps. The elliptic solver needs the solution computed for the mesh $\mathcal{T}_n^{m_n}$ on each of the triangulations $\mathcal{T}_{n+1}^k$

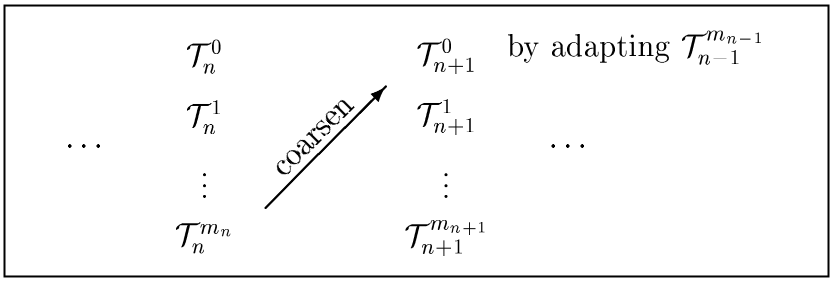

Figure 6: Mesh transfer between timesteps n and $n+1$

The following solution assumes that both triangulations are generated by refinement/derefinement algorithms from the same coarse triangulation. While assembling the local stiffness matrices and right–hand sides on triangulation $\mathcal{T}_{n+1}^k$ we need to find the values of the solution on $\mathcal{T}_n^{m_n}$ at the integration points. Our solution is to connect both triangulations by linking each triangle with a pointer to the finest triangle (of the other triangulation) which includes or equals the current triangle, see Figure 7.

Both (coarse) triangulations are connected at the start of the computations. Each time a triangle is refined or derefined this information is updated in a straightforward manner. The routines to do this are registered at the events `RefineTriangle` and `DeleteTriangle`.

Now let us discuss the actual implementation. The first part is the initialization, here a simplified pseudocode:

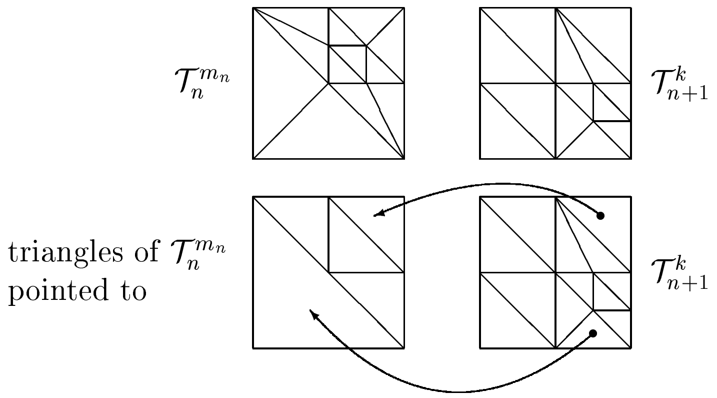

Figure 7: Connecting two triangulations

```
#define ALIGN 4
int partnerTriangle;
void InitMeshTransfer(TRIANGUALATION *triang1,
                      TRIANGUALATION *triang2)
  {
    Triangle *t, *tPartner

    partnerTriangle = GetAccess(accTriangle, sizeof(*Triangle),
                                ALIGN, "partner");
    Register(SetPartner, RefineTriangle,
             "set partner at refinement");
    Register(UnsetPartner, DeleteTriangle,
             "unset partner at derefinement");

    forall (Triangle* t∈triang1)
      {
        tPartner = FindPartner(t, triang2);
        SetPointer(t, partnerTriangle) = tPartner;
        SetPointer(tPartner, partnerTriangle) = t;
      }
    return;
  }
```

Note that the strings used in the calls to `GetAccess` and `Register` are only needed for documentation and debugging, they need not to be unique. The routine `SetPartner` is always called when a triangle `t` is refined.

```
void SetPartner(Triangle *t)
  {
    Triangle *tPartner = GetPointer(t, partnerTriangle);

    if ((t->refType)==(tPartner->refType))
      {
        SetPartersOnSons(t, tPartner);
        SetPartersOnSons(tPartner, t);
        return;
      }
    SetSameParterOnSons(t, tPartner);
    return;
  }
```

The field `refType` of the `Triangle` data structure contains the refinement type (green, red or not refined) of a triangle. The routines `SetPartersOnSons` and `SetSameParterOnSons` will do what their names imply. The routine `UnsetPartner` which is called just before a triangle is derefined looks even more simple:

```
void UnsetPartner(Triangle *t)
  {
    Triangle *tPartner = GetPointer(t, partnerTriangle);

    if ((t->refType)==(tPartner->refType))
      SetSameParterOnSons(tPartner, t);
    return;
  }
```

Now it is relatively easy to find the values of the solution on the previous triangulation. When assembling the local stiffness for a triangle the best triangles to start a search is directly available.

Let us finish this section with a remark on the coarsening of the grid. The implemented strategy must guaranty the return to the coarsest triangles in regions where the solution allows this. The simplest method would be to start with the coarse triangulation at each new time. This technique is used in the older code KASTIO [3]. A more sophisticated strategy is the trimming tree technique [8] which removes just the (local) finest level on the complete domain. The adaptive coarsening strategy implemented in the current version of KARDOS uses the result of the last error estimation, thus loosing a minimum of information.

4 Efficiency of Assembling

KARDOS implements a much more complicated scheme than the backward Euler schema (3). A step in this direction is the reformulation of the equation which has the difference $u(t_{n+1}) - u(t_n)$ as solution of

$$(I - \tau\triangle)(u(t_{n+1}) - u(t_n)) = \tau\triangle u(t_n) \quad . \tag{4}$$

Even in this simple setting we need two matrices, the mass matrix for I and the stiffness matrix for $\triangle$. In the more complex environment of the Rosenbrock scheme four matrices are needed. The ASSEMBLE module of KASKADE includes a data type to define a user problem by user functions for the coefficients of the partial differential equation. This environment is used to assemble the linear system of equations. In the KARDOS environment we would use the interface as follows:

```
massMatrix = Assemble(MassProblem);
stiffnessMatrix = Assemble(LaplaceProblem);
```

but this would result in a very inefficient code. For each call of `Assemble` each triangle is touched once and some calculations (like the transformation of the integration points, the transfer of the previous solution at these points) are done twice. Additionally, the information to store structure of the sparse matrices is doubled. What we need is a much finer interface of the ASSEMBLE module which allows to write the assembling loop directly:

```
sparseStructure = MakeSparseStructure();
massMat = MakeSparseMatrix(sparseStructure);
stiffnessMat = MakeSparseMatrix(sparseStructure);
forall (Triangle* t∈triang)
  {
    localData = ComputeLocalData(t);
    localMassMat = AssembleLocal(MassProblem, localData);
    localStiffnessMat = AssembleLocal(LaplaceProblem,
                                      localData);
    globalIndices = GetNodeIndices(t);
```

```
    AddLocalMatrix(massMat, localMassMat, globalIndices);
    AddLocalMatrix(stiffnessMat, localStiffnessMat,
                   globalIndices);
  }
```

This version is much more efficient, but further optimization is still feasible.

5 Conclusion

The modularization concept described in this paper has proved to be flexible enough to implement complicated algorithms such as needed for KARDOS. Newly emerging software techniques (like C++, Java, or an object oriented design methodology) were used to implement KASKADE 3.0 in C++ [2]. Both versions are freely distributed at `ftp://ftp.zib.de/pub/kaskade`. The changes necessary to adapt KARDOS to the new version request a substantial effort.

References

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

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

[3] Bornemann, F.A.: "An Adaptive Multilevel Approach to Parabolic Equations. III: 2D Error Estimation and Multilevel Preconditioning", IMPACT **2**, p. 1–45, 1992.

[4] Deuflhard, P., Leinen, P., Yserentant, H.: "Concept of an Adaptive Hierarchical Finite Element Code", IMPACT **1**, p. 3–35, 1989.

[5] Erdmann, B., Lang, J., Roitzsch, R.: "KASKADE Manual, Version 2.0", ZIB TR93–5, Berlin, 1993.

[6] Erdmann, B., Lang, J., Seebaß, M.: "Adaptive Solutions of Nonlinear Parabolic Equations with Application to Hyperthermia Treatments",

Proc. Int. Symp. on Advances in Comp. Heat Transfer, Çeşme, Turkey, 1997.

[7] Fröhlich, J., Lang, J.: "Twodimensional Cascadic Finite Element Computations of Combustion Problems", to appear in Comp. Meth. Appl. Mech. Engrg., 1998.

[8] Lang, J.: "Adaptive FEM for Reaction–Diffusion Equations", Appl. Numer. Math. **26**, p. 105–116, 1998.

[9] Lang, J., Erdmann, B., Roitzsch, R.: "Adaptive Time-Space Discretization for Combustion Problems", Proc. 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics (Ed. A. Sydow), Vol. **2** (Numerical Mathematics), p. 149–155, Berlin, 1997.

[10] Lang, J., Merz, W.: "Numerical Simulation of Single Species Dopant Diffusion in Silicon under Extrinsic Conditions", ZIB SC 97–47, Berlin, 1997.

[11] Ousterhout, J.K.: "Tcl and the Tk Toolkit", Addison–Wesley, Reading, 1994.