ELSEVIER

Applied Numerical Mathematics 21 (1996) 439–467

APPLIED NUMERICAL MATHEMATICS

Object-oriented programming of adaptive finite element and finite volume methods ☆

Jinn-Liang Liu [a,*], Ing-Jer Lin [b], Miin-Zhih Shih [a], Ren-Chuen Chen [a], Mao-Chung Hsieh [c]

[a] *Department of Applied Mathematics, National Chiao Tung University, Hsinchu, Taiwan*
[b] *Department of Mechanical Engineering, National Chiao Tung University, Hsinchu, Taiwan*
[c] *Department of Power Mechanical Engineering, National Yunlin Polytechnic Institute, Yunlin, Taiwan*

Abstract

This article describes an object-oriented implementation of the finite element method and the finite volume method in a unified adaptive system using the programming language C++. The system applies to various types of mathematical model problems. Traditionally, different numerical methods for different types of problems are implemented independently by procedural languages such as C and Fortran. Moreover, adaptive analysis programs are more complicated than nonadaptive programs. Nevertheless, these methods share many common properties such as linear system solvers, data structures, a posteriori error analyses, and refinement processes. Some advantageous features of object-oriented programming are demonstrated through the integration of these properties in the adaptive system. New data types of objects specific to adaptive methods are also introduced. The system is well-structured, extendable, and maintainable due mainly to the nature of encapsulation and inheritance of object-oriented programming.

Keywords: Object-oriented programming; C++; Adaptive methods; Finite elements; Finite volumes

1. Introduction

Adaptivity is currently one of the major concepts in practical and large-scale computations [8,35]. Despite many object-oriented programming (OOP) examples of finite element programs which have been developed in recent years to demonstrate the benefits of object-oriented languages in implementation [12–14,22,30,36] some important issues, such as unstructured meshes and their associated data structures, dynamic refinement strategies, and a posteriori error analyses, specific to adaptive implementations remain to be addressed. The programming effort required to create adaptive finite

☆ This work was supported by NSC-grants 82-0208-M-009-060 and 83-0208-M-009-057, Taiwan.
* Corresponding author. E-mail: jinnliu@math.nctu.edu.tw.

PII S0168-9274(96)00044-X

element programs is obviously more involved than that of nonadaptive programs. OOP principles can be powerful tools to develop adaptive programs.

The adaptive implementation presented here is based on three widely used numerical methods, namely, the finite element method (FEM), the least-squares finite element method (LSFEM), and the finite volume method (FVM). The literature on these methods can be found in many mathematical and engineering journals. We do not attempt to elaborate the theoretical aspects of these methods within adaptivity theory. We shall instead illustrate some fundamental features of adaptive methods in connection with the programming implementation in an OOP language. The prototyping language C++ is used. Based on the concepts of adaptivity and OOP, we have developed a research code called AdaptC++.

The objectives of the present article are threefold. First, based on commonly shared procedures such as linear system solvers, data structures, a posteriori error analyses, and refinement processes, we describe how to integrate the adaptive implementation of different numerical methods in a single package without repeating common tasks and common data management. Second, we describe a unified, adaptive, computing environment that can be used for various types of mathematical model problems which are usually treated by different methods. More specifically, a generic formula is presented that can be reformulated into the respective formulas suitable for FEM, LSFEM, and FVM. This generic formula is important in the sense that it serves as a "base" formula which will be "inherited" by those methods in their own reformulation. The concept of this hierarchical formulation fits naturally into the OOP language itself. The features of OOP, particularly those of encapsulation and inheritance, provide a very modular and hierarchical coding structure. The structure is similar to a tree structure and is constructed by various characteristic classes. Our final objective is to illustrate the design of the classes in AdaptC++.

The remainder of the paper is organized as follows. The guiding principles for the design of AdaptC++ will be given in Section 2. These principles are characterized by mathematical models which determine the scope of applications of AdaptC++, by adaptivity which is the primal interest in developing the code, and by OOP for which some standard terminology will be emphasized. Section 3 includes all classes designed for the code. These classes are presented in such a way that their distinctive qualities are described as briefly as possible and yet sufficiently to show the connection of adaptivity and OOP. The standard 1-irregular refinement scheme [4,10,29] is usually implemented in Fortran, a simplified version in C++ is given in Section 4. In Section 5, numerical experiments on three model problems are given to demonstrate the use of abstract classes defined in AdaptC++. Finally, some concluding remarks will be made in Section 6.

2. The governing principles

We begin with an illustration of a typical procedure of adaptive solution analysis, see also Fig. 1. Given a mathematical model problem under study, we first partition the solution domain into a set of finite elements or finite volumes. The model problem is then approximated by either FEM or FVM. After the assembly of global stiffness and mass matrices and a load vector, a system of linear equations can be solved by either a direct or an iterative solver. Once an approximate solution is computed, an a posteriori error analysis is performed to assess the quality of the approximate solution. The error analysis will produce an error estimator and a set of error indicators. The error estimator is

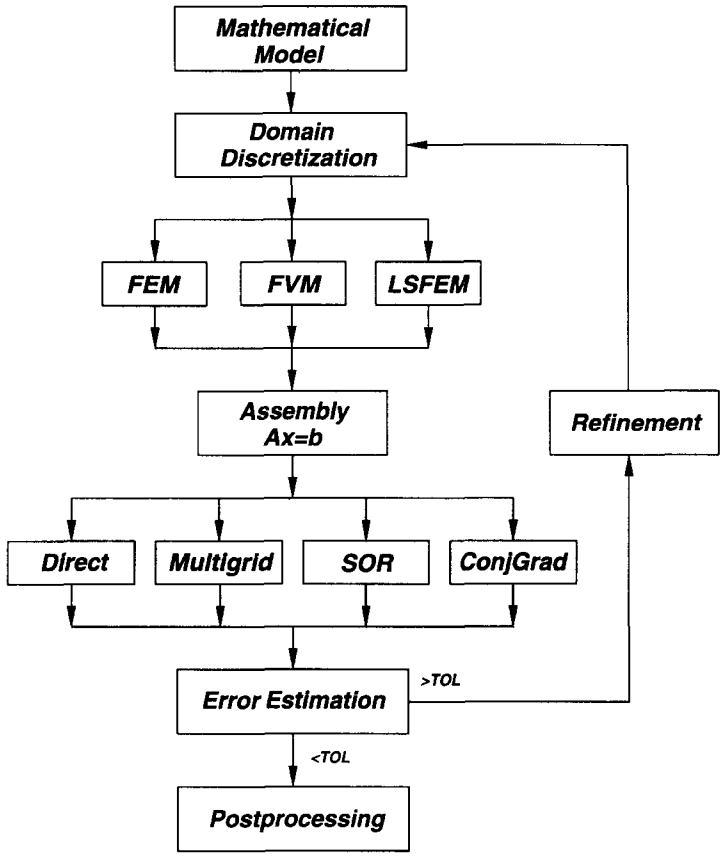

Fig. 1. AdaptC++.

a global assessment of the approximate solution. If the estimator is less than a preset error tolerance, the adaptive process will be terminated and the approximate solution can be postprocessed for further analysis. Otherwise, a refinement scheme is employed to refine or coarsen each of the current elements depending on the magnitude of the error indicator for that element. For FVM, the error indicator is calculated on an element-by-element basis with the finite volumes being the dual of finite elements. A finer partition of the domain is thus created and a new solution procedure is repeated.

The fundamental principles of the design of AdaptC++ can be classified as follows.

2.1. Mathematical models

AdaptC++ requires mathematical model problems to be cast into the following generic form

$$\begin{cases} -\boldsymbol{p}_x(\boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_y) - \boldsymbol{q}_y(\boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_y) + \boldsymbol{r}(\boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_y) = \boldsymbol{f} \quad \text{in } \Omega, \\ B_D(\boldsymbol{u}) = \boldsymbol{g}_D \quad \text{on } \partial\Omega_D, \qquad \partial\Omega = \partial\Omega_D \cup \partial\Omega_N, \\ B_N(\boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_y) = n_1\boldsymbol{p} + n_2\boldsymbol{q} = \boldsymbol{g}_N \quad \text{on } \partial\Omega_N, \end{cases} \tag{2.1}$$

where Ω is a given domain in $\mathbb{R}^2$, $\boldsymbol{u}$, $\boldsymbol{p}$, $\boldsymbol{q}$, $\boldsymbol{r}$, $\boldsymbol{f}$, $\boldsymbol{g}_D$ and $\boldsymbol{g}_N$ are m-dimensional vector-valued functions, and n_1 and n_2 are components of the outward unit normal vector $\boldsymbol{n}$ on the Neumann boundary $\partial\Omega_N$.

Here, $\boldsymbol{p}_x$, $\boldsymbol{u}_y$, etc. denote partial derivatives and $\boldsymbol{u} = (u(1), \ldots, u(m))$, etc. The functions $\boldsymbol{p}$, $\boldsymbol{q}$ and $\boldsymbol{r}$ can be expressed by

$$\begin{cases} p(i) := \sum_{k=1}^{m} \{CP(i,k)u(k) + CPX(i,k)u_x(k) + CPY(i,k)u_y(k)\}, \\ q(i) := \sum_{k=1}^{m} \{CQ(i,k)u(k) + CQX(i,k)u_x(k) + CQY(i,k)u_y(k)\}, \\ r(i) := \sum_{k=1}^{m} \{CR(i,k)u(k) + CRX(i,k)u_x(k) + CRY(i,k)u_y(k)\}, \end{cases} \tag{2.2}$$

for $i = 1, \ldots, m$, where the coefficient functions $CP(i,k)$, $CPX(i,k)$, etc. are specified by the user. This generic formulation allows the user to choose FEM and/or FVM for numerical experiments with AdaptC++. Of course, model problems suitable for FE approximation may not be suitable for FV approximation and vice versa.

For FE approximation, the generic problem (2.1) is formulated in the variational form

$$B(\boldsymbol{u}, \boldsymbol{v}) = F(\boldsymbol{v}), \tag{2.3}$$

where $B(\cdot,\cdot)$ and $F(\cdot)$ are bilinear and linear forms defined respectively by

$$\begin{cases} B(\boldsymbol{u}, \boldsymbol{v}) := \int_{\Omega} (\boldsymbol{p} \cdot \boldsymbol{v}_x + \boldsymbol{q} \cdot \boldsymbol{v}_y + \boldsymbol{r} \cdot \boldsymbol{v}) \, \mathrm{d}x \, \mathrm{d}y, \\ F(\boldsymbol{v}) := \int_{\Omega} \boldsymbol{f} \cdot \boldsymbol{v} \, \mathrm{d}x \, \mathrm{d}y + \int_{\partial\Omega_N} \mathrm{g}_N \cdot \boldsymbol{v} \, \mathrm{d}s. \end{cases} \tag{2.4}$$

We also consider a class of first-order systems of partial differential equations that can be approximated by using LSFEM [3,6,7,17]. In this case the bilinear and linear forms are defined by

$$\begin{cases} B(\boldsymbol{u}, \boldsymbol{v}) := \int_{\Omega} \big(\boldsymbol{r}(\boldsymbol{u}, \boldsymbol{u}_x, \boldsymbol{u}_y) \cdot \boldsymbol{r}(\boldsymbol{v}, \boldsymbol{v}_x, \boldsymbol{v}_y)\big) \, \mathrm{d}x \, \mathrm{d}y, \\ F(\boldsymbol{v}) := \int_{\Omega} \big(\mathrm{f} \cdot \boldsymbol{r}(\boldsymbol{v}, \boldsymbol{v}_x, \boldsymbol{v}_y)\big) \, \mathrm{d}x \, \mathrm{d}y. \end{cases} \tag{2.5}$$

The FEM, furthermore, applies to a class of variational inequalities that can be used to describe, for example, the flow of an incompressible inviscid fluid through an unsaturated porous medium [2,18,27], contact and obstacle problems [16,28], and semiconductor device simulation [19], etc. This class of problems are cast into the following formula [15]

$$B(\boldsymbol{u}, \boldsymbol{v}) - F(\boldsymbol{v}) \geqslant B(\boldsymbol{u}, \boldsymbol{u}) - F(\boldsymbol{u}), \tag{2.6}$$

where the bilinear and linear forms are given in (2.4).

There are several variants of FV schemes [9]. The vertex based FVM is implemented in our code. This FV scheme has the feature that the unknowns are held at the primary cell vertices but conservation is applied over a secondary system of cells centered on the unknowns. In AdaptC++, the primary cells are exactly the finite elements whereas the secondary cells are called the finite volumes, see Section 4. One of the important common properties of FEM and FVM in AdaptC++ is that all unknowns are held at finite element nodal points.

The FV approximation is based on the following formula

$$\int_{\partial b_i \setminus \partial\Omega} (\boldsymbol{q}\,\mathrm{d}x - \boldsymbol{p}\,\mathrm{d}y) + \int_{b_i} \boldsymbol{r}\,\mathrm{d}x\,\mathrm{d}y = \int_{b_i} \mathrm{f}\,\mathrm{d}x\,\mathrm{d}y + \int_{\partial b_i \cap \partial\Omega_N} \mathrm{g}_N\,\mathrm{d}s, \tag{2.7}$$

for all $b_i \in \mathcal{B}$ where $\mathcal{B}$ denotes a FV partition of Ω.

2.2. Adaptivity

Adaptive methodology involves two basic processes: a posteriori error estimation and refinement. The refinement scheme implemented in AdaptC++ will be discussed in Section 4. The a posteriori error estimation can be regarded as the "heart" of the adaptive mechanism. The weak-residual type of error estimation proposed in [21] is presented in a general framework in [20]. It is this unified approach for error estimation that motivates us to combine FEM, FVM, LSFEM, and variational inequalities in one package with the aid of OOP.

Let $\boldsymbol{u}_h$ be a computed solution by FEM or FVM. Let $\widetilde{\boldsymbol{e}}$ denote an error estimate of the exact error $\boldsymbol{u} - \boldsymbol{u}_h$. The error estimate is computed elementwise and is based on the formula

$$B(\widetilde{\boldsymbol{e}}, \boldsymbol{w}) = F(\boldsymbol{w}) - B(\boldsymbol{u}_h, \boldsymbol{w}) \tag{2.8}$$

for FEM, FVM, and LSFEM, and based on

$$B(\widetilde{\boldsymbol{e}}, \boldsymbol{w}) - \left[F(\boldsymbol{w}) - B(\boldsymbol{u}_h, \boldsymbol{w})\right] \geqslant B(\widetilde{\boldsymbol{e}}, \widetilde{\boldsymbol{e}}) - \left[F(\widetilde{\boldsymbol{e}}) - B(\boldsymbol{u}_h, \widetilde{\boldsymbol{e}})\right] \tag{2.9}$$

for variational inequalities. We refer to [20,21] for more details of the implementation of weak-residual error estimators. Note particularly that the errors of the FE and FV solutions are estimated using the same formula and that the solution processes of approximation and estimation are exactly the same for FEM, LSFEM, and variational inequalities.

2.3. OOP

The standard terminology in OOP such as abstract data type, object, class, encapsulation, message, function, inheritance, polymorphism, dynamic binding, etc. can be found in many programming books, see, e.g., [32]. We also refer to [36] for their use in finite element programming. However, the following terms are particularly emphasized owing to their importance in our presentation that follows.

(i) *Classes*: A class specification has two parts: (1) a class declaration which describes its component members, i.e., data members and function members, and (2) method definitions which describe how certain class member functions are implemented. When a class inherits from another class, the original class is called a base class and the inheriting class is called a derived class. A derived class includes all nonprivate features of its ancestors and then adds its own characteristics.

(ii) *Pure virtual functions*: A pure virtual function is a function that has no definition within the base class. Consequently, the function must be defined in one of its derived classes.

(iii) *Abstract classes*: A class that contains at least one pure virtual function is said to be abstract. No objects may be generated by using an abstract class, since it contains one or more undefined functions. It nevertheless creates pointers. This allows a support of dynamic linking (run-time polymorphism), which relies upon the pointer to select a proper function from derived classes.

3. Classes specific to AdaptC++

The classes designed in AdaptC++ are divided into two groups. The first group, shown in Fig. 2, consists of six auxiliary classes, *Node*, *Element*, *Matrix*, *Vector*, *BdryData*, and *GaussQuad* which are associated with the base class *Adaptor*. They are neither derived from nor derive any class. The second group, see Fig. 3, consists of *Adaptor* and its descendant classes where the classes shown in ellipses are abstract classes and those in rectangles are user-defined classes. The directed acyclic graph represents the relation of inheritance within classes. It clearly indicates how the user can choose a path (a methodology) to perform a numerical experiment with AdaptC++.

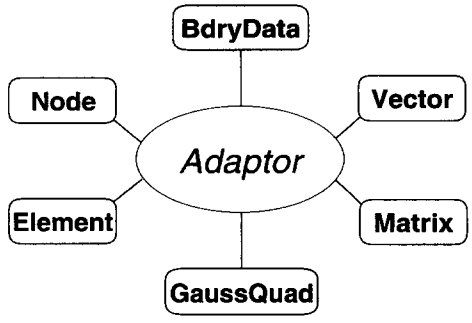

Fig. 2. Auxiliary classes.

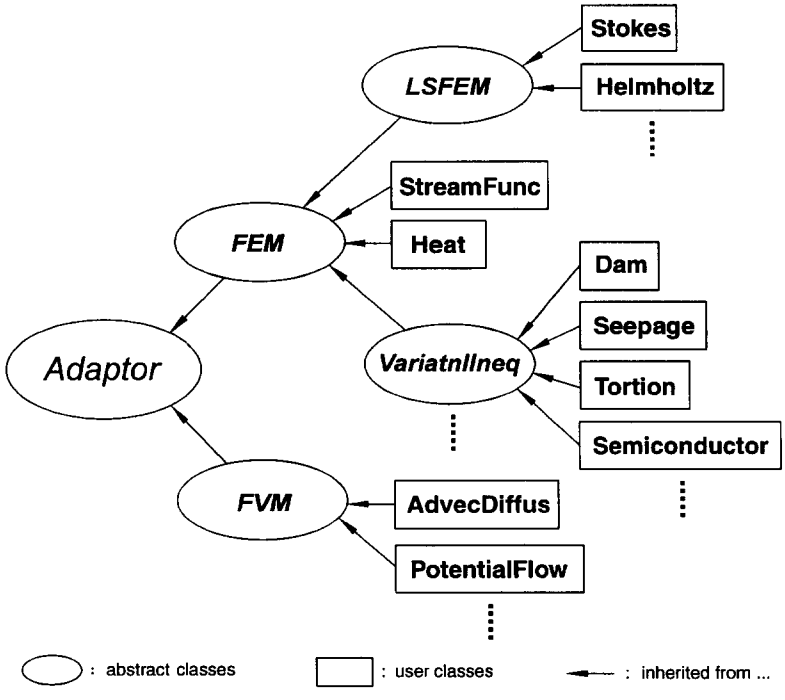

Fig. 3. Abstract classes and user classes.

Each individual class is characterized by its data members and member functions which are designed in order to perform certain tasks distinguishable from other classes. Specification of classes by tasks, data members, and member functions provides a very manageable coding layout. We describe the classes of AdaptC++ using this specification.

3.1. Auxiliary classes

All degrees of freedom associated with FEM or FVM are held at regular nodes of some 1-irregular mesh. Data for each regular node includes node_index, node_cood, and node_value corresponding respectively to the sequence number and the coordinates of that node, and the computed m-dimensional function value at that node; see Table 1. Once a refinement or an approximation is completed, these data will be stored by functions SetNodeIndex, SetNodeCood, or SetNodeValue. Similarly, these data can be retrieved for estimation, refinement, or postprocessing via GetNodeIndex, etc.

The refinement process dynamically generates a family of hierarchical elements. The data member elem_level in Table 2 indicates the rank of an element in the hierarchy. The pointers *father, **son, **neighbor, and **node determine the unique location of the element in the tree structure of the hierarchy and hence the exact location in a specific mesh. Using pointers to define these data members enables the advantageous feature of dynamic memory allocation in OOP to be fully taken for the dynamic data structure associated with the adaptively unstructured mesh. Quadrilateral elements of NFEARS [24] are implemented in our code. The use of pointers also provides users with a flexible choice of FE spectral order. For instance, the double pointer **node allows users to use four-noded (linear) or eight-noded (quadratic) quadrilaterals with which the associated data will be allocated at run time. Similarly, using double pointers, the object values of the variables son and neighbor are determined at run time, i.e., an element can be devided into two or four subelements (the latter is implemented here) and it may have two to four neighboring elements depending whether or not it is next to the boundary. Again, these data members are operated by the Set and Get functions analogous to those in *Node*. The Print function will be called by the member function PostProcess of *Adaptor* when the current mesh geometry is requested for visualization.

The classes *Matrix* and *Vector* of Tables 3 and 4 are designed for data management on sparse matrices which are naturally generated by FEM and FVM. Objects of *Matrix* and *Vector* are link-list objects in which only nonzero entries are stored with the begin pointers *begin. Manipulation on these nonzero entries is performed via NonZeroSearch.

Table 1
Class *Node*

Tasks	Member functions	Data members
Set data	SetNodeIndex	node_index
	SetNodeCood	node_cood
	SetNodeValue	node_value
Get data	GetNodeIndex	
	GetNodeCood	
	GetNodeValue	

Table 2
Class *Element*

Tasks	Member functions	Data members
Set data	SetElemIndex	elem_index
	SetElemLevel	elem_level
	SetFather	*father
	SetSon	**son
	SetNeighbor	**neighbor
	SetNode	**node
	SetEdge	*edge
Get data	GetElemIndex	
	GetElemLevel	
	GetFather	
	GetSon	
	GetNeighbor	
	GetNode	
	GetEdge	
Print data	Print	

Table 3
Class *Matrix*

Tasks	Member functions	Data members
Set dimension	SetRowSize	rows
	SetColSize	cols
		*begin
Get dimension	GetRowSize	
	GetColSize	
Get data	NonZeroSearch	

Table 4
Class *Vector*

Tasks	Member functions	Data members
Set dimension	SetVectorSize	size
		*begin
Get dimension	GetVectorSize	
Get data	NonZeroSearch	

Table 5
Class *BryData*

Tasks	Member functions	Data members
Set data	SetBCData	*dof_index
		*dof_value

Table 6
Class *GaussQuad*

Tasks	Member functions	Data members
Get data	GetXCoord	*x_coord
	GetYCoord	*y_coord
	GetWeights	*weights
		dim
		pts

Essential boundary conditions on the Dirichlet part of boundary $\partial\Omega_D$ are specified in the class *BdryData* (Table 5). At each regular node, the number of degrees of freedom is equal to m the dimension of unknown function $\boldsymbol{u}$. The data member dof_index indicates which component of the vector $\boldsymbol{u}$ should be imposed with its specified boundary value (dof_value) per SetBCData. The boundary data will then be used in the assembling procedure in *Adaptor* per ImpBdryCond.

The class *GaussQuad* (Table 6) provides the coordinates and weighting coefficients for Gaussian integration. For double integrals, the number of integrating points (pts) in each direction is assumed to be the same. For example, in a square region, i.e., dim = 2, the functions GetXCoord and GetYCoord will return the coordinates (x_coord(i), y_coord(j)) of pts^{dim} Gaussian nodes, $i, j = 1, \ldots, \text{pts}$, with the corresponding weights weights(k), $k = i + j \times \text{pts}$.

3.2. Abstract classes

Most data members and member functions required for a complete process shown in Fig. 1 are declared and partially defined in *Adaptor* (Table 7). Many important features of OOP can be seen in this class. We describe the class by means of its tasks in the order shown in the flow chart of Fig. 1.

(1) *Input*: The functions GenericP,Q,R,F,GN,GD declared as pure virtual in *Adaptor* create pointers which should ultimately point to some user class in which these functions are explicitly defined according to the generic form (2.1). The data member sol_dim denotes the dimension of the vector-valued unknown function. The number of initial elements (init_elem_no) and the number of initial regular nodes (init_node_no) are given in a user file called InitMesh. This file should also include the initialization data: the coordinates of the nodes, the relation between neighboring elements and nodes, and the boundary type (interior, Dirichlet, Neumann) of each one of four edges of the elements. The member function ReadInitMesh reads these data and then passes it to the classes *Node* and *Element* for data allocation.

(2) *Approximation*: Approximation using FEM is based on the formula (2.3) or (2.6) which requires the specification of the bilinear and linear forms. These two forms are coded in the

Table 7
Class *Adaptor*

Tasks	Member functions	Data members
Input	GenericP,Q,R,F,GN,GD†	sol_dim
	ReadInitMesh	init_elem_no
		init_node_no
Approximation	GenericP,Q,R,F,GN,GD†	elem_order
	BilinearForm†	
	LinearForm†	
Assembly	GetStiffMatrix†	stiff
	GetLoadVector†	load
	ImpBdryCond	
Solution	Solve	solver_type
	Direct	solution
	Multigrid	
	SOR	
	ConjGrad	
Estimation	BilinearForm†	norm_type
	LinearForm†	error_indicator
	GetApproErrNorm	appro_err_norm
	GetApproSolNorm	appro_sol_norm
	RelativeError	relative_error
	Tolerance	tolerance
		errest_type
Refinement	UniformRefine	init_elem_loca
	AdaptiveRefine	total_levels
	ElementDivide	total_elems
	Level	total_nodes
		total_dofs
		refine_criterion
Data search	StartElemSearch	*init_elem_address
	SearchElemTree	*now_loca_address
	SearchUndividedElem	*elem_pointer
	EndElemSearch	
Output	PostProcess	post_process

†: pure virtual functions

functions BilinearForm and LinearForm in the class *FEM*. For FVM, the functions GenericP,Q,R,F,GN,GD will be used since the approximation is based on the formula (2.7). The spectral order (elem_order) of the finite element approximation can be chosen as either linear or quadratic, whereas for FVM only linear approximation is implemented in the code.

(3) *Assembly*: The computation of local stiffness matrices and load vectors and their assembling procedure are solely determined by the approximation method FEM or FVM. Hence, the pure virtual functions GetStiffMatrix and GetLoadVector should be defined in *FEM* or *FVM*. However, the way that the essential boundary conditions are imposed is the same for both FEM and FVM since all prescribed values are stored with the regular nodes of the boundary. Therefore, the imposition function ImpBdryCond for boundary data is defined in this class. These functions generate a matrix stiff and a vector load which are objects of classes *Matrix* and *Vector*, respectively.

(4) *Solution*: The system of linear equations produced in the previous stage is solved by either a direct method (Direct) such as Gaussian elimination or by an iterative method. The choice is made via the declaration of the data member solver_type. The iterative methods that are implemented in our code are multigrid, SOR, and conjugate gradient methods. Because of encapsulation, any solver written in C or C++ can be easily incorporated in our code. These functions are accessed indirectly by the interface function Solve which then produces a solution (solution) on the current mesh.

(5) *Estimation*: The functions BilinearForm and LinearForm will be used in both approximation, i.e., (2.3) and (2.6), and estimation, i.e., (2.8) and (2.9), with different construction of shape functions for trial and test functions. There are several ways of constructing shape functions for the a posteriori error estimation [21]. They can be specified by the data member errest_type. Once the approximate solution u_h is available, a local problem based on (2.8) or (2.9) is solved in each element to obtain a local solution $\tilde{e}$. The solution is then computed by GetApproErrNorm to get an error indicator in some suitable norm (the default is the energy norm, i.e., norm_type = ENERGY) for that element. All error indicators are stored in the vector error_indicator. GetApproErrNorm also computes the error estimator (appro_err_norm) of the current solution. The function GetApproSolNorm is called for measuring the approximate solution in the energy norm (appro_sol_norm). The relative error (relative_error = appro_err_norm/[appro_sol_norm + appro_err_norm]) provides one of the stopping criteria (e.g., relative_error $\leqslant$ tolerance = 0.01, see Fig. 1). This is implemented in RelativeError.

(6) *Refinement*: Both uniform and adaptive refinement schemes (UniformRefine and AdaptiveRefine) are provided by the code. The refinement process will generate trees of elements and nodes. The total number of trees equals the number of initial elements (init_elem_no) which are also the roots of those trees. The structure of the tree consists of the location of the root element (init_elem_loca), the hierarchy of the tree (total_levels), the total number of elements and nodes (total_elems, total_nodes), and the total number of degrees of freedom (total_dofs). More algorithmic details for the member functions AdaptiveRefine and ElementDivide will be given in Section 4.

(7) *Data search*: To search data in a particular element, the function StartElemSearch resets the current element pointer (*elem_pointer) to an address of some root element (*init_elem_address). The function SearchElemTree is implemented by a depth first search algorithm that requires

an additional look-ahead pointer *now_loca_address. SearchElemTree visits all elements of the current tree. The function SearchUndividedElem returns the search of leaves of the tree which are undivided elements where the data is stored. The search is completed with the function EndElemSearch.

(8) *Output*: The standard output data provided by the code are appro_sol_norm, appro_err_norm, relative_error, etc. The function PostProcess generates a formatted data file which contains coordinates of nodes in the current mesh and can be postprocessed by, e.g., the software Mathematica™ to draw a diagram of the mesh.

In *FEM* (Table 8), the generic problem (2.1) (GenericP,Q,R,F,GN,GD) is reformulated into the variational problem (2.3). The reformulation is coded in the member functions BilinearForm and LinearForm which are used for the calculation of local stiffness matrices (GetStiffMatrix) and load vectors (GetLoadVector).

The functions BilinearForm and LinearForm defined in *LSFEM* (Table 9) are based on the formula (2.5). The assembly functions inherit from those of *FEM*.

All member functions of approximation and assembly in *FEM* are inherited by the class *VariatnlIneq* (Table 10). However, the solution function SOR has to be modified since the approximate solution of (2.6) should satisfy a discrete constraint condition which is determined by the constraint functions UpConstraint and LowConstraint, see, e.g., [15].

Table 8
Class *FEM*: public *Adaptor*

Tasks	Member functions
Input	GenericP,Q,R,F,GN,GD†
Approximation	BilinearForm
	LinearForm
Assembly	GetStiffMatrix
	GetLoadVector
Estimation	BilinearForm
	LinearForm

Table 9
Class *LSFEM*: public *FEM*

Tasks	Member functions
Input	GenericP,Q,R,F,GN,GD†
Approximation	BilinearForm
	LinearForm
Estimation	BilinearForm
	LinearForm

Table 10
Class *VariatnlIneq*: public *FEM*

Tasks	Member functions
Input	GenericP,Q,R,F,GN,GD†
	UpConstraint†
	LowConstraint†
Solution	SOR
Estimation	BilinearForm
	LinearForm

Table 11
Class *FVM*: public *Adaptor*

Tasks	Member functions
Input	GenericP,Q,R,F,GN,GD†
Approximation	GenericP,Q,R,F,GN,GD†
Assembly	GetStiffMatrix
	GetLoadVector
Estimation	BilinearForm
	LinearForm

For FV approximation (Table 11), the functions GetStiffMatrix and GetLoadVector use directly the input functions GenericP,Q,R,F,GN,GD according to the formula (2.7). For error estimation, the functions BilinearForm and LinearForm are defined as those in *FEM* because they use the same formula (2.8).

3.3. User classes

We only describe the user classes *Stokes* and *StreamFunc* in Fig. 3. Other classes are defined in the same fashion. The model problems tested by AdaptC++ are from references [17] for *Stokes*, [7] for *Helmholtz*, [3] for *Heat*, [34] for *Dam*, [2] for *Seepage*, [16] for *Torsion*, [19] for *Semiconductor*, [25] for *AdvecDiffus* and [23] for *PotentialFlow*.

From the user's viewpoint, the implementation of a user class is simple. Only the input functions GenericP,Q,R,F,GN,GD (Table 12) are required for a user to define if the standard output data generated by the output function PostProcess in *Adaptor* are sufficient. Of course, the ouput function can be redefined in the user class.

In order to draw a diagram of streamlines (Postprocess in Table 13), one may solve the Poisson equation with a load function in terms of the computed velocity field. Again, the problem can be approximated by the standard FEM with now the generic form different from that of *Stokes*.

Table 12
Class *Stokes*: public *LSFEM*

Tasks	Member functions	Data members
Input	GenericP,Q,R,F,GN,GD	ReInv
Output	PostProcess	

Table 13
Class *StreamFunc*: public *FEM*

Tasks	Member functions
Input	GenericP,Q,R,F,GN,GD
Output	Postprocess

4. A refinement algorithm

The 1-irregular mesh refinement method was first proposed in [5] and has been implemented in many adaptive finite element research or commercial codes, e.g., FEARS [4], NFEARS [24], SAFES [11], ADAPT^{TM} [26]. Its implementation is usually based on a data structure designed for Fortran. Nevertheless, the general tree structure of adaptive meshes is very suitable for C++ because its hierarchical properties can be fully exploited by the very same properties of OOP. For example, the labeling algorithm developed by Rheinboldt and Mesztenyi [29] is a dynamic access algorithm on the tree which is designed for a procedural language such as FORTRAN and considerably improves data management of the unstructured refinement process. But the algorithm entails a complicated labeling strategy for nodes, edges, coordinates, elements, etc. Using C++, the work of labeling is in fact supported by the language itself; hence, producing a significant reduction of coding effort.

The refinement method is implemented here by a recursive algorithm based on the following two rules:

Rule 1. Let N_0, see Fig. 4, be an element to be refined. If the refinement level of N_0 is less than or equal to all the refinement levels of its neighboring elements, N_1, N_2, N_3, then it is refined without further refinement of its neighbors.

Rule 2. Otherwise, see Fig. 5, the neighboring elements, N_2 and N_3, with lower level are refined before N_0 is refined.

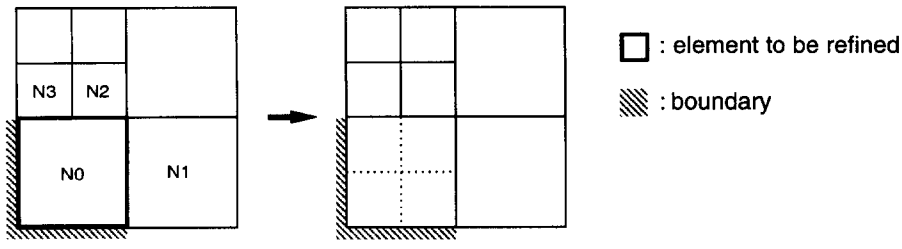

Fig. 4. Recursive refinement rule 1.

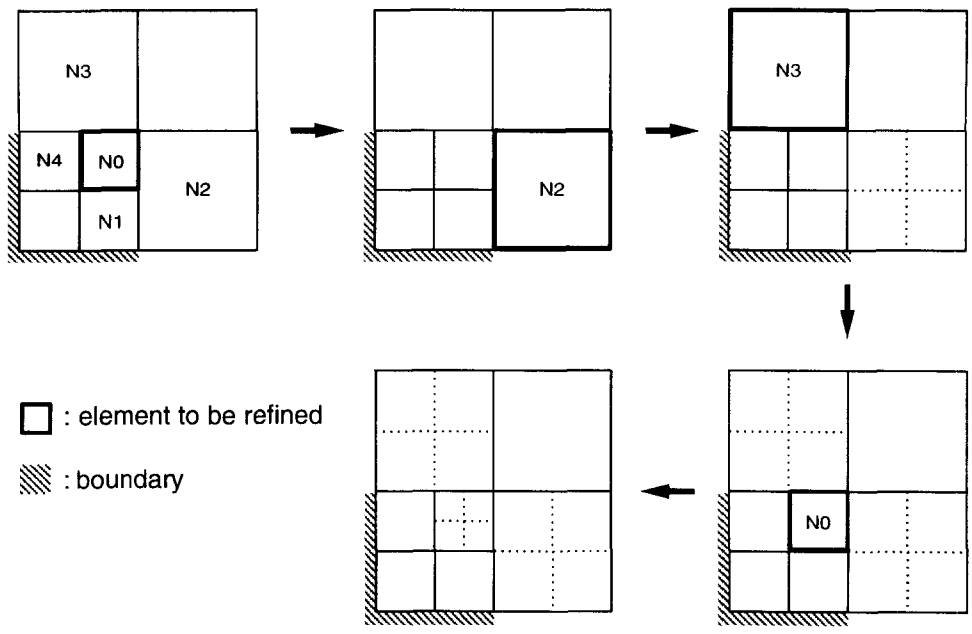

Fig. 5. Recursive refinement rule 2.

```
void Adaptor::AdaptiveRefine()
{
   Element *ptr;
   for(i=1; i≤4; i++)
      if(&elem_pointer→GetNeighbor(i)!=NULL)
         if(elem_pointer→GetElemLevel()>
         elem_pointer→GetNeighbor(i).Level())
         {
            ptr=elem_pointer;
            elem_pointer=&elem_pointer→GetNeighbor(i);
            AdaptiveRefine();
            elem_pointer=ptr;
         }
   ElementDivide();
}
```

Fig. 6. A recursive program of 1-irregular mesh refinement.

By these two rules, the 1-irregularity can be kept for all successive refinements. The recursive algorithm (Fig. 6) is defined in the member function AdaptiveRefine of the class *Adaptor*.

After the two rules have been checked, the majority of the refinement process takes place in ElementDivide. For simplicity, we only state the steps of the action in Fig. 7 where the element N_0 is admissible for refinement.

```
void Adaptor::ElementDivide()
{
   Step 1: A new family of N0 (the parent) with four offsprings is created.
   Step 2: Set the refinement level of each offspring to be that of the parent plus one.
   Step 3: The type (interior, Dirichlet, Neumann) of the offspring's edge is defined as an interior type
           if the edge is the interface of two siblings, otherwise it is inherited from that of the parent.
   Step 4: The relation (south, east, etc.) between each offspring and its neighbor is newly defined if
           the neighbor is its sibling, otherwise it is inherited from its parent.
   Step 5: Set the data (coordinates, node number, etc.) of new regular nodes in the family. The
           attributes of an old node are inherited by the offspring that shares the node with its parent.
}
```

Fig. 7. Refinement procedure.

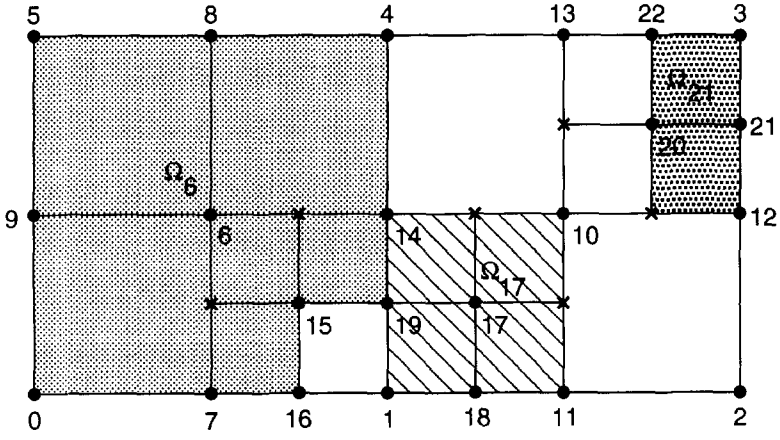

Fig. 8. A 1-irregular FE mesh.

Fig. 8 shows a particular 1-irregular mesh, i.e., the maximum number of irregular nodes (marked by $\times$) on an element side is one. In implementation, no degrees of freedom will be associated with these irregular nodes. Accordingly, the support of shape functions defining a basis for a finite element space should change adaptively with regular nodes, for example, the shaded subdomains Ω_6, Ω_{17} and Ω_{21} are the supports of the shape functions corresponding, respectively, to the regular nodes 6, 17 and 21. The finite element spaces so constructed preserve the conformality required by the standard finite element approximation provided that some special element constraint methods [10] are used to invoke continuity across interelement boundaries of elements of different size.

For FV approximation, control volumes have to adapt accordingly to their dual elements. Thus 23 control volumes in the dual mesh of Fig. 8 are shown, in dotted lines, in Fig. 9. Comparing Figs. 8 and 9, one can easily observe that it is impractical to develop refinement schemes based on control volumes, since shapes of volumes differ dramatically in comparison with those of elements, especially if a data structure were associated with volumes. Using the 1-irregular FE mesh refinement scheme, thirteen different patterns of boundaries of control volumes in the reference element can be classified depending on how many and where irregular nodes are located in the element, see Fig. 10.

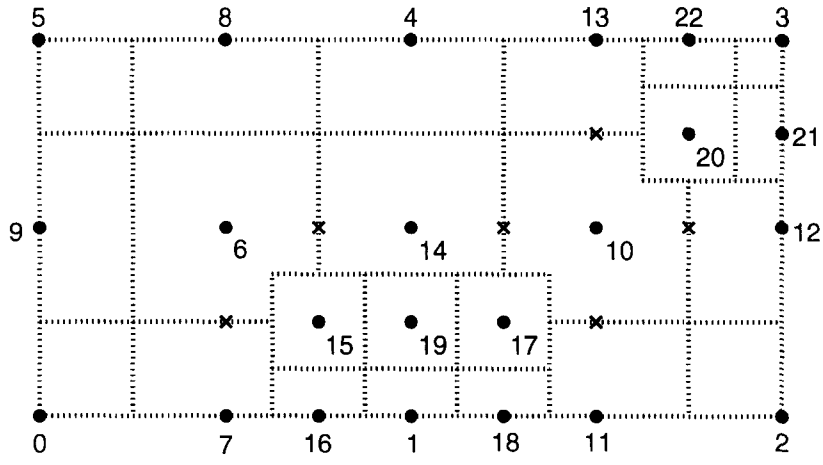

Fig. 9. A 1-irregular FV mesh.

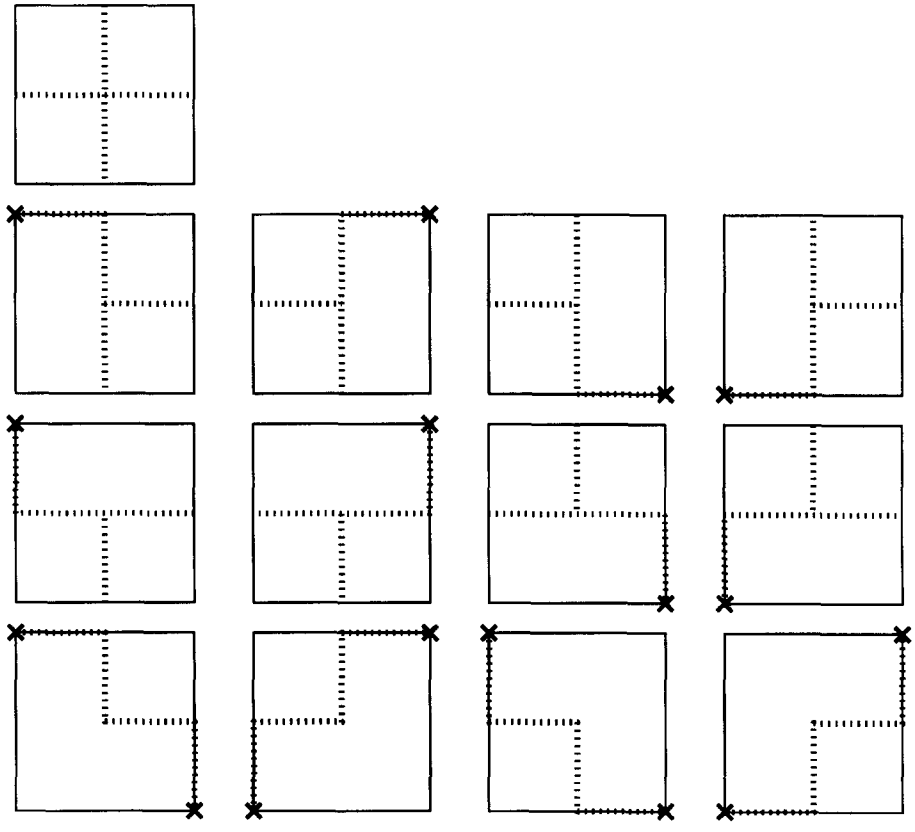

Fig. 10. Boundary patterns of control volumes in element.

5. Test model problems

In this section, we present three model problems for which four user classes *Stokes*, *StreamFunc*, *Semiconductor*, and *AdvecDiffus* are respectively derived from the abstract classes *LSFEM*, *FEM*, *VariatnlInq*, and *FVM*. The first model problem is described with more details from a user's viewpoint while the other two are briefly illustrated.

5.1. A 2D driven cavity flow

The use of least-squares principles for the approximate solution of the Navier–Stokes equations of incompressible flow has been increasingly studied in the past few years, we refer to [6] for further references. Our adaptive implementation of LSFEM is illustrated by a model problem of the Stokes equations cast into a first-order system involving the velocity, vorticity, and pressure as dependent variables.

5.1.1. Mathematical modeling

By introducing the vorticity $\omega = \partial v/\partial x - \partial u/\partial y$, the 2D dimensionless Stokes equations can be written as

$$L\boldsymbol{u} = \begin{bmatrix} 0 & 0 & \nu\frac{\partial}{\partial y} & \frac{\partial}{\partial x} \\ 0 & 0 & -\nu\frac{\partial}{\partial x} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & 0 & 0 \\ \frac{\partial}{\partial y} & -\frac{\partial}{\partial x} & 1 & 0 \end{bmatrix} \begin{bmatrix} u \\ v \\ \omega \\ p \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ 0 \\ 0 \end{bmatrix} = \boldsymbol{f} \quad \text{in } \Omega, \tag{5.1a}$$

where u and v are the x and y components of velocity, p the total pressure, ν the inverse of Reynolds number (ReInv) and f_1 and f_2 the given body forces. Let boundary conditions be written in operator form as

$$R\boldsymbol{u} = \boldsymbol{g}_D \quad \text{on } \partial\Omega. \tag{5.2b}$$

The LSFEM for (5.1) is based on the minimization of the least-squares functional

$$J(\boldsymbol{v}) = \int_\Omega (L\boldsymbol{v} - \boldsymbol{f}) \cdot (L\boldsymbol{v} - \boldsymbol{f})\,\mathrm{d}x\,\mathrm{d}y$$

on the Sobolev space

$$S = \{\boldsymbol{v} \in [H^1(\Omega)]^4 \colon R\boldsymbol{v} = \boldsymbol{g}_D \text{ on } \partial\Omega\}.$$

The minimizer $\boldsymbol{u} \in S$ of J equivalently solves the variational problem

$$B(\boldsymbol{u}, \boldsymbol{v}) := \int_\Omega (L\boldsymbol{u} \cdot L\boldsymbol{v})\,\mathrm{d}x\,\mathrm{d}y = \int_\Omega (\boldsymbol{f} \cdot L\boldsymbol{v})\,\mathrm{d}x\,\mathrm{d}y =: F(\boldsymbol{v}), \quad \forall \boldsymbol{v} \in S. \tag{5.2}$$

In flow simulation, it is often necessary to calculate the stream function of the flow. Using the velocity field, the stream function ψ of (5.1) can be determined by solving another variational problem

$$B(\psi, w) := \int_\Omega (\nabla\psi \cdot \nabla w)\,\mathrm{d}x\,\mathrm{d}y = \int_\Omega (u_y - v_x)w\,\mathrm{d}x\,\mathrm{d}y =: F(w). \tag{5.3}$$

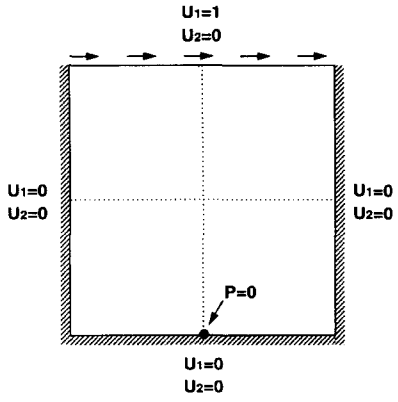

Fig. 11. Problem definition.

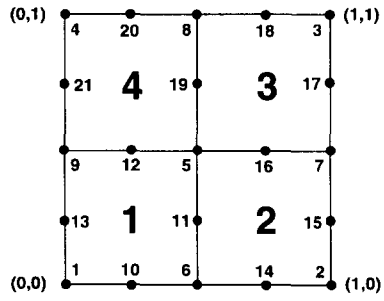

Fig. 12. Initial mesh.

5.1.2. Definition of input member functions

For the 2D driven cavity flow problem defined in Fig. 11, a list of C++ programs of the user classes *Stokes* and *StreamFunc*, an input file containning the data of the initial mesh, and the main program are given in the appendix.

5.1.3. Output

Starting with the initial mesh shown in Fig. 12, AdaptC++ automatically generates a sequence of approximate solutions on the corresponding adaptive meshes until the relative error is less than 0.01 or the total number of degrees of freedom is greater than 100,000 a default value in *Adaptor*. The final mesh is given in Fig. 13 which was obtained by invoking the member function PostProcess of *Adaptor*. Moreover, the diagrams of the computed vorticity, velocity, and stream function are given in Figs. 14, 15 and 16, respectively. These diagrams were obtained by means of the same function PostProcess redefined in *Stokes*. Figs. 13–16 evidently show that AdaptC++ is effectively capturing the singular behavior of the problem which occurs at the top corners.

5.2. A simulation of reverse biased pn-junctions

The use of the class *VariatnlIneq* is demonstrated by a device simulation model given in [19]. We consider a device occupying a bounded polygonal domain $\Omega \subset \mathbb{R}^2$ whose stationary behavior is ruled by the drift-diffusion equations

$$-\mathrm{div}(\varepsilon\nabla\psi) = q(D - n + P),$$
$$\mathrm{div}J_n = qR, \qquad J_n = q(D_n\nabla n - \mu_n n\nabla\psi),$$
$$\mathrm{div}J_p = -qR, \qquad J_p = -q(D_p\nabla p - \mu_p p\nabla\psi),$$

where usually the electric potential ψ and the carrier concentrations n and p for electrons and holes are unknown while the permittivity ε, the doping profile D, the elementary charge q, the electron and hole diffusivities D_n and D_p, the electron and the hole mobilities μ_n and μ_p, and the recombination-generation rate R are given parameters of the problem.

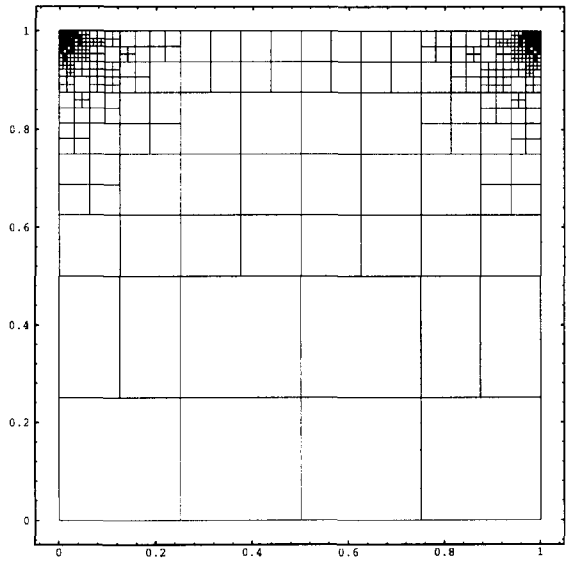

Fig. 13. Final mesh.

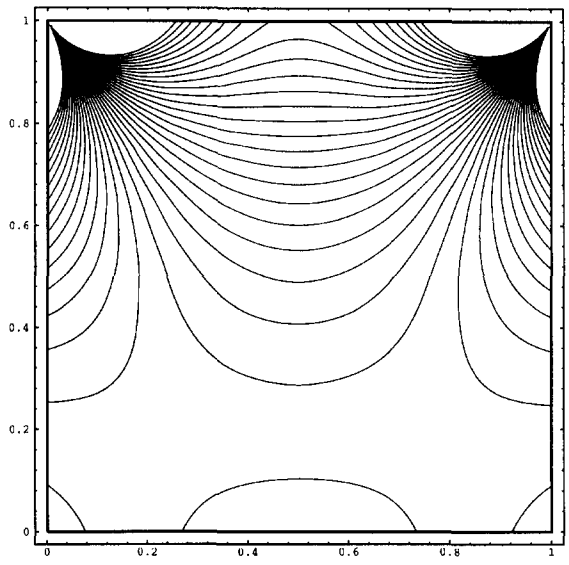

Fig. 14. Vorticity contour.

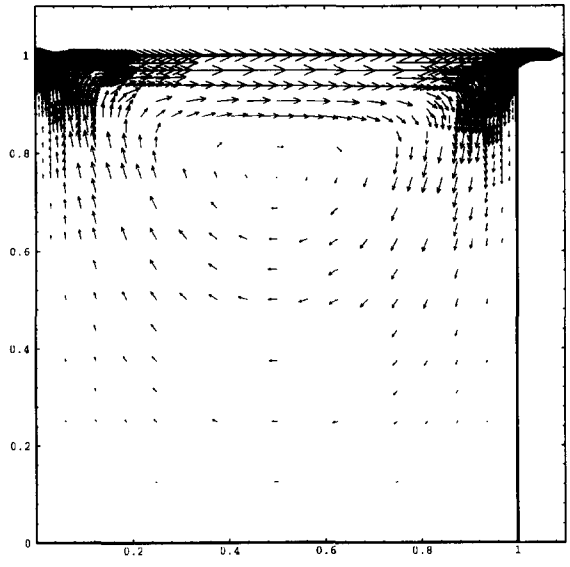

Fig. 15. Velocity field.

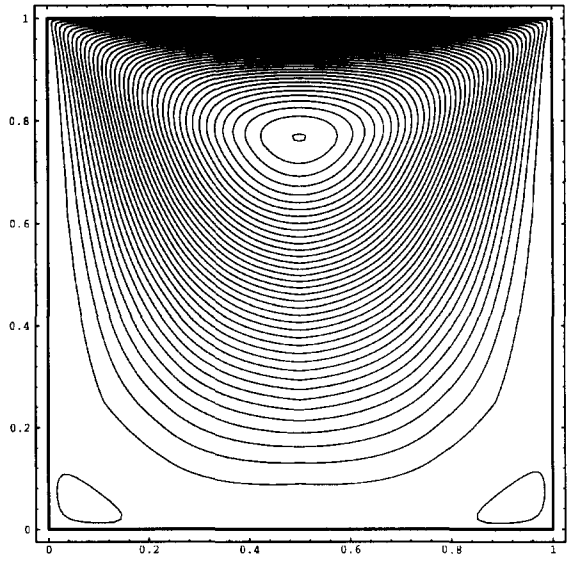

Fig. 16. Streamline

The boundary $\partial\Omega$ of Ω is spilt into (ohmic) contacts $\partial\Omega_C$ and insulating segments $\partial\Omega_I$. This leads to Dirichlet boundary conditions for ψ, n and p on $\partial\Omega_C$ and vanishing electric field $\nabla\psi$ and current densities J_n, J_p on $\partial\Omega_I$. This model whose advantages and limits are thoroughly discussed in [31] can be considerably simplified under strongly reverse bias conditions. The geometric data of the model is given in Fig. 17.

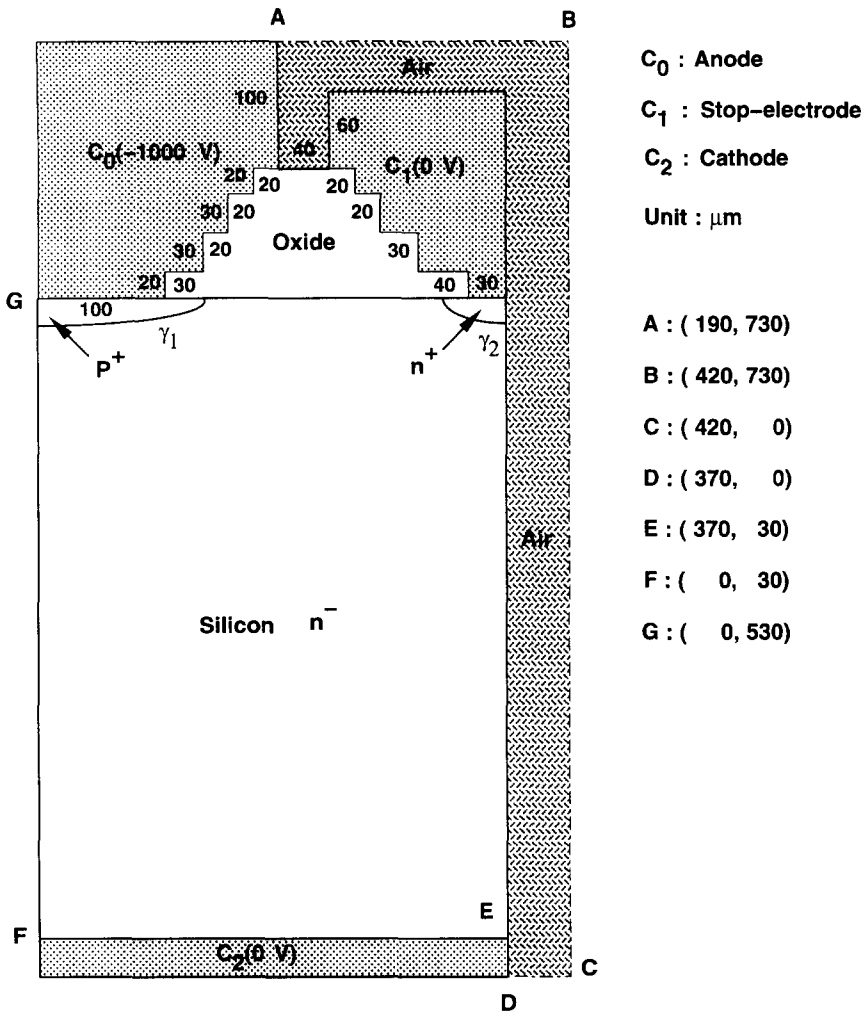

Fig. 17. Computational geometry of Model 5.2.

The model problem can be formulated as (2.6) with

$$B(\psi, v) = \int_{\Omega} \varepsilon \nabla \psi \cdot \nabla v \, \mathrm{d}x \, \mathrm{d}y, \qquad F(v) = q \int_{\Omega} Dv \, \mathrm{d}x \, \mathrm{d}y$$

for all functions u and v in the convex set

$$K = \{ v \in H(\Omega) | \ -\Phi_0 \leqslant v(x, y) \leqslant 0 \text{ a.e. on } \Omega \},$$

where

$$H(\Omega) = \{ \varphi \in H^1(\Omega) | \ \varphi = -\psi_0 \text{ on } C_0, \ \varphi = 0 \text{ on } \partial\Omega_c \backslash C_0 \}.$$

The permittivity ε is given by $\varepsilon = \varepsilon_0 \varepsilon_r$, where

$$\varepsilon_0 = 8.854 \cdot 10^{-14}, \qquad \varepsilon_r = \begin{cases} 1 & \text{air,} \\ 3.9 & \text{oxide,} \\ 11.7 & \text{silicon.} \end{cases}$$

The doping profile D has the values

$$D = \begin{cases} -10^{17}\ \text{cm}^{-3} & \text{in } p^+, \\ 10^{19}\ \text{cm}^{-3} & \text{in } n^+, \\ 8 \cdot 10^{13}\ \text{cm}^{-3} & \text{in } n^-. \end{cases}$$

And the interfaces γ_1 and γ_2 of $p^+ - n^-$ and $n^+ - n^-$ are given as

$$\gamma_1 := \frac{x^2}{130^2} + \frac{(y - 0.053)^2}{20^2} = 1,$$

$$\gamma_2 := \frac{(x - 0.037)^2}{50^2} + \frac{(y - 0.053)^2}{5^2} = 1.$$

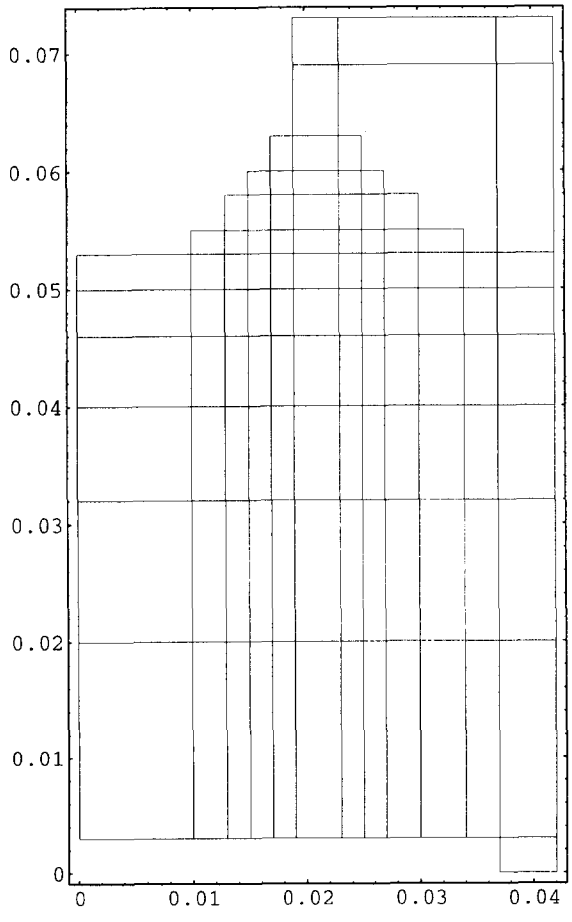

Fig. 18. Initial mesh of Model 5.2.

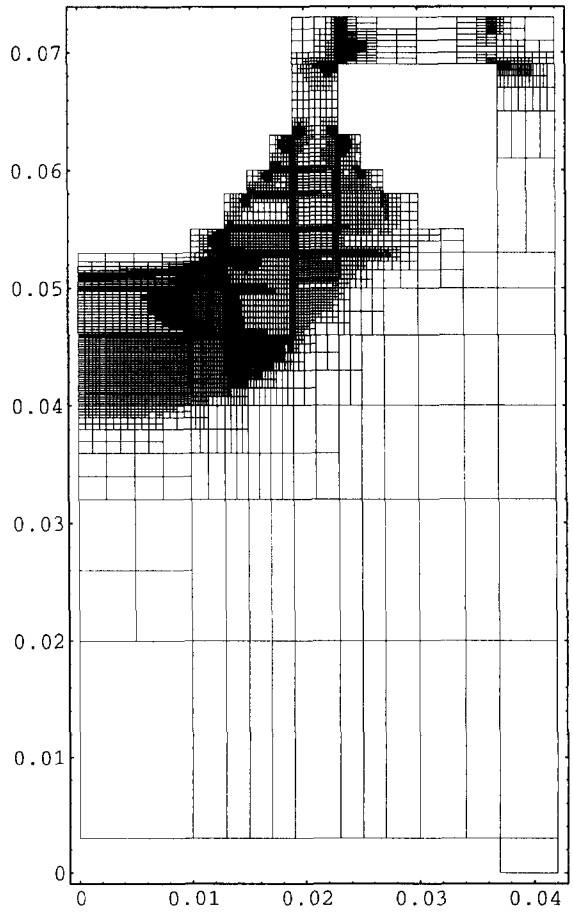

Fig. 19. Final mesh of Model 5.2.

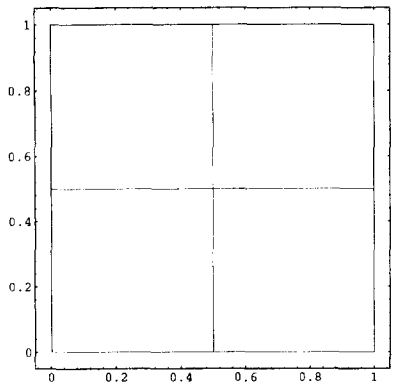

Fig. 20. Initial mesh of Model 5.3.

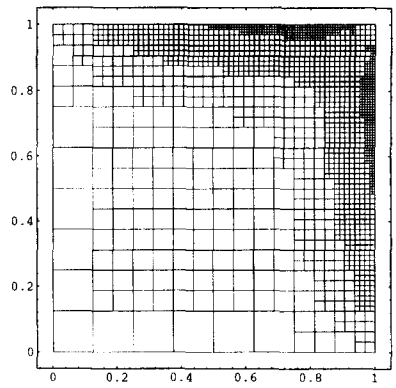

Fig. 21. Final mesh of Model 5.3.

The elementary charge $q = 1.602 \cdot 10^{-19}$. The boundary conditions are

$$\psi|_{\partial\Omega_C} = \begin{cases} -\psi_0 & \text{on } C_0, \\ 0 & \text{on } C_1, \\ 0 & \text{on } C_2, \end{cases}$$

with $\psi_0 = 1000$. Along AB a linear increase of the potential is assumed which then is kept zero along $BCDE$, and this device is assumed to be isolated along $\partial\Omega_I = FG$ so that

$$\frac{\partial}{\partial n}\psi\Big|_{\partial\Omega_I} = 0.$$

The initial and final meshes are given in Figs. 18 and 19, respectively.

5.3. An advection–diffusion problem

The finite volume class *FVM* is tested by the following linear advection–diffusion problem given in [25]

$$-\nabla \cdot (\nu \nabla u - \boldsymbol{a}u) + 2u = f, \quad \text{in } \Omega = (0,1)^2,$$
$$u = 0, \quad \text{on } \partial\Omega,$$

where $\boldsymbol{a} = (1,1)$, ν the inverse of Reynolds number and the body force f is given so that the exact solution $u = xy(1 - \exp((x-1)/\nu))(1 - \exp((y-1)/\nu))$. This is a typical singularly perturbed problem, whose solution has elliptic boundary layers along the two sides $\{(1,y) \mid 0 < y < 1\}$ and $\{(x,1) \mid 0 < x < 1\}$. For $\nu = 0.1$, the initial and final FE meshes are given in Figs. 20 and 21, respectively. Note that the finite volumes are implicitly embedded in these finite element meshes.

6. Concluding remarks

It is shown that an integration of different numerical methods applying to various mathematical model problems in a unified adaptive environment is possible provided that all adaptive methods share

a common data structure. Toward this unified approach, the following aspects of AdaptC++ are also discussed and illustrated.

Theoretical aspect: We present a generic formulation suitable for adaptive FE and FV approximations of first-order and second-order systems of boundary value problems, with a subclass corresponding to second-order variational inequalities. A posteriori error estimation and refinement scheme are two basic components in adaptive methodology. We have used the weak residual error estimators [1,20,21] which are simple to implement in the sense that they disregard input model problems and only rely on very general linear and bilinear forms and boundary conditions. This is in contrast to other estimators, see, e.g., [4], that depend on the problem, the location of singularities, the quantity of interest (displacement, stress, etc.), and the spectral order of approximation. The data structure depends very much on the refinement scheme. To unify adaptive FE and FV methods, we have chosen the 1-irregular mesh refinement scheme since the complexity of the construction of adaptive control volumes increases as the irregularity increases, see Fig. 10.

Implementational aspect: The concepts of inheritance and encapsulation of OOP play the most important role in our development of the code. Both of the generic formulation and the refinement scheme lead naturally to a hierarchical structure which is perfectly compatible with the hierarchical property of the programming language itself. Data encapsulation helps define classes with clearly set contours. Inheritance organizes classes into a tree. Consequently, the program is much more readable and modular than in a procedural language. It definitely promotes modification, extension, maintenance, and reuse of the code. In addition, the implementation of an object-oriented program requires less time and produces small programs. The size of AdaptC++ amounts to 297 KB. We also feel that the code is very easy to use. For example, the definition of user classes is quite simple and general. Essential boundary conditions can be easily modified or changed since they are implemented as an auxiliary class. The auxiliary class and the initial mesh file are independent of the numerical methods. Consequently, the same input information (model problem, boundary conditions, and initial data) can be used for all methods (FEM, FVM, conjugate gradient methods, multigrid methods, etc.) provided by the code. Moreover, the performance of different methods can be evaluated on a more uniform basis.

We next briefly remark on the choice of the language C++. Smalltalk and C++ are the most widely used OOP languages today [33]. These two languages differ on some basic design tradeoffs such as weak typing vs. strong typing, dynamic binding vs. static binding, single inheritance vs. multiple inheritance, etc. [33]. Smalltalk, used in [36], is a pure object language that is very simple to learn and is relatively easier to use than C++. However, since C is currently the dominant development language in the PC and workstation market, there are many well-developed numerical programs in C that can be readily and easily incorporated into C++ programs. For example, the source codes of the linear system solvers listed in the flow chart of Fig. 1 are written in C. Our preference of C++ is primarily based on our experience with the C environment and its easy translation to OOP.

The current version of AdaptC++ only addresses steady 2D problems. Its extension to transient 3D problems evidently can be based on the same governing principles, although this is by no means a straightforward modification. The most fundamental change will be the redesign of the data structure, adaptive refinement algorithm, and the construction of shape functions associated with 1-irregular 3D meshes.

Acknowledgements

The authors would like to thank the referees for their helpful comments and suggestions.

Appendix A

In this section, we give a complete list of user programs (Figs. A.1–A.7) for the 2D driven cavity flow discussed in Section 5.1. The adaptive procedure can be easily understood by the main program illustrated in Fig. A.1.

For the model problem (5.1a), the corresponding generic functions $\boldsymbol{p}$ and $\boldsymbol{q}$ in (2.1) are zero functions. The user-defined functions GenericP(x,y) (Fig. A.4) and GenericQ(x,y) need not explicitly define each one of zero coefficient functions $CP(i,k)$, $CQ(i,k)$, etc. in (2.2), since they are initialized to zeros in the constructor of the class *Adaptor*.

```
# include "Stokes.hpp"
int main(int argc, char *argv[ ])
{
    Stokes driven;
    driven.ReadInitMesh();
    do
    {
        driven.GetStiffMatrix();
        driven.GetLoadVector();
        driven.ImpBdryCond();
        driven.Solve();
        driven.GetApproSolNorm();
        driven.GetApproErrNorm();
        driven.AdaptiveRefine();
        driven.RelativeError();
    } while( driven.Tolerance() > driven.RelativeError())
    driven.PostProcess();
}
```

Fig. A.1. The main program for the 2D driven cavity flow.

```
# (initial nodes,elements)
(21,4)

# node-index : coordinate ( x, y )
1:( 0 , 0 ) 2:( 1 , 0 ) 3:( 1 , 1 ) 4:( 0 , 1 ) 5:( 0.5, 0.5) 6:( 0.5, 0 )
7:( 1 , 0.5) 8:( 0.5, 1 ) 9:( 0 , 0.5) 10:(0.25, 0 ) 11:(0.5 ,0.25)
```

Fig. A.2. The initial mesh (Fig. 12).

```
12:(0.25,0.5 ) 13:( 0 ,0.25) 14:(0.75, 0 ) 15:( 1 ,0.25) 16:(0.75, 0.5)
17:( 1 ,0.75) 18:(0.75, 1 ) 19:(0.5 ,0.75) 20:(0.25, 1 ) 21:( 0 ,0.75)

# element-index : node ( n1, n2, n3, n4, n5, n6, n7, n8 ),
# bdry-type ( s, e, n, w ), neighbor ( s, e, n, w )
1:(1,6,5,9,10,11,12,13),(D,I,I,D),(0,2,4,0)
2:(6,2,7,5,14,15,16,11),(D,D,I,I),(0,0,3,1)
3:(5,7,3,8,16,17,18,19),(I,D,D,I),(2,0,0,4)
4:(9,5,8,4,12,19,20,21),(I,I,D,D),(1,3,0,0)
```

Fig. A.2. (continued).

```
Stokes::Stokes()
{
   ReInv = 1.0;
   sol_dim = 4;
   elem_order = BIQUADRATIC;
   solver_type = CG;
   norm_type = ENERGY;
   errest_type = BLTENT;
   refine_criterion = 0.1;
   tolerance = 0.01;
   post_process = YES;
}
```

Fig. A.3. The constructor of the class *Stokes* (initialization).

```
void Stoke::GenericP(x,y)
{
}

void Stoke::GenericR(x,y)
{
   CR(3,3)=1.0;
   CRX(1,4)=1.0;
   CRX(2,3)=-ReInv;
   CRX(3,2)=-1.0;
   CRX(4,1)=1.0;
   CRY(1,3)=ReInv;
   CRY(2,4)=1.0;
   CRY(3,1)=1.0;
   CRY(4,2)=1.0;
}
```

Fig. A.4. Member functions of the class *Stokes*.

```
BdryData Stokes::GenericGD(x,y)
{
   BdryData bc;
   if (y==1.0)
   {
      if (x!=0.0 && x!=1.0)
      {
         bc.SetBCData(1,1.0);
         bc.SetBCData(2,0.0);
      }
   }
   else
   {
      bc.SetBCData(1,0.0);
      bc.SetBCData(2,0.0);
   }
   if (x==0.5 && y==0.0)
   {
      bc.SetBCData(4,0.0);
      /* The first argument represents the fourth component
      the pressure which is prescribed to be zero,
      the value of the second argument. */
   }
   return bc;
}
```

Fig. A.5. A member function of the class *Stokes*.

```
void StreamFunc::GenericP(x,y)
{
   CPX(1,1)=-1.0;
   CPY(1,1)=-1.0;
}
```

Fig. A.6. A member function of the class *StreamFunc*.

```
BdryData StreamFunc::GenericGD(x,y)
{
   BdryData bc;
   bc.SetBCData(1,0.0);
   return bc;
}
```

Fig. A.7. A member function of the class *StreamFunc*.

References

[1] S. Adjerid and J.E. Flaherty, Second-order finite element approximations and a posteriori error estimation for two-dimensional parabolic systems, *Numer. Math.* 53 (1988) 183–198.
[2] M. Ainsworth, J.T. Oden and C.Y. Lee, Local a posteriori error estimators for variational inequalities, *Numer. Methods Partial Differential Equations* 9 (1993) 23–33.
[3] A.K. Aziz and J.-L. Liu, A weighted least squares method for the backward–forward heat equation, *SIAM J. Numer. Anal.* 28 (1991) 156–167.
[4] I. Babuška and A. Miller, The post-processing approach in the finite element methods, *Internat. J. Numer. Methods Engrg.* 20 (1984) 1085–1109, 1111–1129 and 2311–2324.
[5] I. Babuška and W.C. Rheinboldt, Error estimates for adaptive finite element computations, *SIAM J. Numer. Anal.* 15 (1978) 736–754.
[6] P.B. Bochev and M.D. Gunzburger, Accuracy of least-squares methods for the Navier–Stokes equations, *Comput. & Fluids* 22 (1993) 549–563.
[7] C. L. Chang, A least-squares finite element method for the Helmholtz equation, *Comput. Methods Appl. Mech. Engrg.* 83 (1990) 1–7.
[8] K. Clark, J.E. Flaherty and M.S. Shephard, eds., Proceedings of the 3rd ARO Workshop on Adaptive Methods for Partial Differential Equations, *Appl. Numer. Math.* 14 (1–3) (1994).
[9] P.I. Crumpton, J.A. Mackenzie and K.W. Morton, Cell vertex algorithms for the compressible Navier-Stokes equations, *J. Comput. Phys.* 109 (1993) 1–15.
[10] L. Demkowicz, J.T. Oden, W. Rachowicz and O. Hardy, Toward a universal h-p adaptive finite element strategy. Part 1. Constrained approximation and data structure, *Comput. Methods Appl. Mech. Engrg.* 77 (1989) 79–112.
[11] J.C. Diaz, R.E. Ewing, R.W. Jones, A.E. McDonald, D.U. von Rosenberg and L.M. Uhler, Self-adaptive local grid refinement application in enhanced oil recovery, in: *Proceedings Fifth Internat Sympos on Finite Elements and Flow Problems*, Austin, TX (1984).
[12] Y. Dubois-Pèlerin, T. Zimmermann and P. Bomme, Object-oriented finite element programming: II. A prototype program in Smalltalk, *Comput. Methods Appl. Mech. Engrg.* 98 (1992) 361–397.
[13] G.L. Fenves, Object-oriented programming for engineering software development, *Engrg. Comput.* 6 (1990) 1–15.
[14] B.W.R. Forde, R.O. Foschi and S.F. Stiemer, Object-oriented finite element analysis, *Comput. & Structures* 35 (1990) 355–374.
[15] R. Glowinski, J.L. Lions and R. Trémoliéres, *Numerical Analysis of Variational Inequalities* (North-Holland, Amsterdam, 1981).
[16] R.H.W. Hoppe and R. Kornhuber, Adaptive multilevel method for obstacle problems, *SIAM J. Numer. Anal.* 31 (1994) 301–323.
[17] B.-N. Jiang and C.L. Chang, Least-squares finite elements for Stokes problem, *Comput. Methods Appl. Mech. Engrg.* 78 (1990) 297–311.
[18] D. Kinderlehrer and G. Stampacchia, *An Introduction to Variational Inequalities and Their Applications* (Academic Press, New York, 1980).
[19] R. Kornhuber and R. Roitzsch, Self-adaptive finite element simulation of bipolar, strongly reverse-biased pn-junctions, *Comm. Numer. Methods Engrg.* 9 (1993) 243–250.
[20] J.-L. Liu, On weak residual error estimation, *SIAM J. Sci. Comput.*, to appear.
[21] J.-L. Liu and W.C. Rheinboldt, A posteriori finite element error estimators for indefinite elliptic boundary value problems, *Numer. Funct. Anal. Optim.* 15 (1994) 335–356.
[22] R.I. Mackie, Object oriented programming of the finite element method, *Internat. J. Numer. Methods Engrg.* 35 (1992) 425–436.

[23] S.F. McCormick, *Multilevel Adaptive Methods for Partial Differential Equations* (SIAM, Philadelphia, 1989).
[24] C.K. Mesztenyi and W.C. Rheinboldt, NFEARS: A nonlinear adaptive finite element solver, Computer Science TR-1946, University of Maryland, College Park (1988).
[25] J.J.H. Miller and S. Wang, An exponentially fitted finite volume method for the numerical solution of 2D unsteady incompressible flow problems, *J. Comput. Phys.* 115 (1994) 56–64.
[26] J.T. Oden, J.M. Bass, C.-Y. Huang and C.W. Berry, Recent results on smart algorithms and adaptive methods for two- and three-dimensional problems in computational fluid mechanics, *Comput. & Structures* 35 (1990) 381–396.
[27] J.T. Oden and N. Kikuchi, Theory of variational inequalities with applications to problems of flow through porous media, *Internat. J. Engrg. Sci.* 18 (1980) 1173–1284.
[28] J.T. Oden and N. Kikuchi, *Contact Problems in Mechanics* (SIAM, Philadelphia, 1988).
[29] W.C. Rheinboldt and C.K. Mesztenyi, On a data structure for adaptive finite element mesh refinements, *ACM Trans. Math. Software* 6 (1980) 166–187.
[30] S.-P. Scholz, Elements of an object-oriented FEM++ program in C++, *Comput. & Structures* 43 (1992) 517–529.
[31] S. Selberherr, *Analysis and Simulation of Semiconductor Devices* (Springer, New York, 1984).
[32] B. Stroustrup, *The C++ Programming Language* (Addison-Wesley, Reading, MA, 1986).
[33] A. Taylor, *Object-Oriented Information Systems: Planning and Implementation* (Wiley, New York, 1992).
[34] R.E. White, *An Introduction to the Finite Element Method with Applications to Nonlinear Problems* (Wiley, New York, 1985).
[35] O.C. Zienkiewicz, Computational mechanics today, *Internat. J. Numer. Methods Engrg.* 34 (1992) 9–33.
[36] T. Zimmermann, Y. Dubois-Pèlerin and P. Bomme, Object-oriented finite element programming: I. Governing principles, *Comput. Methods Appl. Mech. Engrg.* 98 (1992) 291–303.