An object-oriented framework for the implementation of adjoint techniques in the design and control of complex continuum systems

Rajiv Sampath and Nicholas Zabaras*†

Sibley School of Mechanical and Aerospace Engineering, 188 Frank H. T. Rhodes Hall, Cornell University, Ithaca, NY 14853-3801, U.S.A.

SUMMARY

Specific object-oriented software design concepts are elaborated for a novel implementation of a class of adjoint optimization problems typical of the infinite-dimensional design and control of continuum systems. For clarity, the design steps and ideas are elucidated using an inverse natural convection design problem. Effective application of software design concepts such as inheritance, data encapsulation, information hiding, etc., is demonstrated through instances from the example considered. Two test numerical examples are considered and the CPU statistics for one of these problems are compared with those corresponding to a procedural implementation of the same problem. The numerical examples include a three-dimensional inverse design problem that demonstrates the effectiveness of the present object-oriented approach in developing dimension-independent robust design codes. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: object-oriented programming; adjoint method; inverse natural convection; finite element method; functional optimization; convection–diffusion problems

1. INTRODUCTION

Many fields of science and engineering rely on various mathematical models to describe dynamic processes in nature or to analyse information gathered from experiments. These models act as approximate prototypes of an actual system and can be used to simulate the physical system to a scale depending on the accuracy of the model. Such models typically involve either a single differential equation (PDE) or a system of PDEs and corresponding boundary conditions and initial data. Due to the complex nature of these equations with probable non-linearity and inter-couplings, simulation of continuum systems typically requires considerable amount of numerical
calculation. Moreover, in a typical commercial application, the software developed must provide for user friendly interfaces linking to visualization software and other accessories. As a consequence, an average PDE application software requires considerable amount of human and machine power. Another important issue that a numerical software developer needs to consider is allowing provisions for changes or expansions, for example, allowing the use of various linear/non-linear system solvers or implementing additional inter-coupling algorithms between the various PDEs. Developing codes which are dimension-independent is also a typical requirement.

Traditionally, numerical software developers have based their work on procedural languages such as FORTRAN or C. In large-scale applications, the procedures involved are clustered into libraries and are usually linked to external software such as LAPACK for linear algebra computations [1]. These ideas remained stagnant for quite sometime and software management seemed a daunting task.

Early 1990s saw a sudden trend towards applying the paradigms of object-oriented programming (OOP) to numerical computing problems [2]. This approach seemed quite attractive due to its well-defined mechanisms for modular design, re-use of code and for creating flexible applications that can be easily extended with respect to problem classes as well as solution techniques. Since then, there has been considerable progress in the application of object-oriented techniques to numerical computing problems involving systems of PDEs [3–7]. Important works that need mention include the Diffpack [8] project where OOP techniques were used to develop software libraries appropriate for the finite element and finite differences analysis and simulation of any general system of PDEs. There has also been comprehensive work published in the novel implementation of more conventional finite element (FEM) simulator codes for PDEs using object-oriented approaches. These include contributions in References [9–12] and in many more recent works.

Over the last two decades, the field of numerical computing has branched itself into two distinct subfields. While a large group of researchers is still concerned with the simulation of direct problems and with increasing the accuracy, efficiency, complexity and scales over which such simulations are conducted, a smaller separate group has emerged, building a sub-field which deals with the design and control of processes using computational optimization techniques.

These new problems are usually referred to by epithets such as inverse problems or optimal design/control problems. The solution techniques proposed for their solution have extended the finite- and infinite-dimensional optimization theory [13, 14] to the mathematical design and control of processes that are governed by a system of PDEs. Optimization techniques have been applied to various control and design problems. Examples of fluid flow control problems are given in References [15–22] and further examples in areas such as reaction diffusion processes, weather prediction, economics and planning, pollution control, ground water modelling, wave propagation, vibration, etc., can be found in References [23–27] and in the references therein.

A typical design or control problem takes the form of a continuum system governed by one or more PDEs with insufficient or no boundary conditions available in part of the boundary. In addition, over-specified boundary conditions are supplied in another part of the boundary or within the domain. These problems are generally ill-posed and a quasi-solution is being sought in an optimization sense. In simple terms, the ill-posed problem is converted to a well-posed functional optimization problem defined over both space and time. The minimization problem requires the calculation of the gradient of the cost functional. Of interest to this paper is the solution of design and control problems using infinite-dimensional optimization schemes in which the gradient of the cost functional is calculated using a properly defined continuum adjoint problem. Calculation
of the step size to proceed through a gradient-based optimization algorithm typically requires a solution of a continuum sensitivity problem. Both the adjoint and sensitivity problems require the solution of the direct problem.

The solution of design and control continuum problems requires a significant programming effort due to their special structure necessitating solution of the systems of PDEs in the direct, adjoint and sensitivity problems. These requirements far surpass the computations involved in a typical simulation of a direct problem. These design problems are quite amenable to an object-oriented approach due to the similar structure of the PDEs involved in the three problems and the solution techniques used to solve these problems. Typically, the adjoint and sensitivity problems are defined as systems of PDEs that are in nature similar to the PDE system of the direct problem. Moreover, the solution techniques usually do not differ between each of the three problems. Even though there has been considerable work performed in the area of objects-based design applied to FEM simulation problems, there is no evidence of any research efforts in extending these concepts to design optimization problems. The main goal of the current work is to provide such a framework for a class of design optimization problems formulated using the adjoint technique. For simplicity of the presentation, a typical but simple case of an inverse natural convection problem is considered that consists of two coupled systems of PDEs (the energy convection/diffusion equation and the incompressible Navier–Stokes equations). However, the ideas presented in this paper are general and applicable to a whole class of adjoint-based optimization problems.

The rest of this article is organized as follows. In the next section, the basics of an object-oriented approach to the finite element solution of a continuum system are reviewed and a few details on the Diffpack [8] library of classes for analysis of systems governed by partial differential equations is provided. Diffpack is used here as a platform for the development of the present work but any other object-oriented frameworks can be employed as well. In Section 3, the direct and inverse design natural convection problems are defined. This is followed by Section 4 in which the design of the various classes and the specific advantages of utilizing an object-oriented C++ platform are highlighted. Finally, Section 5 presents typical numerical examples of inverse natural convection design problems where issues dealing with the accuracy and CPU time requirements of such large-scale OOP simulations are addressed.

2. OBJECT-ORIENTED CONCEPTS AND DETAILS ON Diffpack

The main elements of object-oriented programming have been widely published in References [28–30]. However, for completeness of the presentation, a few general concepts are addressed here and more details can be obtained in the above references.

The primitive element of object-oriented programming is an object, which is an instance of a class [31]. An object is characterized by an identity, a state and a behaviour. In simple terms, the identity of the object is defined by grouping a set of operations and data which are related into a single entity called a class. The process of grouping data and functions into a capsule is referred to as encapsulation. The state is represented by the current memory of the object and is implemented by variables representing the data. The behaviour is how the object interacts with other objects, i.e. what information can be obtained from the object. Not all the data encapsulated into an object is usually retrievable by other objects. An object typically has a public interface and a private representation and usually keeps the two facets quite distinct. This principle is known
as information-hiding. Information-hiding allows us to remove from view some portion of those features which have been encapsulated by the object. This is useful in increasing the leverage gained from abstraction and to design code that can be more easily modified, maintained and extended.

Other important elements in object-oriented terminology include inheritance and polymorphism. Inheritance is the ability of one class to define the behaviour and data structure of its instances as a superset of the definition of another class or classes. In other words, one can say that one class is just like another class except that the new class includes something extra or different. Inheritance provides us with a mechanism for classification. With it, one can create taxonomies of classes. Polymorphism is a feature associated with inheritance. This is the ability of several objects of different classes to respond to the same message in a different way. In other words, a message can be defined in several classes with different implementations. This is done through defining what is known as virtual functions. With this preliminary background in object-oriented methods, a brief survey of the Diffpack software is given next for easy reading of the programming concepts presented in this paper. For further details the reader is advised to consult References [32–34]. However, it would be appropriate to emphasize that even though a few programming instances would be particular to Diffpack, the ideas presented in this paper are general and applicable to any object-oriented environment.

The Diffpack software is essentially a collection of C++ classes organized in libraries. The application area is mainly the numerical solution of PDEs with a particular focus on finite element methods. The libraries are especially constructed for rapid prototyping of simulators for new problems using classes at a high abstraction level. The CPU-intensive numerics are implemented in low-level classes to provide numerical efficiency. The main classes and functionalities follow next.

In the following presentation, real denotes a Diffpack macro that represents scalar real numbers and allows an easy switch between single and double precision. Pointers are treated in Diffpack using the class HandleID. This class allows for the automatic deletion of an object when all pointers bound to that object have been deleted.

Continuous fields are treated in Diffpack using the classes FieldFE (for scalar fields) and FieldsFE (for vector fields). An object of these classes is bound with a particular finite element grid (object of class GridFE). The functionality of the class FieldFE allows the calculation of the nodal values of the field, the calculation of the field at any point within the domain, re-binding of the field with another grid, etc. Also, note that each GridFE object contains geometric information as well as information specific to a given boundary value problem (e.g. problem-specific boundary indicators).

The class LinEqAdm provides the necessary structure for the selection of various linear algebraic equation solvers. The user via a menu (class MenuUDC) is allowed to select from a list of available methods or to implement his own solvers. The Diffpack class Ptv(real) provides an efficient way to define 'point' objects.

The class FiniteElement provides the framework for all basic finite element calculations like evaluation of shape functions and their derivatives at Gauss points. The stiffness and load calculations and the assembly of the linear system of equations is performed using the class FEM. The particular problem-dependent calculations are defined in a simulator class derived from FEM. The integration of domain contributions to the stiffness matrix and load vector is executed through virtual function integrands, whereas the boundary contribution to the stiffness/load vector from boundary forcing terms (e.g. mixed boundary condition on a cooled side in a heat transfer...
problem) is provided through \textit{integrands4side}. In problems with selective integration (for example, use of reduced integration), different integration procedures can be specified through the virtual function \texttt{calcElmMatVec} which calculates the element matrix and vector contributions to the linear system solver. Finally, basic time management as required by time integration algorithms is performed using the class \texttt{TimePrm}.

In the next section, the mathematical problem constituting the \textit{inverse design natural convection problem} is defined. This problem is chosen as an example problem in the following discussion solely for the purpose of demonstrating the effectiveness of the object-oriented techniques developed in the present paper. It is the simplest continuum system that maintains the general form of the problems of interest as it is defined with the coupling of two PDEs, namely the Navier–Stokes equations and the convection/diffusion energy equation. The interested reader is advised to consult [20] for various numerical and mathematical issues dealing with this problem.

3. DIRECT AND INVERSE DESIGN NATURAL CONVECTION PROBLEMS

3.1. Problem definition

Let \( \Omega \) be a closed bounded region in \( \mathbb{R}^{n_{sd}} \), where \( n_{sd} \) is the number of space dimensions, with a piecewise smooth boundary \( \Gamma \) (Figure 1). An incompressible viscous fluid fills the domain and thermal gradients due to applied boundary fluxes induce natural convection in the fluid.

The equations governing the natural convection system are the incompressible Navier–Stokes equation and the energy equation. These equations are given below in a non-dimensional form. The key dimensionless quantities referred are the Prandtl number (\( Pr \)) and the Rayleigh number (\( Ra \)). They are defined as \( Pr = \frac{v}{\alpha} \) and \( Ra = g \beta (T_{in} - T_{ref}) L^3 / \nu \), respectively, where \( L \) denotes a characteristic length of the domain, \( \beta \) the thermal expansion coefficient, \( g \) the gravity constant, \( \alpha \) the thermal diffusivity, \( v \) the kinematic viscosity, \( T_{in} \) the initial temperature and \( T_{ref} \) a reference temperature.
The conservation of momentum for the velocity field $v(x,t)$ in $(x,t) \in \Omega \times [0,t_{\text{max}}]$ is given by

$$\frac{\partial v}{\partial t} + (\nabla v)v = \nabla \cdot \sigma - PrRa \theta e_g$$

(1)

where the constitutive equation defining the stress tensor is given as

$$\sigma = -pI + Pr[\nabla v + (\nabla v)^T]$$

(2)

The incompressibility condition takes the form

$$\nabla \cdot v(x,t) = 0, \quad (x,t) \in \Omega \times [0,t_{\text{max}}]$$

(3)

In the equations above, $I$ is the second-order unit tensor and $e_g$ is a unit vector in the direction of gravity. The variation of density is modelled with the Boussinesq approximation, i.e. (in a dimensional form)

$$\rho = \rho_0[1 - \beta(T - T_{\text{ref}})]$$

The temperature field in $\Omega \times [0,t_{\text{max}}]$ is governed by the following energy equation:

$$\frac{\partial \theta}{\partial t} + v \cdot \nabla \theta = \nabla \cdot \nabla \theta$$

(4)

The known initial conditions are the following:

$$v(x,0) = v_m(x), \quad x \in \Omega$$

(5)

and

$$\theta(x,0) = \theta_m(x), \quad x \in \Omega$$

(6)

The fluid velocity $v$ is assumed known at the boundary $\Gamma$. The no-slip condition is used here, i.e.

$$v(x,t) = 0, \quad (x,t) \in \Gamma \times [0,t_{\text{max}}]$$

(7)

On the part $\Gamma_h$ of the boundary $\Gamma$, a heat flux boundary condition is applied, whereas on the remaining part of the boundary $\Gamma_g$, a temperature boundary condition is considered, i.e. $\Gamma_h \cup \Gamma_g = \Gamma$ and $\Gamma_h \cap \Gamma_g = \emptyset$ (see Figure 1). However, the distribution of the boundary heat flux on $\Gamma_{h0} \subset \Gamma_h$ is not known ($\Gamma_{h0} \cup \Gamma_{h1} = \Gamma_h, \Gamma_{h0} \cap \Gamma_{h1} = \emptyset$).

The known flux and temperature distributions are given below:

$$\theta(x,t) = \theta_g(x,t), \quad (x,t) \in \Gamma_g \times [0,t_{\text{max}}]$$

(8)

$$q(x,t) = \nabla \theta(x,t) \cdot n = q_1(x,t), \quad (x,t) \in \Gamma_{h1} \times [0,t_{\text{max}}]$$

(9)

whereas the unknown flux distribution on $\Gamma_{h0}$ is

$$q(x,t) \equiv \nabla \theta(x,t) \cdot n = q_o(x,t), \quad (x,t) \in \Gamma_{h0} \times [0,t_{\text{max}}]$$

(10)

The objective of the inverse design problem is to calculate the unknown heat flux $q_o(x,t), (x,t) \in \Gamma_{h0} \times [0,t_{\text{max}}]$ given Equations (1)–(9) and some additional information about the temperature field. It is here assumed that the temperature field is a priori known in a subset $\Gamma_I$ of the boundary $\Gamma_{h1}$.
where the flux \( q_1(x,t) \equiv q_t(x,t) \) is also known, i.e.

\[
\theta(x,t) = \theta_m(x,t), \quad (x,t) \in \Gamma_I \times [0,t_{\text{max}}]
\]

(11)

where \( \theta_m(x,t) \) is known.

The over-specified thermal boundary condition on \( \Gamma_I \) (Equation (11)) together with Equations (1)–(9) define an ill-posed inverse problem that can be solved to calculate the unknown heat flux \( q_o \) on \( \Gamma_{b0} \). Once this flux is calculated, Equations (1)–(10) can be used to solve a direct natural convection problem for the calculation of the temperature and velocity for each \((x,t) \in \Omega \times [0,t_{\text{max}}]\).

With the flux \( q_o \) as an input parameter (i.e. for a given guess function \( q_o(x,t), (x,t) \in \Gamma_{b0} \times [0,t_{\text{max}}] \)) and without using the temperature \( \theta_m \) on the boundary \( \Gamma_I \), one can use Equations (1)–(10) to define a direct (boundary/initial value) problem with its solution denoted as \( \theta(x,t; q_o) \) and \( \mathbf{v}(x,t; q_o) \), where \((x,t) \in \Omega \times [0,t_{\text{max}}]\). The definition of this direct boundary value problem is summarized in Box I. Using the solution of this direct problem, an optimization problem is next defined that calculates \( q_o \) such that the computed solution \( \theta(x,t; q_o) \) of the direct problem on \( \Gamma_I \times [0,t_{\text{max}}] \) is as close as possible to the given data \( \theta_m \).

3.2. The optimization algorithm

The objective of the inverse problem defined above is to look for a quasi-solution \( \tilde{q}_o(x,t) \in L_2(\Gamma_{b0} \times [0,t_{\text{max}}]) \) such that

\[
\mathcal{J}(\tilde{q}_o) \leq \mathcal{J}(q_o), \quad \forall q_o \in L_2(\Gamma_{b0} \times [0,t_{\text{max}}])
\]

(12)

where

\[
\mathcal{J}(q_o) = \frac{1}{2} \| \theta(x,t; q_o) - \theta_m(x,t) \|^2_{L_2(\Gamma_I \times [0,t_{\text{max}}])}
\]

\[
= \frac{1}{2} \int_0^{t_{\text{max}}} \int_{\Gamma_I} [\theta(x,t; q_o) - \theta_m(x,t)]^2 \, d\Gamma \, dt
\]

(13)

In the above equation, \( \theta(x,t; q_o) \) is defined as the solution of the direct natural convection problem of Box I with the thermal boundary conditions given by Equations (8) and (9) and with the flux \( q_o(x,t) \) applied on \( \Gamma_{b0} \). Note that the temperature distribution \( \theta_m(x,t), (x,t) \in \Gamma_I \times [0,t_{\text{max}}] \) (Equation (11)) does not enter the definition of the direct problem given in Box I. Instead, it appears as a part of the cost functional in the above-stated optimization problem.

As is typical with all inverse problems, the main difficulty with the above optimization problem is the calculation of the gradient \( \mathcal{J}'(q_o(x,t)) \) of the cost functional in the \( L_2(\Gamma_{b0} \times [0,t_{\text{max}}]) \) space. Introducing the directional derivative \( D_{\Delta q_o} \mathcal{J}(q_o) \equiv (\mathcal{J}'(q_o), \Delta q_o)_{L_2(\Gamma_{b0} \times [0,t_{\text{max}}])} \) of \( \mathcal{J}(q_o) \) and using Equation (13), one can write the following:

\[
D_{\Delta q_o} \mathcal{J}(q_o) \equiv (\mathcal{J}'(q_o), \Delta q_o)_{L_2(\Gamma_{b0} \times [0,t_{\text{max}}])} = (\theta(x,t; q_o) - \theta_m(x,t), \Theta(x,t; q_o, \Delta q_o))_{L_2(\Gamma_I \times [0,t_{\text{max}}])}
\]

(14)
Box I. Direct problem to define $\theta(x, t; q_0)$ and $v(x, t; q_0)$.

\[
\frac{\partial \theta(x, t; q_0)}{\partial t} + v(x, t; q_0) \cdot \nabla \theta(x, t; q_0) = \nabla^2 \theta(x, t; q_0), \quad (x, t) \in \Omega \times [0, t_{\text{max}}] \tag{15}
\]

\[
\frac{\partial v(x, t; q_0)}{\partial t} + (\nabla v(x, t; q_0)) v(x, t; q_0) = \nabla \cdot \sigma(x, t; q_0) - RaPr \theta(x, t; q_0) e_g, \quad (x, t) \in \Omega \times [0, t_{\text{max}}] \tag{16}
\]

\[
\sigma(x, t; q_0) = -p(x, t; q_0) I + Pr [\nabla v(x, t; q_0) + (\nabla v(x, t; q_0))^T], \quad (x, t) \in \Omega \times [0, t_{\text{max}}] \tag{17}
\]

\[
\nabla \cdot v(x, t; q_0) = 0, \quad (x, t) \in \Omega \times [0, t_{\text{max}}] \tag{18}
\]

\[
\theta(x, 0; q_0) = \theta_m(x), \quad v(x, 0; q_0) = v_m(x), \quad x \in \Omega \tag{19}
\]

\[
v(x, t; q_0) = 0, \quad (x, t) \in \Gamma \times [0, t_{\text{max}}] \tag{20}
\]

\[
\theta(x, t; q_0) = \theta_g, \quad (x, t) \in \Gamma_g \times [0, t_{\text{max}}] \tag{21}
\]

\[
\frac{\partial \theta}{\partial n}(x, t; q_0) = q_1(x, t), \quad (x, t) \in \Gamma_{h1} \times [0, t_{\text{max}}] \tag{22}
\]

\[
\frac{\partial \theta}{\partial n}(x, t; q_0) = q_0(x, t), \quad (x, t) \in \Gamma_{h0} \times [0, t_{\text{max}}] \tag{23}
\]

where the sensitivity temperature field $\Theta(x, t; q_0, \Delta q_0) \equiv D_{\Delta q_0} \theta(x, t; q_0)$ and the sensitivity velocity field $V(x, t; q_0, \Delta q_0) \equiv D_{\Delta q_0} v(x, t; q_0)$ are defined as the linear in $\Delta q_0$ parts of $\theta(x, t; q_0 + \Delta q_0)$ and $v(x, t; q_0 + \Delta q_0)$, respectively, calculated at $q_0$, i.e.

\[
\theta(x, t; q_0 + \Delta q_0) = \theta(x, t; q_0) + \Theta(x, t; q_0, \Delta q_0) + \mathcal{O}(\|\Delta q_0\|^2_{L_2(\Gamma_{10} \times [0, t_{\text{max}}])}) \tag{24}
\]

\[
v(x, t; q_0 + \Delta q_0) = v(x, t; q_0) + V(x, t; q_0, \Delta q_0) + \mathcal{O}(\|\Delta q_0\|^2_{L_2(\Gamma_{10} \times [0, t_{\text{max}}])}) \tag{25}
\]

As it becomes clear from Equation (14), the calculation of the gradient $\mathcal{J}'(q_0)$ requires the evaluation of the adjoint to the sensitivity of the temperature operator. The definition of the thermal and fluid flow sensitivity problems that define $\Theta(x, t; q_0, \Delta q_0)$ and $V(x, t; q_0, \Delta q_0)$ is given in Box II. The corresponding adjoint problem that defines the adjoint fluid flow and thermal fields $\phi(x, t; q_0)$ and $\psi(x, t; q_0)$, respectively, is given in Box III. The definition of Box III is such that an analytical calculation of the gradient of $\mathcal{J}(q_0)$ is feasible. Indeed, with the definition of the adjoint problem

Copyright © 2000 John Wiley & Sons, Ltd.  

given in Box III and after some lengthy but rather straightforward calculations (see Reference [20]), the gradient of the cost functional \( J'(q_0) \) is obtained in terms of the adjoint temperature field as

\[
J'(q_0(x,t)) = \psi(x,t;q_0), \quad (x,t) \in (\Gamma_{\delta_0} \times [0,t_{\max}])
\] (26)

**Box II. Sensitivity problem to define \( \Theta(x,t;q_0,\Delta q_0) \) and \( V(x,t;q_0,\Delta q_0) \).**

\[
\frac{\partial \Theta(x,t;q_0,\Delta q_0)}{\partial t} + v(x,t;q_0) \cdot \nabla \Theta(x,t;q_0,\Delta q_0) + V(x,t;q_0,\Delta q_0) \cdot \nabla \Theta(x,t;q_0) \\
= \nabla^2 \Theta(x,t;q_0,\Delta q_0), \quad (x,t) \in \Omega \times [0,t_{\max}]
\] (27)

\[
\frac{\partial V(x,t;q_0,\Delta q_0)}{\partial t} + (\nabla V(x,t;q_0,\Delta q_0)) v(x,t;q_0) + (\nabla v(x,t;q_0)) V(x,t;q_0,\Delta q_0) \\
= \nabla \cdot \Sigma(x,t;q_0,\Delta q_0) - PrRa \Theta(x,t;q_0,\Delta q_0) e_g, \quad (x,t) \in \Omega \times [0,t_{\max}]
\] (28)

\[
\Sigma(x,t;q_0,\Delta q_0) = - P(I(x,t;q_0,\Delta q_0) I + Pr[\nabla V(x,t;q_0,\Delta q_0)]^T, \quad (x,t) \in \Omega \times [0,t_{\max}]
\] (29)

\[
\nabla \cdot V(x,t;q_0,\Delta q_0) = 0, \quad (x,t) \in \Omega \times [0,t_{\max}]
\] (30)

\[
\Theta(x,0;q_0,\Delta q_0) = 0, \quad V(x,0;q_0,\Delta q_0) = 0, \quad x \in \Omega
\] (31)

\[
V(x,t;q_0,\Delta q_0) = 0, \quad (x,t) \in \Gamma \times [0,t_{\max}]
\] (32)

\[
\Theta(x,t;q_0,\Delta q_0) = 0, \quad (x,t) \in \Gamma_\delta \times [0,t_{\max}]
\] (33)

\[
\frac{\partial \Theta}{\partial n}(x,t;q_0,\Delta q_0) = 0, \quad (x,t) \in \Gamma_{l1} \times [0,t_{\max}]
\] (34)

\[
\frac{\partial \Theta}{\partial n}(x,t;q_0,\Delta q_0) = \Delta q_0(x,t), \quad (x,t) \in \Gamma_{\delta_0} \times [0,t_{\max}]
\] (35)

The solution of the inverse design problem is obtained by solving the optimization problem defined earlier using any gradient-based method, e.g. the conjugate gradient method (CGM) [13, 35]. The CGM algorithm constructs a sequence \( q_0^0(x,t), q_0^1(x,t), \ldots, q_0^k(x,t), \ldots \), as follows:

\[
q_0^{k+1}(x,t) = q_0^k(x,t) + \alpha^k p^k(q_0^k(x,t)), \quad (x,t) \in \Gamma_{\delta_0} \times [0,t_{\max}]
\] (36)

where \( q_0^k(x,t) \) is the solution at iteration \( k \), \( \alpha^k \) is the step size and \( p^k \) is the conjugate search direction. The calculation of \( \alpha^k \) and \( p^k \) requires the solution of the sensitivity and adjoint problems,
respectively. Box IV briefly summarizes the computational algorithm. With a given tolerance, the solution \( q^k_o \) converges to the optimum solution \( \hat{q}_o \).

Box III. Adjoint problem to define \( \psi(x, t; q_o) \) and \( \phi(x, t; q_o) \).

\[
\frac{\partial \psi(x, t; q_o)}{\partial t} + \mathbf{v}(x, t; q_o) \cdot \nabla \psi(x, t; q_o) = -\nabla^2 \psi(x, t; q_o) + \phi(x, t; q_o) \cdot \mathbf{e}_{\gamma}, (x, t) \in \Omega \times [0, t_{\text{max}}] \tag{37}
\]

\[
\frac{\partial \phi(x, t; q_o)}{\partial t} + (\nabla \phi(x, t; q_o)) \mathbf{v}(x, t; q_o) - (\nabla \mathbf{v}(x, t; q_o))^T \phi(x, t; q_o) = -\nabla \cdot \zeta(x, t; q_o) + \text{Pr} \text{Ra} \psi(x, t; q_o) \nabla \theta(x, t; q_o), \quad (x, t) \in \Omega \times [0, t_{\text{max}}] \tag{38}
\]

\[
\zeta(x, t; q_o) = -\pi(x, t; q_o) \mathbf{I} + \text{Pr} [\nabla \phi(x, t; q_o)]^T, \quad (x, t) \in \Omega \times [0, t_{\text{max}}] \tag{39}
\]

\[
\nabla \cdot \phi(x, t; q_o) = 0, \quad (x, t) \in \Omega \times [0, t_{\text{max}}] \tag{40}
\]

\[
\psi(x, t_{\text{max}}; q_o) = 0, \quad \phi(x, t_{\text{max}}; q_o) = 0, \quad x \in \Omega \tag{41}
\]

\[
\phi(x, t; q_o) = 0, \quad (x, t) \in \Gamma_y \times [0, t_{\text{max}}] \tag{42}
\]

\[
\psi(x, t; q_o) = 0, \quad (x, t) \in \Gamma_y \times [0, t_{\text{max}}] \tag{43}
\]

\[
\frac{\partial \psi}{\partial n}(x, t; q_o) = 0, \quad (x, t) \in (\Gamma_h - \Gamma_f) \times [0, t_{\text{max}}] \tag{44}
\]

\[
\frac{\partial \psi}{\partial n}(x, t; q_o) = \theta(x, t; q_o) - \theta_m(x, t), \quad (x, t) \in \Gamma_f \times [0, t_{\text{max}}] \tag{45}
\]

4. OBJECT-ORIENTED DESIGN

The software design process is often the most difficult and time-consuming phase of object-oriented software development. In general, this process consists of identifying and grouping logical entities and removing uninteresting programming details from the public interface. This is the process of making abstractions. In the context of numerical computing, abstractions are provided readily by the underlying mathematical structure and it is the task of the intelligent programmer to exploit this feature. It is here demonstrated how such mathematical abstractions can be identified and incorporated in the design of classes for the inverse natural convection design problem considered.
in the previous section. However, the following discussion would tend to be general and ideas presented here would be applicable to most design and control problems implemented using the continuum adjoint formulation.

Box IV. The conjugate gradient optimization algorithm.

I: Make an initial guess \( q_0^k(x, t) \in L_2(\Gamma_{h0} \times [0, t_{\text{max}}]) \) and set \( k = 0 \)

II: Calculate the conjugate search direction \( p^k(x, t), (x, t) \in \Gamma_{h0} \times [0, t_{\text{max}}] \)

1. Solve the direct problem for \( \theta(x, t; q_0^k) \) and \( v(x, t; q_0^k) \)
2. Compute the residual \( \theta(x, t; q_0^k) - \theta_0(x, t) \) for \( (x, t) \in \Gamma_I \times [0, t_{\text{max}}] \)
3. Solve the adjoint problem backwards in time for \( \psi(x, t; q_0^k) \) and \( \Phi(x, t; q_0^k) \)
4. Set \( \mathcal{J}'(q_0^k) = \psi(x, t; q_0^k) \) for \( (x, t) \in \Gamma_{h0} \times [0, t_{\text{max}}] \)
5. Set \( \gamma^k = 0 \), if \( k = 0 \); otherwise:

\[
\gamma^k = \frac{\| \mathcal{J}'(q_0^k) \|_{L_2(\Gamma_{h0} \times [0, t_{\text{max}}])}^2}{\| \mathcal{J}'(q_0^{k-1}) \|_{L_2(\Gamma_{h0} \times [0, t_{\text{max}}])}^2}
\]

Polak–Ribiere formula

\[
\gamma^k = \frac{\mathcal{J}'(q_0^k) - \mathcal{J}'(q_0^{k-1})}{\mathcal{J}'(q_0^{k-1})}
\]

Fletcher–Reeves formula

\[
\gamma^k = \frac{\mathcal{J}'(q_0^k) - \mathcal{J}'(q_0^{k-1})}{\mathcal{J}'(q_0^{k-1})}
\]

Hestenes–Stiefel formula

6. Define \( p^k(x, t) \): If \( k = 0 \), \( p^0 = -\mathcal{J}'(q_0^k) \); Otherwise, \( p^k = -\mathcal{J}'(q_0^k)(x, t) + \gamma^k p^{k-1} \)

III: Calculate the optimal step size \( \alpha^k \)

1. Solve the sensitivity problem for \( \Theta(x, t; q_0^k, p^k) \) and \( V(x, t; q_0^k, p^k) \)
2. Calculate \( \alpha^k \) by

\[
\alpha^k = \frac{-(\mathcal{J}'(q_0^k), p^k)_{L_2(\Gamma_{h0} \times [0, t_{\text{max}}])}}{\| \Theta(x, t; q_0^k, p^k) \|_{L_2(\Gamma_I \times [0, t_{\text{max}}])}^2}
\]

IV: Update \( q_0^{k+1}(x, t) = q_0^k(x, t) + \alpha^k p^k(x, t), (x, t) \in \Gamma_{h0} \times [0, t_{\text{max}}] \)

V: If \( \mathcal{J}'(q_0^{k+1}) < \varepsilon \) (specified tolerance), stop;
Otherwise, set \( k = k + 1 \) and go to Step II

4.1. Introduction to a generic class for convection–diffusion problems

The solution of the inverse natural convection design problem requires the solution (at each CGM iteration) of three sub-problems, i.e. of the direct, sensitivity and adjoint problems shown in Boxes I, II and III, respectively. The first task in the software design process is to model these three individual sub-problems. Considering the mathematical structure of the three sub-problems, one can notice that each sub-problem involves a system of two PDEs as well as corresponding boundary and initial data.

One possible design for the considered direct, adjoint and sensitivity problems (i.e. for the collection of the six PDEs) would be to declare six different classes, each one managing its own
fields, grids, linear system solvers and the whole non-linear solution process. In particular, one could define the classes \texttt{NlHeat} and \texttt{NavierStokes} to manage the heat and flow subproblems in the direct problem of Box I, respectively. Likewise, classes \texttt{AdjointHeat}, \texttt{AdjointFlow}, \texttt{SensitivityHeat} and \texttt{SensitivityFlow} can be defined to manage the heat and flow subproblems in the adjoint and sensitivity problems, respectively. Each of the above classes would contain the corresponding fields (for example, the flow field \( v \) defined in class \texttt{NavierStokes} and the temperature field \( \theta \) defined in the class \texttt{NlHeat}), each defined over a finite element grid. A linear equation system and solver needs to be defined in each of the above classes for the solution of the corresponding (linearized) fields (assuming, for example, that a linearized problem has been set up in a Newton–Raphson or other type of iterative process).

This approach definitely looks more structured than a procedural implementation which typically involves division into subroutines and functions. However, from a closer perspective, one can notice that such a strategy does not take into account the specific nature of the six PDEs and suffers from not recognizing the similarities in the problems involved and would thus involve repeated code. Since common features are not factored and stored in one place, code maintenance would also prove to be a difficult task. For example, in order to incorporate a change in the solution process, one would need to make the corresponding changes in six different places instead of one. If this functionality is factored and stored in one place then one would just require to derive a sub-class from the default version to incorporate the change. In a system of much greater magnitude (for example, in a system where each of the direct, sensitivity and adjoint problems is defined with more than two PDEs), factoring out commonalities would greatly ease the software process and reduce the occurrence of errors.

Recognizing the nature of the PDEs involved, one can notice that the governing equations for the direct, adjoint and sensitivity flow and heat problems (see Boxes I, II and III, respectively) are similar in nature and that they fall under the general category of convection-diffusion equations. Also, the PDEs corresponding to the direct problem are nonlinear and a standard predictor-multicorrector method can, for example, be used for their solution process [36, 37]. Thus, a general abstract base class \texttt{ConvectionDiffusion} is defined here to manage the solution process for the model convection–diffusion equation by using a default predictor multi-corrector scheme (see Box V). This class basically manages the solution process of the following discrete non-linear system:

\[
[M]\{\dot{x}\} + [C]\{x\} = \{F\}
\]  

which is the standard form to which all six PDEs are reduced (in a weak Petrov–Galerkin sense) before the time integration takes place. As shown in Box V, the class \texttt{ConvectionDiffusion} contains a generic grid object, a degree-of-freedom object to manage the mapping between nodal field values and the degrees of freedom in the linear system, a time integration parameter object to manage the time stepping of the model equation (Equation (46)) and the generic finite element fields (representing the continuous field variables) as needed for the time integration. In addition, it provides functionalities to go through time stepping, to solve for the primary unknowns at each time step, to update fields, etc. At each time step, the solution of the primary finite element field requires the solution of a linear system. The linear system to be assembled is specific to the PDE being solved and hence the virtual functions \texttt{integrands4side} and \texttt{integrands} are redefined in classes derived from \texttt{ConvectionDiffusion} for each particular PDE. Initial and boundary conditions for a PDE are problem dependent and hence the functions \texttt{setIC} and \texttt{essBC} which define such conditions, are declared as pure virtual and are provided in a problem dependent class. Further details on this follow in a later stage of this paper.
The virtual function adm() administers the whole menu system which reads the problem parameters. The virtual function define() builds the menu system by defining the various parameters, whereas the function scan() reads in the actual values of the problem parameters from an input file and assigns them to the corresponding variables.

The present construction of a generic class for convection–diffusion problems easily accommodates changes in the solution methodology and the time stepping procedures. For example, an implementation of a different variational formulation for the flow equations, such as a SUPG/PSPG stabilized method [38, 39] which allows for equal-order-interpolation velocity-pressure elements or a deforming-spatial-domain/space-time method [40, 41] to accommodate moving boundaries and interfaces, can easily be introduced in the present construction by deriving a sub-class of the class ConvectionDiffusion and suitably redefining the virtual functions integrands() and integrands4side(). Also, in the event that the user desires to apply time-integration techniques other than the default predictor-multicorrector scheme, such as a multistep method (see for example References [38] or [42]) or a theta scheme [43], one needs to derive a sub-class and redefine the function solveAtThisTimeLevel().

Remark. Another alternative approach to the design of the model problem solver, Convection-Diffusion, could be to just let this class manage the generic fields, grids, etc. and in addition provide a pointer to a generic time integrator which can be modeled as an abstract class TimeIntegrator that manages the time integration process at each time step. This class can be made
independent of any particular time-integration algorithm. A number of algorithm specific classes such as PredictorCorrector, MultiStep, ThetaScheme, etc., can be derived from this base class as shown in Figure 2. The base class pointer can then be bound to derived class objects during run time through instructions received through a menu.

Having factored out the common features in an abstract class, the next task is to incorporate the specific differences which distinguish one PDE (and associated initial/boundary data) from another, into the software. To incorporate these specific differences, six classes one for each PDE are declared as explained before. These classes deal with only the specific differences and obtain the common functionalities through the base class ConvectionDiffusion from which all the above six classes are derived. Box VI shows a few important members of the class NlHeat and how only specific differences are encoded in this class. All the other five PDE classes have been developed on the same lines. The class DomainBoundaryConnectivity from which the class NlHeat is derived is a class that is basically defined to obtain the connectivity information between the domain nodes and the nodes on the boundary (similarly the classes SensitivityHeat and AdjointHeat are derived from the class DomainBoundaryConnectivity).

As is evident from Boxes V and VI, the class NlHeat contains very few members in contrast to the class ConvectionDiffusion. Due to the very similar nature of the various PDEs involved, all the common features are accumulated in the abstract base class and the specific PDE class contains only the differences from the model problem. Other than the fields and members specific to the NlHeat class, this class also provides the routines integrands and integrands4side which were declared as virtual in the base class ConvectionDiffusion. The specific integrands and integrands4side depend on the particular weak form of the PDE which in turn depends on the specific terms of the PDE as well as on the specific nature of the boundary conditions. Other member functions in NlHeat include overriding versions of define and scan which read in related data from the menu.

The pointer heat_flux is for providing the heat flux boundary condition \( q_o \) on \( \Gamma_{b0} \times [0,t_{\text{max}}] \). Unlike other boundary conditions, the heat flux boundary condition is not specific to the NlHeat class but is also a part of the Optimization class to be defined below. Indeed, this flux condition is used here as a parameter to the direct problem in the sense presented in Box I and it is the same as that calculated during each CGM iteration (see Box IV).

The class NlHeat also defines the pure virtual function neumann() to provide for neumann type of boundary conditions (i.e. heat flux conditions). The neumann() function provides the boundary heat flux on all parts of the boundary of the liquid domain including the boundary \( \Gamma_{b0} \). In fact, this function refers to the pointer heat_flux for providing the flux contribution on the boundary \( \Gamma_{b0} \). It is declared as pure virtual in the class NlHeat (see also Section 4.2) to allow flexibility in accommodating boundary heat flux distributions of any kind (e.g. varying over space and time). The input parameters side and local_node in the function neumann() are provided to specify the exact location of a particular node in the finite element where the boundary flux contributions are to be calculated. The flux contributions on the boundaries \( \Gamma_{b0} \) and \( \Gamma_{b1} \) of the heat flux boundary \( \Gamma_b \) are distinguished by assigning proper boundary indicators to the various boundaries. The particular distribution of the heat flux \( q_1(x,t),(x,t) \in \Gamma_{b1} \times [0,t_{\text{max}}] \) must be provided by the user as a protected function in an application class derived from NlHeat.

It is noted that the function neumann() is not declared in the class ConvectionDiffusion since this class is used to simulate not only scalar convection-diffusion problems such as heat flow problems but also vector equations, as for example, the Navier–Stokes equations. In almost all problems dealing with the Navier–Stokes equations, only essential boundary conditions (no-slip
Box V. The main members of the class ConvectionDiffusion.

class ConvectionDiffusion: public HandleId, public FEM,
   public MenuUDC, public Store4Plotting
{
protected:
   Handle(GridFE) grid; // finite element grid
   Handle(DegFreeFE) dof; // mapping nodal values \rightarrow linear system
   Handle(TimePrm) tip; // time loop parameters
   Handle(FieldsFE) alpha; // FE field, the primary unknown
   Handle(FieldsFE) alpha_prev; // solution at previous time step
   Handle(FieldsFE) alpha_dot; // time rate field of the primary unknown
   Handle(FieldsFE) alpha_dot_prev; // rate field at previous time step
   real gamma; // time integration parameter
   real error_tolerance; // error tolerance for the nonlinear solver
..
   Vec(real) linear_solution; // solution of linear subsystem
   Handle(LinEqAdm) lineq; // linear system and solution
..
   virtual void timeLoop();
   virtual void solveAtThisTimeLevel();
   virtual void fillEssBC();
   virtual void integrands(ElmMatVec& elmat, FiniteElement& fe);
   virtual void calcElmMatVec(int elm_no, ElmMatVec& elmat,
                              FiniteElement& fe);
   virtual void integrands4side(int side, int boind, ElmMatVec& elmat,
                                FiniteElement& fe);
   virtual real essBC(int node_number) = 0; // essential boundary conditions
   virtual void setIC() = 0; // initial conditions
..
public:
   virtual void define(MenuSystem& menu, int level = MAIN);
   virtual void scan(MenuSystem& menu);
   virtual void adm(MenuSystem& menu);
..
   virtual void solveProblem(); // solves the complete problem
   virtual void storeResults()=0; // stores problem dependent fields
   virtual void updateDataStructures(); // updates the fields from \( t_n \) to \( t_{n+1} \)
..};

conditions) are provided on the boundary and using neumann() as a pure virtual function in the class ConvectionDiffusion would unnecessarily complicate the simulator as this would require a dummy declaration in every flow solver.

Box VI. The main members of the class NiHeat.

```cpp
class NiHeat: public ConvectionDiffusion, public DomainBoundaryConnectivity {
    friend class Direct;
    friend class Optimization;

    protected:
        Handle(FieldsFE) u;  // known fluid velocity
        Handle(Mat(real)) heat_flux;  // heat flux \eta on \Gamma_{80} \times [0, t_{max}]
        UpwindFE PG;  // Petrov-Galerkin weights

        virtual void integrands(ElmMatVec& elmat, FiniteElement& fe);
        virtual void integrands4side(int side, int boind, ElmMatVec& elmat, FiniteElement& fe);
        virtual real neumann(FiniteElement& fe, int side, int local_node) = 0;  // boundary heat flux

    public:
        virtual void define(MenuSystem& menu, int level = MAIN);
        virtual void scan(MenuSystem& menu);
        virtual void storeResults();

    };
```

Finally, the UpwindFE object is used to define the Petrov-Galerkin weights. Here, the UpwindFE object provides for a classical SUPG weighting function as proposed by Brooks and Hughes [36]. Any other suitable weighting functions can also be added to the hierarchy, such as for example SUPG/PSPG weighting functions, as proposed by Tezduyar et al. [38]. Since the definition of these weights requires knowledge of the velocity field \( u \), the UpwindFE object is only introduced in classes derived from ConvectionDiffusion (such as the class NiHeat in Box VI) where the velocity field \( u \) is defined.

4.2. Handling of problem-dependent functionalities

In almost all PDE applications, there are certain parameters and functions which vary from problem to problem. These problem-specific changes always present themselves as an obstacle in an object-oriented numerical software development. The problem is intensified in large-scale FEM codes as a lot of effort goes into their design and implementation and the developers would prefer to have the code general enough to accommodate at least a range of related problems. In the following discussion, some guidelines are provided to eliminate problem-specific functionalities from the design of a general simulator.

Typically classes like NiHeat, NavierStokes, etc., which solve a PDE-based boundary/initial value problem need to accommodate various boundary and initial data. Each application problem would have its own specific differences and hence to solve the problem these differences should
be incorporated directly in the code or passed on with an input file or command line arguments. In large-scale problems, it is rare that one could pass all the specific problem details through an input file or command line arguments. Some details, as for example specific thermal boundary conditions, need to be a part of the code. This problem can be solved here by an elegant use of pure virtual functions. In the present work, boundary and initial data are declared as pure virtual functions in the main PDE class (see e.g. \texttt{neumann()} in \texttt{NlHeat}, Box VI) and the specific boundary conditions are passed on to the simulator through a problem-dependent class derived from the base class. For example, a class \texttt{NlHeatDer} can be derived from \texttt{NlHeat} and that could provide for the specific problem details particular to an application. The OOP concept of \textit{late or dynamic binding} also allows for more than one application to share the same generic code as shown in Figure 3. Depending on the contents of the base class pointer \texttt{heat} (shown inside the ellipse in Figure 3), the corresponding derived class version of the pure virtual function \texttt{neumann} is executed during runtime. In fact, this flexibility allows the user to combine any application-specific boundary conditions to any variational algorithm or time-stepping procedure available in the hierarchy. Moreover, this construction allows for a logical organization of the various application-specific files in individual application directories which can be linked to the generic classes stored in the main simulator directory.

4.3. Modelling of the direct, adjoint and sensitivity sub-problems

Having completed the development of various PDE classes, the next task in the development of the inverse design simulator is to combine the individual simulators to form the direct, adjoint and sensitivity problems. In the present work, three problem classes \texttt{Direct}, \texttt{Adjoint} and \texttt{Sensitivity} are defined to manage the corresponding problems. Box VII shows the main members of the class \texttt{Direct}. The other two classes are similar in construction.

As can be seen from Box VII, each of the sub-problem classes have pointers to the corresponding heat and flow pointers. These classes mainly provide the time-stepping routines and some additional routines such as storing of fields.

It can be noted from Boxes I–III that the involved PDEs are not stand-alone but are coupled to one another. For example, the heat transfer problem is affected by the velocity field and similarly the fluid flow problem is affected by the temperature field. Thus, the various fields involved need to be intercoupled for a realistic implementation. The intercoupling of fields is not just restricted to the direct problem. There are plenty of such couplings in both the sensitivity and adjoint problems.

An elegant way to achieve such a coupling is by binding the pointers to the various fields involved. The set of program command lines shown below demonstrates how this coupling is achieved for the case considered above:

\begin{verbatim}
heat → v. rebind (flow → v())
flow → T. rebind (heat → T())
\end{verbatim}

where \texttt{heat} and \texttt{flow} are pointers to \texttt{NlHeat} and \texttt{NavierStokes} classes, respectively. Also, since the time integration of the direct problem involves solving the heat and flow problem simultaneously and the same being the case for the adjoint and sensitivity problems, the ‘time integration objects’ of the pointers to the subclasses \texttt{NlHeat} and \texttt{NavierStokes} need to be bound so that there exists only one ‘time integration object’ (e.g. the same time step for all problems,
Box VII. The main members of the class `Direct`.

```cpp
class Direct: public HandleId
{
    friend class Optimization;

protected:
    Handle(NilHeat) heat;  // pointer to heat transport problem
    Handle(NavierStokes) flow;  // pointer to flow problem
    Handle(TimePrm) tip;  // time integration parameters
    Handle(Mat(real)) residual;  // residual temperature \( \theta - \theta_m \) on \( \Gamma_f \times [0,t_{max}] \)

    virtual real prescribedTemperature(const Ptv(real) x, real time) = 0;  // \( \theta_m \) on \( \Gamma_f \times [0,t_{max}] \)

public:
    virtual void define(MenuSystem& menu, int level = MAIN);
    virtual void scan(MenuSystem& menu);
    virtual void adm(MenuSystem& menu);
    virtual void solveProblem(int iteration);  // solves the direct problem at a given CGM iteration
    virtual void timeLoop();  // executes the time integration of the heat and flow equations
    virtual void storeResults(Boolean append);  // stores the thermal and flow fields at all times
    virtual void updateDataStructures();  // updates the fields at the end of the time step
    virtual void solveAtThisTimeLevel();  // solves the heat and flow equations at a given time step
    virtual void storeResidual(int iteration);  // stores the residual at the end of the direct problem solution

};
```

etc.). This process is shown in the command lines below for a simple case:

```
heat → tip.rebind (tip())
flow → tip.rebind (tip())
```

where `tip` refers to the object that manages the time integration (initialization of time loop, time-stepping process, etc.).
4.4. Main simulator class

The main optimization routine is implemented as a class Optimization. Box VIII shows a few important members of the class Optimization. This class has pointers to the classes Direct, Adjoint and Sensitivity. Other than these members, the class Optimization contains the heat flux solutions at the current and previous iterations. Other intermediate vectors required in the conjugate gradient method are also defined as members in the class. This class also provides for three alternate formulas for the calculation of the search vector. The particular method to be used for an application is decided by the user and passed on through the input. In addition to generic menu administration and storing routines, this class provides the complete conjugate gradient algorithm, inner product routines for calculation of various inner products in the CGM algorithm, calculation of direction vector based on the method and finally evaluation of the cost functional at each iteration.

In concluding this section, the use of virtual functions and inheritance in codes is strongly advocated in problems dealing with solution of systems of PDEs due to the various reasons presented above. Deriving a subclass instead of extending an existing class has the advantage that the old code is still intact for purposes of testing. We believe that the ideas of OOP in this section may play an important role in implementation and verification procedures for simulators solving systems of PDEs and in particular in design simulators based on the adjoint method as applied to continuum systems.

5. NUMERICAL RESULTS AND COMPUTATIONS

Two examples are examined in this section. The first numerical example is a two-dimensional inverse natural convection problem in a square cavity. This problem was addressed earlier in Reference [20] and was chosen here as a test case. CPU comparisons with published data show that an object-oriented based implementation approach to inverse design problems provides efficiency that is comparable to that observed when using procedural language approaches (such as a C or a FORTRAN based implementation). In the second example, a three-dimensional extension of the problem of Example 1 is considered. Almost all realistic simulations typically require a detailed three-dimensional calculation if the ‘edge effects’ are to be captured effectively. Due to the complexity of the FEM implementation of PDE-based problems most procedural codes tend to be dimension dependent. It is demonstrated through the present simple 3-D example that an object-oriented approach does not require much effort to maintain dimension independent codes. The authors believe that to the best of their knowledge this is the first time that a numerical implementation is reported in the literature for a 3-D inverse natural convection design problem.

5.1. Example 1

The example considered here is taken from Reference [20] where it was originally defined. A fluid with Prandtl number unity and $Ra = 10^4$ is contained in a cavity at a homogeneous temperature $\theta_i = 1$ and the top, bottom and right side walls of the cavity are kept insulated, whereas the left side wall is maintained at a constant temperature $\theta_m = 0$ at $t > 0$ (see Figure 5). The solution of
Box VIII. The main members of the class Optimization.

```cpp
#include <iostream>
#include <string>

class Optimization: public HandleId
{
  protected:
    Handle(Direct) direct; // pointer to direct problem
    Handle(Adjoint) adjoint; // pointer to adjoint problem
    Handle(Sensitivity) sensitivity; // pointer to sensitivity problem
    Mat(real) heatFlux; // primary unknown: design flux
    Mat(real) heatFlux_prev; // known flux at previous iteration
    Handle(Mat(real)) gradient_Qo; // ∇J(q^k)
    Mat(real) gradient_Qo_prev; // ∇J(q^{k-1})
    Handle(Mat(real)) directionVec; // p^k(x,t),(x.t) ∈ Γj0 × [0,t_{max}]
    Mat(real) directionVec_prev; // p^{k-1}(x,t),(x,t) ∈ Γj0 × [0,t_{max}]
    Handle(Mat(real)) residual; // θ − θ_m on Γ_f × [0,t_{max}]
    Handle(Mat(real)) sens_T_gammas_I; // Θ on Γ_f × [0,t_{max}]
    String method; // selection of CG update formula
  
  virtual void conjugateGradient();
  virtual real costFunctional();
  virtual void calculateDirectionVec(int iteration);
  virtual real innerProduct(Mat(real) & f, Mat(real) & g, int n);
  ....
  
  public:
    virtual void define(MenuSystem & menu, int level = MAIN);
    virtual void scan(MenuSystem & menu);
    virtual void adm(MenuSystem & menu);
    virtual void driver(); // performs CGM algorithm
    virtual void storeResults(int iteration); // stores variables on file
    virtual void updateDataStructures();
    ....
};
```

the direct natural convection problem corresponding to the above data with a uniform 20 × 20
4-noded element mesh and Δt = 0.0025 results in the heat flux q_I(y,t) (see Figure 6) on the left
vertical wall. The SUPG formulation as proposed by Brooks and Hughes [36] is implemented here
for the solution of the flow equations.

The inverse problem is then defined as follows: with the bottom and top cavity walls insulated
and a known heat flux history q_I(y,t), t ∈ [0,1] applied at x = 0, find the heat flux history, q_o(y,t),
at the right-side wall, in order to maintain a temperature θ(y,t) ≡ 0, t ∈ [0,1], at x = 0 and y ∈ [0,1]. The flux q_I on the left-side wall is graphically represented in Figure 6, for t ∈ [0,1] and
y ∈ [0,1]. As such, both temperature and flux are prescribed on the left vertical wall, whereas no
thermal boundary conditions are available at x = 1. The solution to this problem that corresponds
Figure 4. A sketch of the inverse design simulator (optimization), the base classes and some layers of the internal objects. A solid line indicates class derivation (‘is-a’ relationship), whereas dashed lines represent a pointer/handle (‘has-a’ relationship).

to the flux $q_t$ and temperature $\theta(y,t) = 0$ at $x = 0$, is obviously $\tilde{q}_o(y,t) = 0$. More details can be obtained in the original reference.

$20 \times 20$ bi-linear uniform rectangular elements are used in the analysis of the direct, sensitivity and adjoint problems, with the time step $\Delta t = 0.0025$ for all of them. The maximum time is taken to be $t_{\max} = 1$. The total number of time steps for the solution of each of the direct, sensitivity and adjoint problems in one global CGM iteration is 400.

The initial guess $q_0^o$ was prescribed as $q_0^o(y,t) = 1 - t$. For this initial guess and a tolerance of $1.0e-07$, the CGM resulted in convergence in 8 iterations. The final optimum flux is shown in Figure 7 along with $q_k^o$ at some intermediate iterations. The variation of the cost functional and of the norm of the cost functional gradient at some iterations are tabulated in Table I. Further details on the numerical issues are omitted and the interested reader is advised to consult Reference [20] where a complete numerical study of this problem has been conducted. Next, a comparison is given between the computational times of the present implementation and that presented in Reference [20].

The present implementation was performed on an IBM RS-6000 (SP2) node at the Cornell Theory Center. The implementation was a serial implementation. The results reported in Reference
5.1. Example 1

The example considered here is the inverse natural convection example. A schematic of the problem is shown in Figure 5. The given heat flux history is applied on the left vertical wall of the cavity (Example 1). The goal is to determine the unknown boundary conditions.

![Figure 5. Schematic of the inverse natural convection example.](image)

![Figure 6. Given heat flux history $q_I(y,t)$ on the left vertical wall of the cavity (Example 1).](image)

### Table I. Variation of the cost functional and the norm of the gradient through the iterations (Example 1).

<table>
<thead>
<tr>
<th>Iter</th>
<th>$J(q_o)$</th>
<th>$|J'(q_o)|$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.96e−02</td>
<td>1.74e−01</td>
</tr>
<tr>
<td>2</td>
<td>3.42e−05</td>
<td>2.04e−03</td>
</tr>
<tr>
<td>3</td>
<td>8.86e−06</td>
<td>2.32e−03</td>
</tr>
<tr>
<td>5</td>
<td>4.07e−07</td>
<td>4.03e−04</td>
</tr>
<tr>
<td>7</td>
<td>1.34e−07</td>
<td>6.40e−05</td>
</tr>
<tr>
<td>8</td>
<td>6.97e−08</td>
<td>9.76e−05</td>
</tr>
</tbody>
</table>

[20] were obtained on a DEC-alpha 200 workstation with a double precision arithmetic. The computational cost of each CGM iteration (which includes the direct, adjoint and sensitivity problems, each involving 400 time steps) is shown in Table II, where the numbers are in CPU seconds in the corresponding machines. The comparison of the results is contrary to the general belief that object-oriented platforms like C++ are less efficient than procedural ones like C. The reason for this are two folds. First, the comparisons show the CPU times on two different machines and hence the comparisons are only approximate. Moreover, the C++ code was highly optimized and the computationally taxing numerics were coded in low-level (minimizing the number of flops) and hence enhancing the computing efficiency. This was not an issue in the earlier C-based implementation. In conclusion, an optimized OOP approach to inverse design problems can lead to codes that are as efficient as codes generated with procedural programming languages.

5.2. Example 2

The example considered here is the three-dimensional extension of the earlier example. It is of interest to this paper to demonstrate the ability of the present OOP-based code to handle both...
Figure 7. Initial guess $q^0_0 = 1 - t$ and the calculated flux $q_0(y,t)$ at iterations 1, 3, and 8 (optimal).

Table II. CPU time statistics for one CGM iteration (Example 1).

<table>
<thead>
<tr>
<th></th>
<th>Direct problem</th>
<th>Adjoint problem</th>
<th>Sensitivity problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zabaras and Yang [20]</td>
<td>1306.6</td>
<td>1512.4</td>
<td>1478.1</td>
</tr>
<tr>
<td>Present</td>
<td>780.0</td>
<td>1320.0</td>
<td>1320.0</td>
</tr>
</tbody>
</table>
2-D and 3-D problems. This was not possible in the earlier C-based implementation given in Reference [20]. For this reason and due to the significant computational cost associated with such three-dimensional inverse simulations, only a moderate FE mesh of $10 \times 10 \times 10$ brick elements is used in the calculations. Moreover, a modified version of the flow solver incorporating a SUPG/PSPG stabilized method [38] is used for the solution of the direct, adjoint and sensitivity flow equations to reduce the computational cost. The above-mentioned change in the flow solver is easily accommodated into the current framework as explained earlier in Section 4.1. The remaining problem details are exactly as in the first example. Figure 8 shows a schematic of the problem domain and boundary fluxes.

The three-dimensional inverse design problem is here defined exactly as in the first example problem, i.e.: with all walls other than the left and right walls insulated and a known temperature $\theta(y,z,t) = 0$ and heat flux history $q_I(y,z,t)$ applied at $x = 0$, find the heat flux $q_0(y,z,t)$ applied at $x = 1$ for $t \in [0,1.0]$. The heat flux $q_I$ (not shown here due to the complex three-dimensional variation) is numerically determined by solving the direct problem with $\theta(y,z,t) = 0$ at $x = 0$ and insulated flux conditions at $x = 1$. The objective of the above defined inverse problem is obviously to recapture the insulated conditions ($q_0(y,z,t) = 0$) on the right boundary ($x = 1$). The initial guess heat flux is taken to be $q_0(y,z,t) = 1 - t$ as in the earlier example. The initial flux, the optimal flux and flux distributions at two intermediate iterations are shown in Figures 9 and 10. Note that, Figure 9 shows the flux distribution along the vertical centreline on the side $\Gamma_{h0}$ (i.e. $y = 0.5$), whereas Figure 10 shows the flux distribution along the horizontal centreline on the boundary $\Gamma_{l0}$ (i.e. $z = 0.5$). The calculated cost functional values at intermediate iterations are tabulated in Table III. The CGM algorithm is terminated at 13 iterations, which corresponds to a tolerance of $1.0e^{-07}$. The computational cost for the direct, adjoint and sensitivity subproblems in one CGM iteration are 6884.41, 7444.99 and 7638.92 CPU seconds, respectively. Considering the moderate discretization, the obtained optimal solution is an excellent approximation of the insulated boundary conditions. For a more accurate solution a much finer grid than the one used here is required.
Figure 9. Initial guess heat flux $q_0(0.5,z,t) = 1 - t$ and the calculated flux $q_k(0.5,z,t)$ at iterations 1, 7 and 13 (optimal).

Table III. Variation of the cost functional and the norm of the gradient through the iterations (Example 2).

<table>
<thead>
<tr>
<th></th>
<th>Iter 1</th>
<th>Iter 2</th>
<th>Iter 3</th>
<th>Iter 5</th>
<th>Iter 9</th>
<th>Iter 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J(q_o)$</td>
<td>5.99e−02</td>
<td>3.21e−05</td>
<td>1.19e−05</td>
<td>1.49e−06</td>
<td>2.78e−07</td>
<td>9.29e−08</td>
</tr>
<tr>
<td>$|J'(q_o)|$</td>
<td>2.11e−01</td>
<td>3.12e−03</td>
<td>2.22e−03</td>
<td>2.09e−04</td>
<td>5.01e−05</td>
<td>2.63e−05</td>
</tr>
</tbody>
</table>
6. CONCLUSIONS

Some generic ideas for an efficient object-oriented software development for the inverse design of continuum systems governed by PDEs were addressed. Emphasis was given to design techniques based on the continuum adjoint method. For a lucid presentation of the ideas, the class of inverse natural convection design problems was adopted. The logical development of various objects and classes to represent and manage the inverse design problem were addressed step by step, which included design of individual PDE classes, handling of problem specific details and intercoupling of fields. The design process was elaborated in a sequential manner with emphasis given to the
use of inheritance, virtual functions and abstract classes. Attention was given to the development of a prototype simulator which avoids repetitive codes.

Two numerical examples were considered for validation and testing. The first example examined a 2-D inverse design natural convection problem in a square cavity. This example was considered here to illustrate the ideas, to show the obtained accuracy and also to compare computing times with a non-optimized C-based implementation of the same physical problem. The second example considered a 3-D extension of the first problem and demonstrated the effectiveness of using an object-oriented approach in designing dimension-independent codes. CPU time comparisons showed that a well-designed C++ code can out beat the numerical efficiency of a non-optimized C code. In general, the performance of a well-designed C code is better than that of an OOP based implementation. However, the benefits of the increased flexibility and robustness gained by the use of an object-based approach far surpass the drawback of the required higher CPU time.

ACKNOWLEDGEMENTS

This work was partially supported by a grant from Alcoa Laboratories. The computing support and facilities provided by the Cornell Theory Center are acknowledged. The academic license for using the various libraries of Diffpack is appreciated.

REFERENCES


