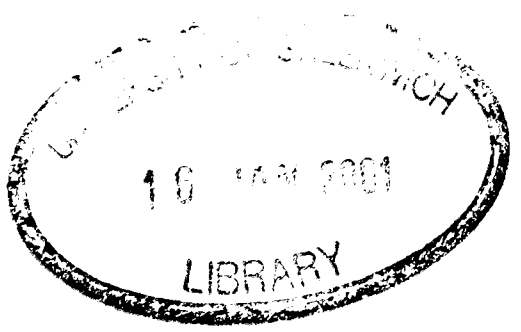
An investigation into the feasibility, problems and benefits of re-engineering a legacy procedural CFD code into an event driven, Object Oriented system that allows dynamic user interaction

John Andrew Clark Ewer

A thesis submitted in partial fulfilment of the requirements of the University of Greenwich for the degree of Doctor of Philosophy



July 2000

The University of Greenwich,
School of Computing and Mathematical
Science, Park Row, Greenwich, SE10 9LS.

Acknowledgements

I wish to dedicate this thesis to my parents who have helped me in more ways than I could possibly mention during my many years in academia. Their support and encouragement have been vital to my studies - The holidays were good too.

I would like to express my deepest gratitude to everyone involved in my Ph.D. research: Professor Brian Knight, for his continued enthusiasm, support and patience with regards to my research. I am also indebted to him for his advice and constructive criticism about the technical matters of writing this thesis. My other supervisor, Doctor Don Cowell has also helped me during my studies. Doctor Mayur Patel's enthusiasm, expertise and insights have helped to make "SMARTFIRE" into a significant research tool. Professor Ed Galea has likewise supported the advances that have been made during this research and always promoted the work. My former colleague, Doctor Steve Taylor has provided support from his complementary research into Case Based Reasoning for automated mesh generation. His comments and criticisms have helped to make this work more robust than would otherwise have been the case. I would also like to thank Doctor Miltos Petridis for his prior Ph.D. investigations on "FLOWES" that indicated that my research was feasible. Doctor Fuchen Jia and Angus Grandison have recently helped to fine tune "SMARTFIRE" and to remove a number of implementation glitches and grey areas. Their diligent and thorough fire field modelling validation work (using "SMARTFIRE", "Phoenics" and "Flow3D") have greatly helped during the validation of "SMARTFIRE". Thanks also to Professor Mark Cross for my learning experiences with the "Physica" project.

Doctor Nick Croft has been largely responsible for the quality of this work. His complementary Ph.D. studies in the area of algorithm development, for unstructured mesh CFD, have been the mainstay of this research and his intelligent insights and willingness to help others have been beneficial to many of the researchers in this department, including myself.

Finally, it remains only to thank the University of Greenwich and the EPSRC for funding the "SMARTFIRE" project and this Ph.D. research.

Abstract

This research started with questions about how the overall efficiency, reliability and ease-of-use of Computational Fluid Dynamics (CFD) codes could be improved using any available software engineering and Human Computer Interaction (HCI) techniques. Much of this research has been driven by the difficulties experienced by novice CFD users in the area of Fire Field Modelling where the introduction of performance based building regulations have led to a situation where non CFD experts are increasingly making use of CFD techniques, with varying degrees of effectiveness, for safety critical research. Formerly, such modelling has not been helped by the mode of use, high degree of expertise required from the user and the complexity of specifying a simulation case. Many of the early stages of this research were channelled by perceived limitations of the original legacy CFD software that was chosen as a framework for these investigations. These limitations included poor code clarity, bad overall efficiency due to the use of batch mode processing, poor assurance that the final results presented from the CFD code were correct and the requirement for considerable expertise on the part of users. The innovative incremental re-engineering techniques developed to reverse-engineer, re-engineer and improve the internal structure and usability of the software were arrived at as a by-product of the research into overcoming the problems discovered in the legacy software. The incremental re-engineering methodology was considered to be of enough importance to warrant inclusion in this thesis. Various HCI techniques were employed to attempt to overcome the efficiency and solution correctness problems. These investigations have demonstrated that the quality, reliability and overall run-time efficiency of CFD software can be significantly improved by the introduction of run-time monitoring and interactive solution control. It should be noted that the re-engineered CFD code is observed to run more slowly than the original FORTRAN legacy code due, mostly, to the changes in calling architecture of the software and differences in compiler optimisation: but, it is argued that the overall effectiveness, reliability and ease-of-use of the prototype software are all greatly improved. Investigations into dynamic solution control (made possible by the open software architecture and the interactive control interface) have demonstrated considerable savings when using solution control optimisation. Such investigations have also demonstrated the potential for improved assurance of correct simulation when

compared with the batch mode of processing found in most legacy CFD software. Investigations have also been conducted into the efficiency implications of using unstructured group solvers. These group solvers are a derivation of the simple point-by-point Jaccobi Over Relaxation (JOR) and Successive Over Relaxation (SOR) solvers [CROFT98] and using group solvers allows the computational processing to be more effectively targeted on regions or logical collections of cells that require more intensive computation. Considerable savings have been demonstrated for the use of both static- and dynamic- group membership when using these group solvers for a complex 3-dimensional fire modelling scenario. Furthermore the improvements in the system architecture (brought about as a result of software re-engineering) have helped to create an open framework that is both easy to comprehend and extend. This is in spite of the underlying unstructured nature of the simulation mesh with all of the associated complexity that this brings to the data structures. The prototype CFD software framework has recently been used as the core processing module in a commercial Fire Field Modelling product (called "SMARTFIRE" [EWER99-1]). This CFD framework is also being used by researchers to investigate many diverse aspects of CFD technology including Knowledge Based Solution Control, Gaseous and Solid Phase Combustion, Adaptive Meshing and CAD file interpretation for ease of case specification.

Table of Contents

1 Introduction

1.1 Overview

The widespread and ever-increasing use of Computational Fluid Dynamics (CFD) [SPALDING81] for the simulation of physical fluid flows has highlighted some striking weaknesses to be found in many existing CFD software systems.

The most important problems obstructing the effective use of CFD simulations are:-

- the requirement for large amounts of numerical processing power,
- the extended duration of simulations, the high degree of complexity of the CFD algorithms,
- the poor reliability of the solution process and
- the nature of the development techniques traditionally used to create CFD systems.

Leaving the development issues aside, for the moment, it is worth considering the stages of setting up, running and interpreting the results from a typical CFD simulation because this gives some insight into the complexity of the software. Typically a CFD expert will specify the geometry and the "known" physical and boundary condition properties in a pre-processing specification tool or even in a simple text-based script file. This will be followed by the generation of a suitable computational Finite Difference (FD) mesh. It should be noted that the nature and quality of the finite difference (FD) mesh is critical to obtaining good results from the CFD simulation but the topic of mesh generation is beyond the scope of this research. It is assumed that a mesh of suitable quality is available for all simulations discussed in this research. The numerical computation phase of the simulation will then be started. This phase causes all of the properties, of all of the control volumes within the FD mesh, to be repetitively updated by an iterative algorithm that moves the solution towards a progressively better and better approximate answer. This computational process is generally very time consuming and, for a particular simulation, has no guarantee of ultimately reaching a successful or accurate solution. If all proceeds well with the simulation then the processing will eventually terminate and create some form of output results files which can then be used for post-processing numerical data

analysis or data visualisation. The vast arena that encompasses CFD research has led to a large range of software systems that contain many different algorithms and a myriad of numerical control parameters that modify the run-time behaviour of the various algorithms and solvers. The choice of algorithms and control parameters can lead to widely differing solution behaviour, even on similar simulations, and catastrophic behaviour when used inappropriately. Furthermore a configuration that may not be appropriate at one stage of a simulation may then be suitable (or indeed necessary) at some other stage of processing. The need for algorithm and parametric fine-tuning, for specific classes of CFD simulation, further compound this problem.

This lack of predictable outcome of the numerical processing can be extremely costly in terms of wasted human and computer time resources. Furthermore the solution to any simulation is not guaranteed to produce an accurate or physically meaningful result. This can be extremely costly if the results are to be used for construction design or for safety considerations. Traditionally, when the CFD user has been able to detect an unsuccessful simulation then the whole simulation had to be re-posed and re-started from scratch for even the simplest of configuration changes. A few CFD codes do mitigate these problems by providing various degrees of numerical data or solution status monitoring, during the numerical computations, however the only course of action that is generally available to the user is to terminate a simulation that appears to be unstable or unsatisfactory.

There is little or no reliable knowledge about how to effect beneficial control changes and this means that many aspects of CFD simulation are regarded (and even taught to novices) as a "black art". This is highly unsatisfactory for an ever more widely used simulation technique that is being applied, often inappropriately and inexpertly, to application areas where paramount safety issues exist. E.g. Fire safety aspects of building design, Wing aerodynamics for aircraft design, Cooling system design for Nuclear reactors.

1.2 Aims of the project

Given the problems, with the traditional approach to CFD, this project had been instigated to answer the questions about the applicability and potential benefits of interactive control

techniques for the computational phase of CFD.

1.2.1 Main research question

Prior to this investigation it was not known to what extent CFD systems could be enhanced by the use of interactive control and solution monitoring user interfaces. This was because most existing CFD codes treat the numerical simulation as a "black-box" process that is pre-configured to continue calculating results, to some pre-specified strategy, until processing is deemed to have finished to some prescribed criteria. This project was used to investigate if there are any tangible solution "improvements" made possible by the use of interactive solution control and monitoring. These improvements could be: better performance, greater solution reliability, detection and prevention of errors or simply greater ease-of-use.

The complex nature of all numerical CFD systems means that few users have a complete understanding of how the simulation proceeds from its initial state to a satisfactory set of results. This problem is further compounded by the diversity of the CFD algorithms, numerical approximations, empirical methods, choice of initial conditions and solution control parameters that are used within a particular CFD engine or for a particular simulation. With so many degrees of freedom it is unsurprising that CFD techniques are generally considered to be only useful for- and usable by- CFD experts.

In CFD codes where solution monitoring is provided it generally takes the form of pre-configured graphs or simple numerical value display that cannot be used to explore the full extent of the simulation data in any systematic or comprehensive way. Furthermore the general lack of an interactive control interface means that there is little or no knowledge of what effect control changes will have on the simulation both in terms of speed of execution and continued solution stability. The only general knowledge about CFD simulations concerns the approach often adopted to start a simulation in a reliable and safe way. This knowledge is derived by performing a number of short trial runs with different initial conditions or by using very coarse

computational meshes with few control volumes. Such knowledge does not guarantee that the remainder of the simulation will be successful and it is often necessary for the user to run complex simulations in a number of stages with different initial control parameters for each stage.

Also unknown is the desirability, predicted response and reliability of making interactive changes to the control parameters of a simulation and under what circumstances are the changes appropriate or even counter-productive.

CFD simulation is a highly numerically intensive process and it was not clear to what degree a fully interactive user interface, with all of the problems associated with coupling, would adversely affect the performance of the system. This could have repercussions in terms of user acceptance of the User Interface (UI) supported software due to a perceived poor speed of processing.

It was soon realised that any attempt to perform research on interactive control would be fatally flawed if the numerical CFD component was either unreliable or incomplete. This meant that the largest initial problem facing this investigation was the need for a stable and well-validated software platform on which to investigate user interaction techniques. This could not be guaranteed in a completely new software development because it would be unclear if the reliability of the numerical engine itself was contributing to any observed behaviour during interactive control investigations. Thus it was considered vital to use some form of legacy CFD system as the basis for this research.

1.2.2 Subsidiary research questions

During the course of this investigation it was realised that there were a number of subsidiary questions that required answers.

The requirement for a well validated, robust and complete CFD software system as the basis for the user interaction research lead to the question of how best to re-use such a legacy CFD system in a new development. It was unclear how any legacy system, with all the usual problems

associated with existing "research" developed software, could be re-engineered to allow user interaction and still maintain a reasonable level of solution consistency, performance and extensibility.

It was not known exactly what nature of interaction and quantity of status information would be needed for the system user to determine how to modify the system controls in order to effect the best solution strategy.

1.2.3 Questions arising during this research

During the re-implementation of the CFD code it was realised that the Object Oriented approach adopted for the data structures and control hierarchy would allow highly beneficial modifications to the traditional solution strategy.

One such innovation was the use of "group" solvers which, it was hoped, would provide a framework for investigating localised solution control based on either geometric- or solution determined- regions. This extension was considered to be of sufficient importance to warrant investigation, in its own right, because most traditional CFD codes (particularly those using unstructured mesh storage techniques) do not provide a sufficiently flexible architecture to enable the benefits of such techniques to be researched particularly with the added perspective of interactive solution control.

Also considered was the potential for the automation of any manual control strategy that was demonstrated during this research. This was of considerable interest since the manual interactive control of simulations has a high human resource overhead because an expert CFD user is required to monitor and control the software throughout the duration of the simulation, which, for complex and large mesh cases may extend to many days or even weeks on all but the very fastest hardware platforms.

1.3 Objectives

The aims and research questions described above lead directly to a set of objectives and goals

that constitute a research program capable of providing answers to the unknowns of this study.

1.3.1 Main objective

The main objective of this investigation was to research and test for the potential benefits and disadvantages of using interactive control and monitoring of a CFD code during the computational simulation process.

1.3.2 Subsidiary objectives

The main objective relies on having a suitable interactive CFD code on which to perform the research and so the first subsidiary objectives that must be met were: to analyse, design and implement a fully functional interactive CFD system and subsequently to use it to investigate interactive control techniques. This lead to the following sequentially ordered objectives:

1.3.2.1 Analyse the requirements for the new interactive CFD system

The new interactive CFD system would require sufficient numerical simulation functionality to run non-trivial CFD cases with adequate control options to allow experimental research into the potential benefits of interactive control. This meant that the requirements analysis of the target system would have to be performed prior to designing the architecture and functionality of any new system.

1.3.2.2 Design a prototype software system and user interface

The required system design was imposed on the legacy CFD code to extend the capability to fully interactive control and monitoring. A suitable programming paradigm and target implementation language were selected to enable the user interface and numerical components to be coupled. Wherever possible, heavy use was made of "good" software engineering principles to ensure that the development system has an extended useful life and can be used in the future as a comprehensive CFD application framework for a multitude of research purposes.

The type of user interface and style of interaction were agreed with potential CFD code users so that both the paper- and skeleton-UI prototypes would be constructed so as to allow the target system user interface to be specified fully with assurance of user acceptance.

The User Interface components were chosen to provide the clearest possible view of the current status of the simulation without unduly affecting performance. Controls and monitoring displays were grouped so as to provide simple interface navigation and furthermore the interface was restricted in depth to prevent the user being lost in hard to reach sub-menus. The User Interface was also designed to support both novice and expert users alike. It was necessary to keep the layout and mode of interaction of the interface consistent across all components of the user interface. The actual approach and mode of interaction used were agreed by using evaluation prototypes for discussions with expert CFD practitioners and researchers. Prototypes of the software were also used on taught courses in Fire Safety Modelling, at the University of Greenwich, in order to better understand how novice users would respond to the interactive user interface.

1.3.2.3 Reverse Engineer the legacy CFD code

A methodology was created to re-use the legacy CFD code and to impose the required system design whilst being aware of the potential pitfalls of the traditional approach to CFD development and avoiding the usual handicaps of existing CFD codes. The Reverse Engineering techniques that were used had to maintain absolute functional consistency with the legacy code whilst providing a sufficiently flexible application framework for continued research into CFD techniques.

1.3.2.4 Implement the interactive prototype

The prototype interactive CFD system was constructed by the coupling of the agreed prototype user interface and the re-engineered CFD engine.

1.3.2.5 Validate the correctness of the interactive prototype

The prototype interactive CFD system was validated for computational consistency with the results from the legacy CFD code, with results from other commercial CFD codes and, where available, with experimental data. The new interactive system had to perform a selected range of simulations to an acceptable solution tolerance when compared with existing comparison codes, experimental data or analytical solutions. A suitably diverse coverage of validation cases was devised to exercise all of the coupled numerical modules within the system. Expert CFD practitioners were asked to check that the prototype CFD system produced acceptably consistent results.

1.3.2.6 Construct suitable test cases to test for the benefits of interactive control

Test cases were chosen to evaluate the benefits and disadvantages of the interactive nature of the prototype system.

An investigation of the time benefits or overheads due to the use of user interaction could not be performed easily in different CFD systems. This was because of differences already inherent from the development languages, internal architectures and solution algorithms. Such differences already produce large variations in run times between the various CFD systems. This effect is observed without even considering the influence of the User Interface. Test cases were constructed and used to investigate the benefits of user interaction when optimising the solution strategy as the run proceeds when compared to a "batch mode" run of the same software. These timings were then compared with the non-optimised simulations to give a reasonable indication of relative performance. Other, less quantifiable, benefits (e.g. error detection and prevention or stability enhancement) could really only be investigated by allowing real users to experiment with their own simulations to see if either stability or timing enhancements could be made in practice, but such investigation was outside of the scope of this study. The qualitative benefits, observed during this research, are discussed as appropriate but no extensive investigation was conducted into these benefits.

Group solver control techniques were also investigated to determine if dynamic or static

membership groups offered any benefits to CFD simulation and to ascertain if interactive control of groups was in any way beneficial.

Also researched was the potential for automated solution control using the knowledge gained during the optimisation research mentioned above.

1.4 Contribution to knowledge

The following summary indicates the significance of this work based on the limitations and problems of existing CFD systems in general and the legacy source CFD code in particular.

1.4.1 Development of an incremental re-engineering methodology

The techniques created to reverse-engineer and re-engineer the legacy FORTRAN CFD code, using a novel incremental approach that preserves functional consistency, are likely to prove beneficial to a wide range of legacy numerical software systems both within and outside of the domain of CFD research.

1.4.2 Investigation of new CFD techniques

The prototype system has been used to research numerous enhancements made possible by the structure, availability of dynamic memory allocation and the Object Oriented design paradigm imposed during the software re-engineering. This has allowed research into the benefits of unstructured group solvers [EWER99-3] to be investigated as well as research into the automation of interactive control experiences using Knowledge Based System (KBS) control techniques.

Other techniques, outside of the scope of this thesis, are also being developed within the Smartfire framework [TAYLOR96]. This work is facilitated by the open and extensible software architecture. These techniques include run-time mesh adaption, fire modelling using solid combustion, modelling of thermal radiation and simulation of flash-over.

1.4.3 Investigation of the benefits of interactive control

The prototype interactive CFD system (now being used as the interactive CFD component of the "Smartfire" system [TAYLOR97-1]) has demonstrated some significant benefits for the use of interactive control particularly for the control of solution stability and for solution optimisation.

The interactive control of unstructured "group" solvers has indicated some very important savings for simulation times for cases that have marked solution differences between geometric regions. Furthermore the group solver control allows significantly stratified (layered) flows to be effectively controlled so as to maximise stability whilst minimising computational effort.

1.4.4 Knowledge of practical techniques of interactive control and monitoring of CFD codes

A further technique that has been researched as a prototype within the interactive CFD system is the use of expert CFD user knowledge at controlling the CFD code within an automated Knowledge Based System to represent and act on the rules which can be used to improve the performance or stability of the CFD simulation during processing. The KBS system uses rules that have been elicited from actual simulations that have been interactively controlled by expert CFD users. This research was made possible by the existence of the interactive control interface in the prototype system and the open "Blackboard" architecture which supports external control agents, other than a human operator using the User Interface. Furthermore experience of the issues concerning implementation and interactive control have been attained.

1.5 Practical contribution to CFD research

The prototype CFD system developed in this investigation has evolved into a comprehensive environment (now called "Smartfire") that is extensively used for continuing CFD research within the application domain of fire simulation at the University of Greenwich. This is mostly due to the considerable flexibility created by the re-engineering techniques employed during its development. The interactive control and monitoring interface is popular [HUME97] for a

research based code because it drastically reduces the time taken to assess and monitor the behaviour of newly developed research algorithms.

1.6 Background to this research

The origins of this current work go back to 1987 when Knight, Cowell and Edwards [KNIGHT87] investigated the benefits (to CFD) of using strict Software Engineering design and development techniques for the development of reliable and extensible framework of CFD research. This theoretical consideration highlighted some of the problems that are discussed in this thesis but was not researched in practice due to resource limitations. The investigation was extended into practical research by Petridis in 1995, for his PhD research [PETRIDIS95]. This later research primarily investigated the potential benefits of an integrated Knowledge Based System (KBS) for the dynamic control of a Heat transfer code during the numerical simulation. One problem which was noted during the research was the lack of expertise that was required to control the CFD code in terms of appropriate decision making. This was largely due to the nature of most CFD codes which use batch mode of processing that is pre-configured to solve some flow scenario without any form of user intervention. Further problems, facing the research, were the limited time constraints for development of a prototype system and the lack of access to comprehensive and reliable CFD software which meant that the prototype system only had quite limited capabilities when compared with the fully coupled flow, heat, turbulence and radiation algorithms that are commonly found in commercial CFD codes or more recent in-house research codes. The lack of flow and turbulence handling was particularly restrictive because these sub-models constitute the majority of the complexity of any general purpose CFD system.

The previous research demonstrated that there were important improvements to be made to CFD software if appropriate control expertise could be determined and encapsulated in a KBS. Whilst some expertise had been gained during this prior research, it was by no means complete because of the limited scope and capability of the prototype system. Another problem that was identified was the question of how sufficient information about a particular solution status could be represented for initiating any KBS reasoning [EWER93-1].

A more ambitious project was initiated in order to answer the joint problems of what sort of control to apply and under what conditions to apply it. It was also the intention of this project to create a research tool which could be used to investigate the benefits of interactive software control within the CFD arena on a non-trivial (and preferably safety critical) application area where any benefits would have real and demonstrable importance. Furthermore it was intended that the research tool would form the basis of a CFD application framework, called "SMARTFIRE", which would not only support both KBS and interactive control techniques but also provide a vehicle for continued research within the University of Greenwich.

Collaborative work has also progressed, with other researchers, on other aspects of the "SMARTFIRE" system. One of the problems facing CFD users is the set-up and specification of a case such that the best solution can be obtained in the most efficient way. To this end, researchers have been investigating the scope for using automated set-up tools [TAYLOR97-2] which can take a simplified case specification (usually the geometry and the boundary conditions) and then automatically generate high quality specifications for the CFD simulation.

1.7 Structure of this Thesis

Chapter 1. This chapter has enumerated the research questions posed and answered during this investigation and has also indicated how the objectives of this thesis were decomposed into sub-goals. There is also a summary of the importance of this research in terms of the contribution to knowledge in general and to CFD in particular.

Chapter 2. This chapter gives a background to this area of research and gives an overview of the techniques commonly employed in fluid flow simulation at the beginning of this study. The chapter also discusses some of the alternative techniques and implementation languages that were considered during the early investigations of this research. The chapter finally indicates some of the most recent developments (or lack thereof) in the field of CFD code development.

Chapter 3. This chapter describes the capabilities and limitations of the legacy CFD engine used as a basis for these investigations. The chapter also includes a description of the novel

methodology that was developed for the re-use of the legacy CFD software in order to provide a framework for the research required for this project. The reverse engineering and re-engineering principles developed during this investigation are of significant interest in their own right, particularly when one considers the large amount of useful legacy code that is still in use but is often difficult to maintain or to integrate with newer applications.

Chapter 4. This chapter describes the creation of the prototype CFD system from the re-engineered legacy software. This is the "vehicle" that was to be used for the investigations into the potential benefits of interactive control.

Chapter 5. This chapter discusses the validation of the prototype CFD environment and compares the run-time behaviour with the legacy software from which it evolved. This was necessary to check that the reverse-engineering process has not corrupted the functional behaviour of the software.

Chapter 6. This chapter introduces a set of simulation cases with their results, and consequent interpretations. These test cases were used to investigate the potential benefits of interactive control and monitoring.

Chapter 7. This chapter discusses some preliminary findings of research that was conducted into solution optimisation. The two techniques that were investigated were automated dynamic solution control and a new solver technique called a group solver.

Chapter 8. This chapter offers conclusions about the benefits of these investigations.

Chapter 9. This chapter indicates the need for additional research. This additional research could not be completed or fully investigated due to time constraints.

The Appendices include technical descriptions of the algorithms, design and development techniques employed in the particular class of CFD code used during this research. Published papers, from the author, which are directly relevant to this investigation have also been included in their entirety, so that referring sections of this thesis could be written more concisely.

2 Background to interactive CFD research

2.1 Overview

This chapter discusses some of the features and limitations of the most common commercial CFD codes and assesses some of the approaches that have been used to "improve" CFD modelling prior to this current research. The reasoning behind the decision to use a legacy CFD system as the base code for the current research is also explained. Having established the need for using legacy software, the various techniques that are available for effecting the re-use of a legacy CFD system are discussed. Finally there is a critical assessment of the CFD techniques and previous development style, that were found within the legacy CFD system used for this study.

In order to gain some perspective of the mode of operation and structure of the prototype CFD system it is worth considering the capabilities and mode of interaction typically found in other CFD codes. It should be noted that the CFD codes are not being evaluated for their complexity or relative accuracy of their modelling techniques, or for the diversity of application areas that they cover. Rather the discussion centres on the techniques used for interaction and control, as well as the assurance of reliable simulation and accuracy of results.

Clearly most, if not all, commercial CFD codes are undergoing continuous enhancement and many facilities have been added or further developed during the period of this current research. However, even now, few CFD developers are aiming to provide code interactivity and automated solution control. The observed development emphasis is usually directed to the enlargement of the range and diversity of cases that can be run with the software, the improvement of the numerical models and approximations used in the software and the provision of better quality set-up (i.e. case specification), meshing and post processing data analysis tools. This development strategy seems to assume that the CFD users will always be CFD experts but it has recently been observed that non CFD experts are often turning to CFD techniques in order to support their own areas of expertise.

The various CFD codes, mentioned in these discussions, are by no means exhaustive. The intention is to include a representative set of CFD codes that adequately illustrate the currently available methods and techniques.

2.2 The traditional approach and work elsewhere

2.2.1 Batch mode CFD codes

There is generally insufficient information to apply reliable control of batch mode CFD codes since only residuals and spot value monitoring are provided. Whilst this is less of a problem for simulation optimisation by the control of relaxation values it is a severe problem for the appropriate handling of unstable or divergent solutions where the cause of the problem behaviour is not known.

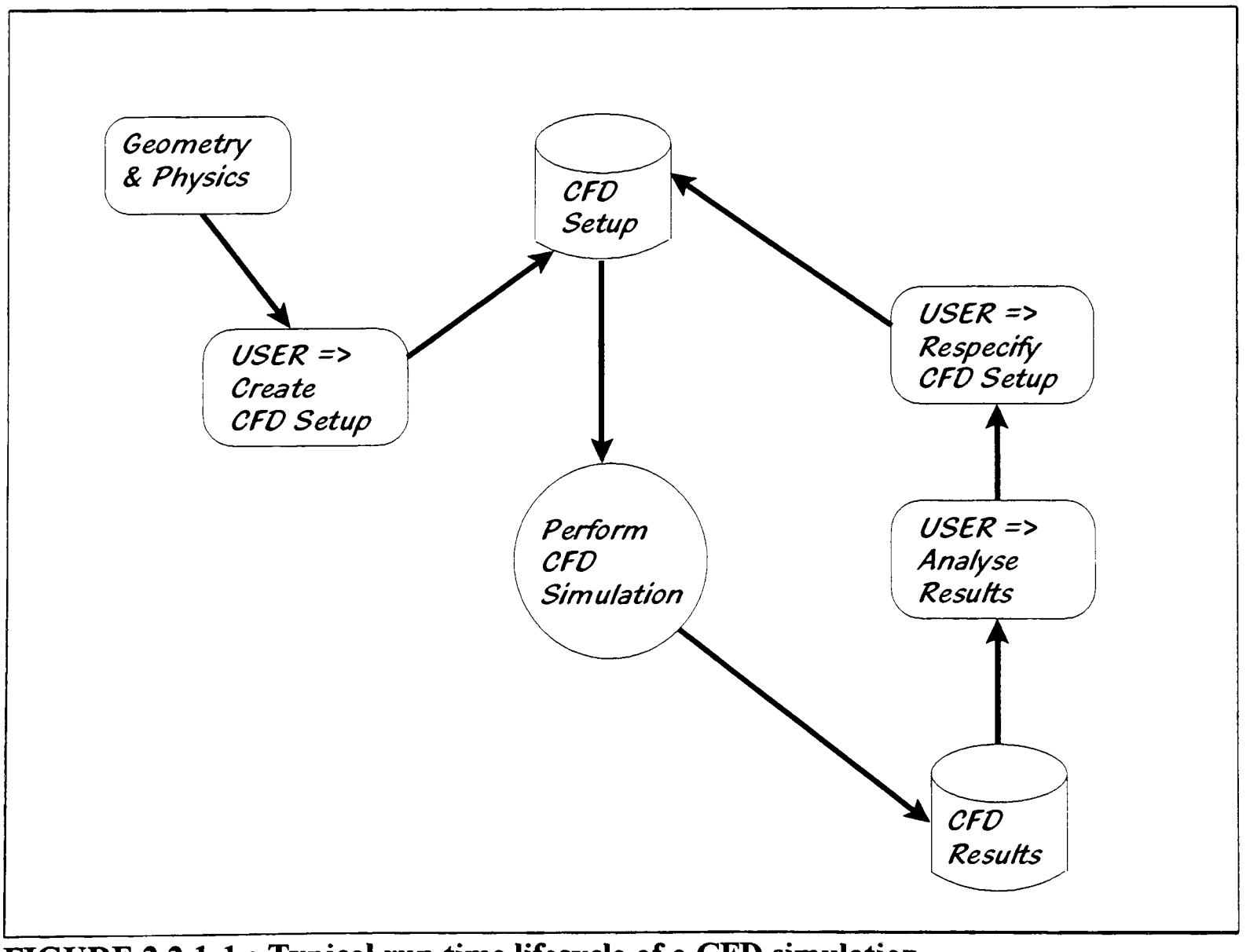


FIGURE 2.2.1-1 : Typical run time lifecycle of a CFD simulation.

The figure (See Figure 2.2.1-1) shows a typical run time simulation life cycle that shows user interaction, processing and outputs. Generally the set-up / configuration, meshing and post processing data analysis tools are all completely separate from the numerical CFD engine. The user cannot easily make use of data visualisation to continuously monitor the solution because the user would have to force the system to output frequent complete result file dumps. Similarly the user is generally unable to "interact" with the solution controls because this generally requires saving a complete restart dump, going back to the set-up tool in order to re-configure a restart using modified control parameters and finally reloading the case and restarting to continue processing.

There are a wide range of CFD codes that fall into this category, including:
Phoenics [CHAM], Flow 3D [FLOW3D91], Easyflow [EASYFLOW90], Fluent, and Astec.

2.2.2 Other approaches to improve batch mode CFD codes

Several attempts have been made at developing Intelligent Front Ends [WILLIAMS88] [JAMBUNATHAN91] for CFD systems. Whilst this research has some scope for improving the CFD simulation process, because of the importance having a high quality case specification, it is likely that the problems facing CFD research (described in Chapter 1) will still apply. Unfortunately the research only used Knowledge Based techniques to support the set-up and specification of a case and did not modify the, essentially, batch mode numerical CFD engines.

Phoenics with rules was an attempt to control the CFD processing centred on an experimental routine build into the Phoenics commercial CFD code. This module contained a fairly simple algorithm to modify the relaxation parameters based on the latest values of residuals for some of the solved variables. A paper by Spalding [SPALDING92] presented simple implementation details of the production rules without validation of the potential benefits.

2.2.3 Other interesting and / or relevant research

"FUNGI" [UPHAM94] is an interesting development because it employs Lex [KERNIGAN88-1] and Yacc [KERNIGAN88-2] to interpret Finite Difference algorithms in a graphical environment. Whilst these Finite Difference algorithms are generally not as complete as those that would be required for a CFD simulation and there are no supporting variables, it is interesting to see a system where the fundamental algorithms are not necessarily hard coded into the source code. The algorithms are actually interpreted from set-up information and the GUI parameters. This form of external algorithm configuration was investigated briefly to allow simple calculation of auxiliary variables within the prototype system.

Similar research has been conducted by Edwards and Hayes into a Visual Programming Interface for Iterative Methods [EDWARDS93]. Early discussions with CFD researchers led to this form of interface being discarded because, whilst extremely powerful for research of algorithm design, it is not especially useful for monitoring the solution status or controlling the simulation.

Scateni has also investigated the potential for creating an Integrated Object Oriented Computational Fluid Dynamics Environment [SCATENI92] but this had only resulted in the development of an Interactive Domain Editor.

Cortex [BANERJEE94] is a co-processing visualisation system designed to allow for interactive control of a CFD system but it requires a specially modified CFD interface for control and data passing. It was also designed for very powerful parallel systems. This is an interesting design for a configurable GUI for co-processing visualisation for CFD simulations.

NetCFD uses a WWW browser to remote control a CFD code (femFlow) from a remote high performance CFD system. The system currently limits data to 2D only at present. The CFD engine is based on the Finite Element Method and this has been criticised by some CFD researchers to have some problems with certain flow simulations. The system also requires high performance networking for communications between the remote processing server and the local

GUI system.

Tworzydlo and Oden have written on the subject of creating an automated environment for computational mechanics [TWORZYDLO93] but this work was not particularly well advanced.

2.2.4 Prior CFD research in the University of Greenwich

Phoenics [CHAM] and Flow-3D [FLOW3D91]: These commercial codes have been used as development environments for the research of additional CFD methods covering nearly all aspects of simulation capability. As yet no research has been applied to the creation of interaction or monitoring tools beyond being able to see residual / spot value graphs and being able to break into the processing to re-specify the set-up.

UIFS: This FORTRAN 2-D unstructured mesh stress and control volume CFD code was the pre-cursor of much of the in-house research at the University of Greenwich. This code was primarily aimed at the simulation of coupled solidification, stress modelling and fluid flow scenarios.

FLOWES [PETRIDIS92]: This C and prolog research tool combined a simple thermal transfer code with a rule inference engine that could automatically modify the solution control parameters based on automated monitoring of the solution status.

CWNN [CROFT98]: This FORTRAN CFD code was a 3-D enhancement of the earlier UIFS code. The aims of the code were to combine multiple physics capabilities including stress, solidification and fluid flow modelling (amongst others) into a single coupled environment that was able to use fully unstructured meshes. This was the legacy code selected for the re-engineering.

2.3 Recent developments

Physica [BAILEY95] [CROSS94]: This FORTRAN code was developed as the embodiment

of the multiple physics concepts on unstructured meshes that are found in CWNN and UIFS, both of these earlier codes were merely research tools used in the development of reliable and fully coupled multiple physics techniques. The emphasis of Physica is for broad coverage of CFD capabilities with high degree of portability and, where possible, the exploitation of high performance computing architectures.

FLO++: This C++ CFD code is quite novel in that it breaks the mould of traditional research techniques which have, almost exclusively, used FORTRAN. The software provides user development routines for extension of the software. The numerical engine is still quite batch mode oriented.

CFX: This FORTRAN CFD code is a recent successor to the earlier FLOW-3D code. The capabilities have been extended beyond those of FLOW-3D but there has been no attempt to incorporate any interactive techniques into the CFD engine, which is still essentially, a batch mode process.

Colt / Phoenics VR [CHAM]: This was a new concept for utilising high powered remote servers to run CFD problems with a set up tool and post processing environment running locally on the user's workstation. The user would create the case locally and "send" it away for simulation. A novel feature of the post processor is the VR style interface that allows the user to "walk through" the geometry and the results data as if it were a real geometry. The CFD simulation would still be run to completion in batch mode by the standard Phoenics CFD code.

2.4 Assessment of the development techniques available to develop a framework for interactive CFD research

This section discusses the various techniques that were available to create a CFD application framework for this research investigation. The various pitfalls and benefits of these approaches are also discussed.

2.4.1 Develop a new CFD engine from scratch and add interactive techniques

It would be possible to create a CFD framework from scratch based on the plethora of available literature. Many of the papers about CFD cover many advanced topics of CFD research and the fine-tuning needed for specific application areas. The problem is that there are very few "single" information sources, which cover all aspects of developing a general-purpose CFD code. There are also a number of additional problems with regards to this approach.

2.4.1.1 Development time factor

Many CFD codes (either commercial or in-house research codes) have been developed over a period of many years by numerous developers. This is largely due to the diverse methods available for CFD research as well as improvements and replacements for both the generic numerical methods and application-area specific empirical techniques. In this sense most CFD code development is an evolutionary process both in terms of increasing hardware capability (which tends to highlight limitations in traditional techniques as the problem size and simulation complexity are increased - For example the availability of more memory and faster hardware may lead researchers to use finer meshes which may actually give less stable simulations or worse results because there are considerably more degrees of freedom for the flow path in a finer mesh) and software capability. Given the limited duration, human resources (Approximately 3 person-years) and CFD expertise available for this research it was concluded that it would not be possible to create a complete and well validated CFD system from basic first-principles.

2.4.1.2 Reliability of CFD software

The huge investment in man-power to develop CFD codes and the customary evolutionary development life-cycle of such codes mean that most CFD codes have a large and dedicated following of users who trust in the capabilities of particular codes (It is also the case that CFD codes have many optimal branches and fine tuned coefficients that are tailored to specific application areas). Indeed many CFD users are so tied to a particular CFD system that they are incapable or extremely reluctant to use any other CFD software outside of their previous

experience. Any new development has to satisfy an extensive coverage of validation against both experimental data and other CFD systems if it is to obtain widespread user acceptance. Such reliability could not be guaranteed in any completely new code development prior to starting research into interactive control and indeed it would be unclear if any observed system behaviour, to interactive control, is due to problems in the reliability of the software or the actual control strategy adopted.

2.4.1.3 Access to CFD expertise

Part of the problem facing new CFD system developers is access to applied mathematicians who are familiar with the "fine-tuning" techniques used in CFD codes. This is vital where necessary approximations and empirical techniques have to be used to extend the software capability into new research areas. Starting a new code development from scratch would be fraught with difficulty due to inexperience on the part of the software developer and the steep learning curve associated with gaining the necessary development skills. This is particularly true of this researcher whose background is in software engineering and physics rather than in the more appropriate applied mathematics.

2.4.1.4 System capabilities

The nature of fluid flow simulation is so diverse in scope that few, if any, CFD codes can hope to behave well in all application areas. Even a limited application area such as fire field modelling is actually very complex when the fundamental physical and chemical processes involved (and their interrelationships) are actually considered. This complexity is further compounded by the nature of the approximations and simulation techniques used within CFD software. Any new development would have to undergo extensive research and testing merely to become an "adequate" simulation system in a specific application area.

2.4.1.5 Other problems with developing from scratch

As previously mentioned any implementation of a CFD code from basic principles is likely to

take considerable time to develop due to the complexity and nature of the algorithms used. Furthermore such development is fraught with difficulty particularly for finding the empirical methods algorithm fine-tuning that give the best possible approximation to true physical behaviour for the target application area when no accurate numerical model exists. It is highly likely that a new code development would get the design and data architecture correct for the desired system but it is probable that the new development would have errors and omissions in the empirical models and core algorithm formulation that could only be removed by extensive research and iterative improvement. Research would need to be conducted into the algorithms and empirical methods employed because various combinations of numerical techniques behave in very different ways even when disregarding the diversity of the target application areas. Most CFD codes are the combination and accumulation of many titbits of implementation knowledge from diverse information sources where such knowledge ranges in scope from the particular values used for a few constants in a particular application area to choices for more reliable solution algorithm for the coupling of the solved variables. Validation of a new development is particularly difficult because of the long development period before the system is in a state that can actually be used to run simulations and at such a stage any errors encountered would require extensive modifications to the underlying software. As has been found in many previous new developments there is unlikely to be trust in the development from potential users without extensive validation against existing software systems and experimental cases. The duplication of effort for developing a new CFD code is particularly problematic for a limited duration project. Generally a new code development would have no support from other developers since the information sources would be reference texts, journal papers and conference proceedings with little access to numerical CFD software developers who have their own work loads.

2.4.2 Add interactive functionality to an existing commercial CFD system

At the beginning of this research a feasibility study was performed to ascertain if an existing commercial CFD code should be used as the framework for this investigation. There were a number of perceived problems with this approach.

2.4.2.1 Sensitivity of commercial software

Most commercial CFD software is closely guarded by the developers because of the huge investment of resources required to develop and maintain it. Access to some of the empirical approximations, numerical methods or specific implementation techniques would give competitors considerable advantage. It is generally the case that commercial software is only infrequently available completely in source code form.

2.4.2.2 Access to commercial software

As previously mentioned the new research requires complete access to all of the software internals because of the requirements for interactive dynamic control. Also the methodology used to incorporate User Interface code within a software system are generally highly invasive for the provision of comprehensive interactive control, run-time data monitoring and run-time visualisation.

2.4.2.3 Authority to modify commercial software

Given the sensitivity of most commercial CFD codes it is unlikely that authority would be given to re-engineer or modify such a code extensively because such modifications would create several different versions of the software which would then have to be maintained. If a decision was taken to only proceed with the modified "interactive" version then that would have implications on the development techniques that would have to be employed (e.g. QA procedures, regression testing, comprehensive validation, software design with management walkthroughs) and also on the company requirements for the delivery software. Such commercially driven research would be unlikely to be as flexible as pure research because of the limitations of product and user requirements which often cannot afford to support extensive speculative pure research.

2.4.2.4 Access to developers' expertise

It is unlikely that a commercial software development house would welcome external access to

the CFD development team because of the likely interruption to work schedules that such interaction would cause. Such access is vital if the new code development is to keep up with bug-fixes, patches and code improvements and is not to stagnate as a dead-end system. The software developer is also likely to need significant help with understanding and correctly accessing the internals of the code.

2.4.3 Extending the capabilities of the existing partially complete CFD code : "FLOWES"

In the case of the "FLOWES" system it was decided that there was too much missing from the basic algorithms to contemplate extending the software because such extension would be very similar to the previous section which described the difficulties facing a new code development. This was particularly true in light of the fact that it is the complex flow and turbulence modules that were not developed within the "FLOWES" software. As mentioned above, the reliability of a completely new software development would be largely unknown and untested and significant research would be required into the choices and fine-tuning of algorithms and empirical methods. Extending an existing system would mean that the implementation language and, to a lesser extent, the data structure would be fixed. This is not necessarily desirable when considering the potential for extended research. There would be no support available from developers since the legacy system was essentially an unsupported prototype that is not going to be developed further or maintained.

2.4.4 Simple automated translation of a legacy CFD system

Some cursory investigation of the tools available indicated that such an approach would lead to little or no improvement in structure of either the data architecture or the procedural hierarchy. It is also possible that there would be small errors in translation process particularly in terms of order of execution of compound statements and array handling between languages. Due to the low stability and fine numerical tolerances of most CFD algorithms it is probable that very minor differences could cause destructive and unpredictable behaviour. However this technique does benefit from the fact that the implementation language can be chosen such that suitable tools and

software libraries are available for the overall development. There are several portability issues relating to some of the available translation tools particularly in the handling of the external files (which is a vital aspect of CFD systems) which are used to maintain complete sets of set-up and solution data outside of any particular run of the software. There would be no documentation, other than that which was available with the legacy CFD code. Generally the translation would be simple to do and quite quick however the quality of translated software is frequently indifferent and often quite poor. This is particularly true of translation software that is intended to allow compilation but not necessarily maintenance or enhancement within the target language. The approach does have one significant benefit because the new system maintains a reasonable one to one mapping with the legacy system but this is only useful if developmental and research work is to be conducted entirely within the legacy system.

2.4.5 Creation of nearly unchanged libraries of numerical routines and the imposition of high level structure

A part way solution to the use of legacy software, whilst acknowledging the need for improved structure, is to impose a high level structure on the software whilst turning much of the legacy code into utility routines and library procedures that now make use of an improved high level data structure or routine hierarchy. This approach could be reasonably easy to implement but depends largely on the existing procedures and data access mechanisms and the clarity and flexibility of their implementation. It is possible that the approach could benefit from the use of mixed language programming but this would adversely affect the system portability. It is also possible that the desired high level structure would not necessarily be consistent with the legacy routines. This technique could be reasonably quick to implement provided that no difficult inconsistencies were encountered. Many of the legacy routines could be used "as-is" depending on the pre-existing structure and nature of the language used. This is likely to result in a straightforward path for upgrade, patches and bug-fixes but easy integration of legacy routines cannot be guaranteed. Performance is likely to be good provided that functional data access or re-assignment are not required as a means of accessing or passing data to low level library routines. It is also likely that code clarity would be improved at a high level within the code but would be generally poor at lower levels. Some of the problems would be dependant on the

nature of the data access mechanisms that would be imposed by mixed language development or the interface to the library routines.

2.4.6 Reverse engineering of a legacy CFD system back to basic design and re-implement from this design

This process would take a considerable time and is by no means easy. The reverse-engineering techniques available [BYRNE91] [BRAND96] to "mine" an existing software system for the underlying design are by no means infallible [BERGEY]. Furthermore it is possible and indeed quite likely that errors will be introduced in the re-design or re-implementation phases of re-engineering. This technique does have the benefit of considerable flexibility in the choice of implementation language. As with a new development there is a long time before the newly developed code can be tested against the legacy code behaviour. However the behaviour is more assured than simply developing a new code from first principles because the design has been extracted from a working and complete software system. Reverse engineering will generally preserve all of the algorithms and empirical methods but may not necessarily preserve an audit trail back to the legacy software. This is particularly true if the structure and data architecture have been drastically altered in the re-design process. It is likely that any updated methods and bug fixes would also have to go through a complete re-engineering and re-implementation process in order to be assured of functional consistency.

2.4.7 Reverse engineering of a legacy CFD system and re-implementation using an incremental approach and imposed data and control architecture

This approach to re-engineering has to make concessions in the design to maintain consistency with legacy system whilst incrementally re-engineering. One of the most important benefits is that the re-engineered system (and the incremental stages) are never very far from a working CFD code that can be validated as being algorithmically correct and consistent with solution behaviour from the legacy software. This gives considerable assurance that the algorithms and empirical methods are preserved during re-engineering. The effort involved is significantly less that would be needed for a complete re-engineering re-implementation from a reverse-

engineered fundamental design, however the resultant system is less likely to have a "perfectly designed" architecture due to dependencies and structure inherited from the legacy software. This technique is more reliable than automated translation or re-structuring since the re-engineered software will be designed so as to improve the architecture to create an extensible, clear and consistent system. The target implementation language can be chosen as required but this will necessitate some form of translation at one of the incremental stages. This technique provides a form of audit trail back to the legacy software so that updated methods and bug fixes can follow through into the new system with less effort than would be found with a totally re-designed system.

2.5 Choice of Implementation language

2.5.1 Overview of language choice

Given that a large number of the overall system requirements had already been specified it was possible for the implementation language choices to be enumerated. Whilst, in theory, the choice of target language was not vital prior to the re-development phase it was found to be helpful to know what programming paradigms and language features would be available for use during the re-engineering.

Languages such as Smalltalk were not considered due to their huge performance hit, caused by their interpreted nature, even though the conceptual structure of such languages is capable of use for implementing scientific software [DUBOIS-PELERIN92].

The legacy CFD software was actually implemented in FORTRAN-77.

2.5.2 Available languages

2.5.2.1 Ada.

Ada had few libraries and suffered from generally quite poor portability due to the limited

availability of quality compilers. Such compilers as there were had a high cost. At this stage the compilers also exhibited limited reliability because Ada is such a complex language. Ada is not quite Object Oriented. It was possible to create very robust software if the particular compiler implementation was correct because of the very strict type checking. It appears that were only a few numerical users. Algorithmic development and use as a research language is likely to be hampered by the very strict type checking and lack of familiarity on the part of the developers. Ada is moderately close, semantically, to the legacy code language.

2.5.2.2 Pascal.

Pascal is often considered to be rather an academic plaything. The portability of Pascal is quite poor. The language is not really Object Oriented although record like structures are available. It has been observed that there are inconsistencies in behaviour between different compilers and platforms. Some compilers exhibit quite good speed but this is highly compiler dependant. There were very few portable libraries available for Pascal. There were few serious numerical developers. Certain compilers and language features helped to ensure reasonably robust implementations. Pascal is moderately close to the legacy code language.

2.5.2.3 FORTRAN-77.

FORTRAN-77 has no Object Orientation hence it would be necessary to emulate Object Orientation using simple data structures and possibly common data and entry points [AFZAL94]. FORTRAN does display good speed and excellent portability due to its maturity and strict specification. Generally there are rather poor language features and only a few portable GUI libraries. FORTRAN has very good numerical libraries but these are not necessarily useful for the CFD system. FORTRAN does not generally support any operating system interface. The majority of numerical developers use FORTRAN. There is little support from the language for robust coding but there are a number of commercial code analysis and code tidying support tools available. FORTRAN is the language that was used to develop the legacy code.

2.5.2.4 FORTRAN-90.

FORTRAN-90 had only a few compilers available at the start of this investigation. Compiler reliability was currently suspect particularly for compilers that translated from FORTRAN-90 to C as part of the compilation. There was rather a lack of libraries other than via FORTRAN-77. Again FORTRAN-90 is not quite Object Oriented although it had introduced more complex data types than were available in FORTRAN-77. There were few users but the language is FORTRAN-77 compliant. The language includes enhanced numerical support for vector and matrix algebra. Generally there was good numerical speed but often, surprisingly, much slower than FORTRAN-77 compilers on the same platform. FORTRAN-90 is consistent with the legacy code language.

2.5.2.5 C++.

C++ has a fully Object Oriented paradigm but the usage can be sometimes be somewhat obscure. There were generally many diverse application libraries particularly for user interface development. The operating system interface built into the language is very good. C++ does suffer from less run-time performance than purely procedural languages (poor speed for dynamic or "late" binding). There is reasonable support for robust coding with moderate to strong type checking but the possible use of C style pointers could give dangerous unrestricted data access. C++ had a quite limited number of numerical developers but its popularity is growing. The language definition is being extended continuously and developers must be aware of the portable sub-set of language if portability is an issue. C++ is a highly flexible and extensible language with considerable possibility for optimisation. C++ has moderate to poor consistency with the legacy code language.

2.5.3 Language chosen for the re-engineered system

The need for a generally portable prototype system that was capable of supporting the Object Oriented data structure paradigm tended to indicate the use C++ as the target implementation language. The wide availability of user interface development libraries and easy access to the

operating system also supported C++ as a suitable language. The deciding factor was the availability of Knowledge Based System development tools that are either compatible with C++ or accessible from within C++.

2.6 The methodology that was adopted for the development of a CFD research framework

The fundamental considerations for the development of the prototype system were for reliable CFD modelling coupled with rapid implementation. The re-use of legacy software offered the greatest potential benefits due to availability, access to knowledge and compatibility with existing pre- and post- processing tools. When the various CFD code development techniques available were considered, for the current project, the incremental re-engineering approach offered the most appropriate solution based on absolute functional consistency, limited development time, access to CFD code developer expertise, flexibility for re-design and ease of maintenance.

The biggest problem facing this technique is that a change of implementation language would require a complete and accurate translation of all of the source code with all of the potential problems that this would entail. This was not seen as an insurmountable difficulty because the incremental re-engineering technique would tend to support the translation stage due to earlier re-structuring stages that are concerned with clarifying the legacy code and ensuring that data access and procedure usage are consistent throughout the whole system.

A complete description of the re-engineering methodology adopted, for this investigation, is given in the following chapter (See Chapter 3).

3 Re-engineering the legacy CFD code

3.1 Overview

Having established the need for using a legacy CFD code as the basis for the creating of a CFD research framework, this chapter discusses the considerations and difficulties that were encountered during the re-engineering as well as giving a description of the reverse engineering methodology itself.

This chapter first highlights the problems facing any re-use of legacy software. This is followed by a discussion of the techniques and features that were known to be needed or were desirable in the re-engineered system and the implications of these features are discussed where they have a bearing on the re-engineering process. The chapter then gives a critical assessment of the implementation techniques and characteristics of the legacy CFD software that was to be used as the basis for the development of a research framework for this investigation.

Finally, this chapter describes the stages used in the reverse engineering methodology. This incremental methodology was specially formulated in order to re-engineer the legacy CFD code and to develop the required prototype CFD application and research framework. A journal paper covering the software re-engineering is included in the appendices [See Appendix 11.2].

The reader might be interested to note that the legacy CFD software system consisted of 107 source files that contained 22,450 Lines-Of-Code (LOC) excluding comments.

3.2 What problems are associated with the re-use of legacy code?

3.2.1 Poor documentation

It is generally the case that research developed codes are largely unsupported by any comprehensive code documentation. The code often serves as its own specification and final

design. Any documentation that is available may be tailored to journals or conference proceedings and is thus unlikely to be concerned with all of the technical development and implementation issues that resulted in a particular instance of the legacy code but rather would be concerned only with the basic algorithm changes that differentiate a particular system from previous approaches.

3.2.2 Evolutionary research code

Generally the priorities for the development of an in-house research code (i.e. not product oriented software) are vastly different from those for commercial software. Often software development companies have adopted strict methodologies for software design, implementation and maintenance. Conversely a purely research based code is likely to have good or excellent mathematical models due to application of new and novel solution techniques. Unfortunately the high quality algorithms are often obscured by a rather poor development style which can lead to monolithic subroutines with too much inline code, poor software module re-use because software grows by modification to pre-existing routines rather than being implemented from design. Research code tends to lack consistent implementation strategy due to the variety of component sources and developers. Further problems occur when there are no strictly imposed and consistent strategies for passing variables, naming conventions or strict software development methodologies which generally result in a working system which is very unclear and exceedingly difficult to extend or maintain. Most research developed codes require considerable tidying and re-structuring before they can be used for commercial systems or for continued research and development by a software development team. A further traditional problem is the nature and experience of most numerical researchers who are generally trained and firmly entrenched in a procedural way of problem solving and software implementation. This is not necessarily the most appropriate or optimal solution to creating a research tool with a long useful life particularly in light of the techniques used for GUI implementation.

3.2.3 Closed and inflexible architecture

The requirements for an interactive and extensible research system are generally very different

from those which are tolerated in a pure research code. The previously enumerated points about data passing mechanisms and procedural structure have much greater importance when the system is to be used for continued research by a multitude of researchers or when access to data is required by other modules and possibly even other co-operative processes.

3.2.4 Archaic implementation language

The traditional language choice that is most commonly used to implement numerical systems in general, and CFD systems in particular, is FORTRAN-77. Whilst FORTRAN is indeed fast and portable it does suffer from being very restrictive particularly as far as data structures are concerned. Furthermore the only conceptual mode of development that is generally supported is for procedural implementations. Attempts at Object Oriented design and implementation using FORTRAN have proved to be possible but these have had very limited acceptance and are often quite unwieldy. Studies [PARSONS94] have shown that ease of maintenance, code clarity and ease of modular implementation and maintenance can be significantly enhanced by the use of Object Oriented development techniques.

3.2.5 Lack of any existing User Interface

Most existing software systems that were developed as research tools are not generally supported by any form of integrated user interface. Partly this is due to the priorities of the researchers which tend to favour algorithm robustness, solution correctness, execution performance and fast implementation rather than any form of interaction techniques. Also many CFD codes started their development at a time when graphical interactive computer terminals were unavailable and hence the CFD software could only be run as black box processes on pre-configured simulations.

3.2.6 Few, if any, library tools

Research development of software tends to design from the top down and does not often concentrate on the generation of library software routines and modules that can be used in any

subsequent research. A well conceived library of software tools can significantly benefit future code development because researchers do not have to implement commonly used routines again and such routines are easy to locate within the software system. The libraries that are available do provide low-level routines such as solvers, norm calculation routines and vector algebra but tend to be over prescriptive of the data structures and, in any case, do not support the real complexity of CFD software which is mostly in the formulation of the coefficient values used in the system matrix, the calculation of auxiliary variables and properties and the interdependencies between coupled variables.

3.2.7 Batch mode of processing

The nature of computers and their development history has led to a situation where many competent (and indeed highly skilled) numerical software developers are unaware of the benefits and possibilities afforded by non-batch mode codes. Batch mode processing tends to dominate the field of CFD research because simulation run times used to be measured in days and weeks and it was thus inconceivable that a user would wish to monitor and interact with a simulation during the entire computational phase. This situation has been improved drastically by the continuously increasing performance gains of current computer systems. Unfortunately the traditional batch mode of processing still holds for most CFD systems when performing the actual numerical simulation. Admittedly there have been advances made to the set-up tools (specification) and post processors (results analysis) but the numerical computational phase remains very similar to the batch mode techniques of legacy software.

3.3 Discussion of CFD techniques used in the legacy code

The particular formulations, approximations and algorithms used in the legacy CFD code (that were subsequently carried forward into the prototype CFD system) are considered to be outside of the scope of this thesis since they concern common numerical formulations of existing algorithm development work that have little bearing on the re-engineering or the imposition of interactive techniques. The interested reader is directed to the appendices section where the CFD technical material and capabilities are described in some detail.

3.4 What considerations have to be made for the re-use of legacy code for use in the new system?

3.4.1 Nature of control and granularity

Since the target for this study was a system which would be able to respond to dynamic solution control there needed to be due consideration to the nature of the control within the legacy system. All of the available aspects of control had to be identified and furthermore each control parameter had to be evaluated so that decisions could be made about "if", "when" and by "how much" could it be safely modified. There were also data dependencies upon control parameters which had to be assessed. For example it would not be correct to modify a relaxation parameter part-way through a solver sweep if this would cause some of the cells to use the old value of relaxation whilst the remainder used the new value of relaxation. Such a situation would introduce potentially unpredictable and unstable solution behaviour and had to be prevented.

3.4.2 Existing looping structure

The existing looping structure within the legacy system was bound to the procedural development style used by most CFD developers in the traditional software development cycle typically found in numerical software developed in a research setting. This was not necessarily appropriate for the target system. The nature of looping required was identified and the implications of changing the looping were also assessed prior to re-engineering. The smallest "chunk" of processing was determined to be the outer "sweep iteration" which causes all of the solved, calculated and auxiliary variables to be updated once.

3.4.3 Existing procedural structure

A comprehensive understanding of the legacy system modularity was ascertained prior to the re-design process. The scope and nature of the procedures had to be determined so that the re-design work would not "break" the algorithms from the legacy code.

3.4.4 Use as part of ongoing research program (involving others)

The fact that the prototype system was to be used as a research framework for CFD techniques had important implications for the use of the legacy software. One such consideration was that the new algorithm syntax and data access mechanisms could not be too "alien" in usage to the intended developers. Another consideration was for the research requirements, within the medium to long term, that would have implications on the form of data structures used and the modularity of the software.

3.4.5 Data structures

The form of the data structures used in the legacy software had to be evaluated so that the functional and data dependencies were known prior to any re-design. It was also necessary to assess the nature and extent of data passing mechanisms. This evaluation ended with an assessment of the most flexible and extensible data structures that could be used in the target system which were still compatible with the algorithms within the legacy software.

3.4.6 Performance issues

CFD code users are highly aware of the overheads of performance because the size and complexity of simulations, together with the computationally intensive numerical CFD processing, lead to extended run-times. When the legacy software was re-designed some consideration had to be given to the performance degradation or improvement that would result from any design changes or implementation differences.

3.4.7 Portability issues

Since there is generally no specific computer hardware that is used to run CFD simulations then, in order to provide adequate user coverage, the software was developed to be as portable as possible. This affected the choice of implementation language and the choice of third party

libraries which could be used. At the outset, the developer was aware that the prototype was intended for use as a framework for future internal research but was also likely to be further developed as a saleable product and thus, it was necessary to consider the commercial aspects of the development too (e.g. minimising the dependencies on external expensive libraries whilst maximising portability).

3.4.8 Integration of GUI components

Graphical User Interfaces are reasonably straightforward in concept but the diverse nature of the underlying operating systems and computer hardware means that there are few truly portable GUI development libraries. Most of the portable GUI libraries have a particular mode of operation that is termed "event driven". There were a number of potential conflicts between an event driven GUI and the legacy procedural software since it was intended that the software is to be fully interactive.

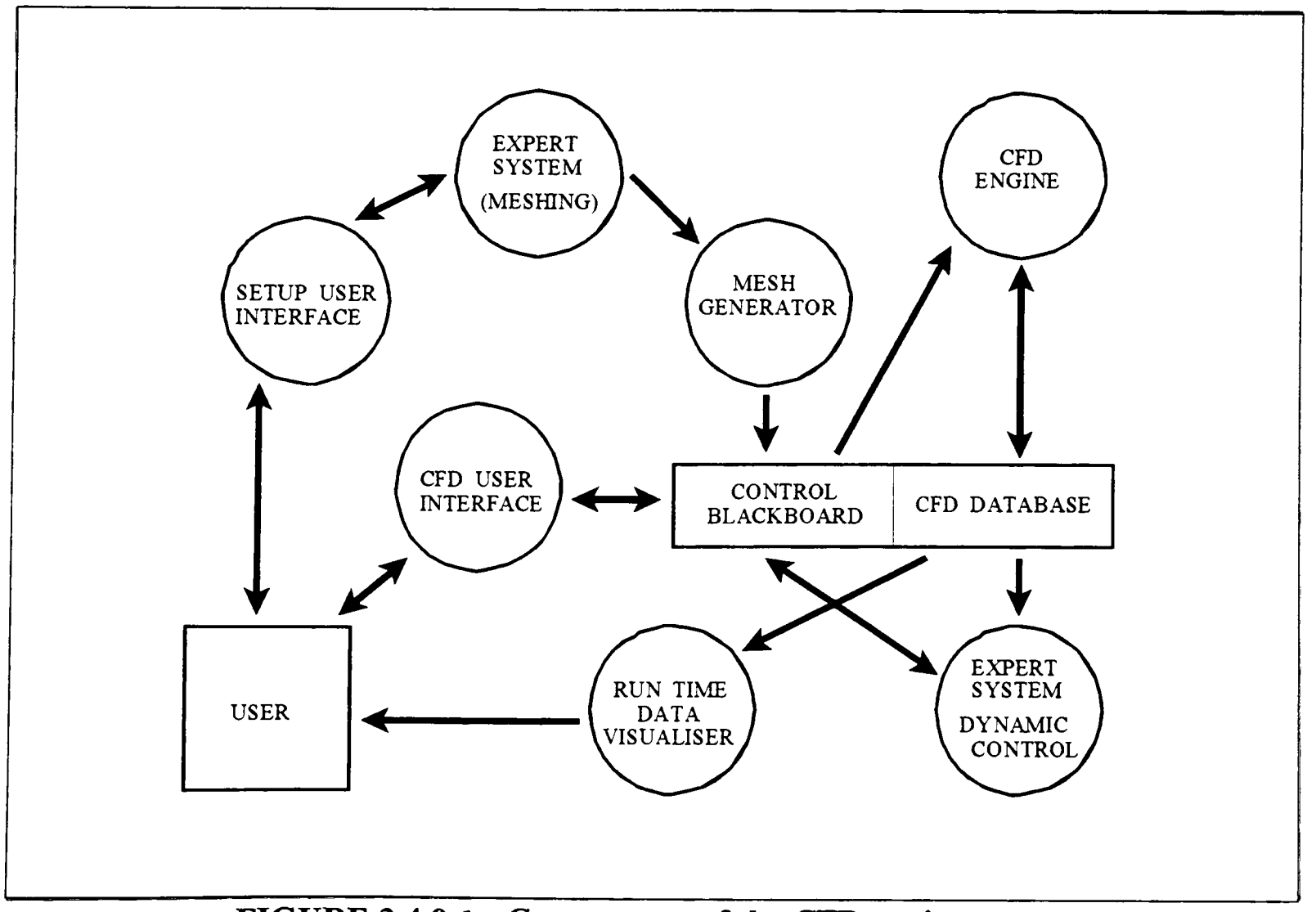


FIGURE 3.4.8-1 : Components of the CFD environment and the data access architecture.

3.4.9 Integration of KBS components

Prior research on FLOWES [PETRIDIS92] [PETRIDIS96] indicated a suitable architecture for the use of Knowledge Based System components that could be used to control the numerical processing using production rules. The communications between the numerical software and the rule system are made possible by the use of a Blackboard. Part of the extended research that used the prototype system was a Knowledge Based System capable of reasoning about appropriate control changes using stored expertise from expert CFD users. It was intended that such changes could be effected by either a knowledgeable user (via the GUI) or the KBS module in order to improve the simulation strategy. For the most part the available KBS tools are unlike traditional programming languages and there were implications as to the choice of implementation language for the CFD system so that the KBS and CFD components could communicate effectively (or indeed at all). There was also a need for evaluation of how and in what form could data from the CFD system be made available to the KBS system and how could control be sent from the KBS to the CFD system (See Figure 3.4.8-1). Such decisions affected the re-design process for the legacy software.

3.4.10 Integration of visualisation components

A significant new development within the CFD system was for the continually updated "run-time" visualisation of data during the computational processing. This implies that the complete solution data for visualisation had to be made available to the visualisation routines whenever the display needed to be updated (such as when a window is exposed or resized or when new data is available). Since the visualisation routines are based on an event driven paradigm it was necessary to provide unrestricted global data access to the visualisation routines.

3.5 What new techniques were needed in the new system and what implications did these requirements have?

The requirements for an open architecture, maintainable CFD system with integrated graphical

interface and dynamic visualisation indicate that Object Oriented design and the event driven paradigm are necessary implementation strategies. These architectural changes are desirable from the point of view of the final delivery system but they have large implications for the approach used and difficulties to overcome in order to re-engineer the legacy software into an appropriate form.

In order to perform the software reverse engineering of the legacy software it was necessary to identify the complete calling architecture, common procedural code and code duplication as well as any "inline" methods. The full design and algorithmic methods used within the legacy software had to be extracted for use in the new system in the most concise and self-consistent form possible to support extended research.

It was also necessary to determine if the legacy software contained any computational objects which could be created within the new system to encapsulate concepts, related data and methods. The creation of abstract types provided significant flexibility and ease of maintenance within the target system. There was also an improvement in general code clarity when objects had intuitive and self-consistent meanings. Often the use of alternative data structures could support lateral research thinking because new ways of problem solving become apparent due to the nature of the objects themselves. The potential problem of using objects was the potentially significant overheads that have been noted in some Object Oriented (OO) implementations [ANGUS91] [DUBOIS-PELERIN93]. It has already been noted that performance is frequently an important consideration when choosing a CFD code and it is vital that the imposition of Object Orientation should not drastically affect performance. Angus et al [ANGUS94] had noticed quite a large overhead for Object Orientation when applied to lower "processing" levels of a "flutter analysis" simulation code. Their approach to overcoming these overheads was to keep the lower levels using simple data structures but to group the simple structures and methods at quite a high level of the calling architecture so that the performance hit of Object Orientation was minimal. Such an approach was considered for the re-engineering of the legacy CFD code but there are a number of potential Object Oriented exploitations that would not be possible with this half-way house re-structuring. Care was taken to avoid the potential problems [WEBSTER] of using an Object Oriented development by investigating the possible problems

before the re-engineering commenced.

Consistency with numerical CFD developers experience necessarily put limits on the implementation language choice and limited the features that were used. The new system was designed to be largely self descriptive and consistent with the known algorithms from the legacy system. It was also a design requirement that data access was self-consistent for all software developers and researchers.

The coding interface between the GUI and numerical CFD code crosses the boundary between an event driven architecture and purely procedural code. The danger with integration of different programming methodologies was that one or other component can tend to dominate with the effect that the GUI could seem unresponsive while the CFD code was always processing or, alternately, the CFD code was never processing or behaving sluggishly as the GUI was always waiting for input. Part of the design strategy that has been demonstrated in prior research is the use of Blackboard objects for inter-component communications and for maintaining GUI defaults.

Since the target system was ultimately intended to have KBS support, there were considerations that had to made during the system design. The first choice was for the implementation language such that data transfer between the CFD code and the KBS component was possible, portable between systems and efficient. If, as seemed likely at that stage, the KBS system was constrained to be a separate process then the overheads and complexity of data transfer by file would have to be assessed. In practice the KBS, GUI and CFD engine are coupled by a global "blackboard" data structure but are, for efficiency reasons, implemented within the same executable. This implementation strategy was deemed to be both appropriate and necessary because of the huge amounts of data required by a CFD simulation. A coupled system gives instant and unrestricted access to all of the simulation data with no communications overheads. A mechanism of restricting control access was considered in order to prevent meaningless or potentially problematic control modifications. There is a distinct possibility that pattern recognition code would be required to summarise status information prior to reasoning although creation of pattern recognition routines is outside of the scope of this research.

Visualisation was deemed to be vital for comprehension, by a CFD user, of patterns and trends in large numerical data sets. The form of data within CFD systems is such that there are three possible forms of visualisation and a variety of visualisation techniques from vector arrow displays to contouring or graphs. The 3-D form of visualisation was considered to be too time consuming and complex for the run-time display of data and was more appropriate to the usual post processing visualisation of results from a completed simulation. A further perceived problem with 3-D visualisation was that the displays can sometimes be uninformative if they are cluttered with too much data simultaneously. Conversely 2-D visualisation is known to be considerably faster and easier to draw but has the disadvantage that only a limited amount of data is displayed at any one time so features could be missed if the 2-D display slice is chosen inappropriately. Graphs are really 1-D visualisations that are very good for displaying time, sweep or distance varying quantities so that data trends and convergence can be ascertained. The 3-D and 2-D visualisation methods require access to the data throughout the computational domain for all variables. The 1-D graphs may also require access to historical data values when used for graphing data against time or iteration sweep.

Consideration was also given to new code algorithms and new application specific modules and techniques. These requirements implied that the new software system had to be coded for clarity and robustness. This would not necessarily be the case for a pure prototype research code. One feature that had already been conceived was for the use of group solvers. It became apparent that there were two techniques that would be rather difficult to implement and use in the legacy system because of the lack of dynamic memory allocation and the use of existing code procedures that are global rather than object-specific. These techniques were for mesh adaption and mesh refinement. It was unlikely that these techniques could be implemented during the current research but consideration was given to the form of data structures and procedure modularity that would support their implementation at a later date.

3.6 Critical evaluation of the legacy code

The following sections give a brief description and assessment of the state of the legacy system

at the time when the legacy software was used for re-engineering. The interested reader is directed to the SMARTFIRE Technical Reference which is included in the Appendices section [See Appendix 11.4]. The Technical Reference covers the numerical and physics modelling capabilities that exist in both the legacy and re-engineered CFD codes.

3.6.1 Coding style

The legacy code was written in standard FORTRAN-77 with no extensions to using compiler specific features (e.g. no use is made of "DO..ENDDO", "DO..WHILE" or aggregate data types) that are available in FORTRAN-90 and as extensions in some FORTRAN-77 compilers. This was good from the point of view of portability but it did mean that "DO..WHILE(..)" loops were actually implemented as "n CONTINUE..IF (..) GOTO n" which can be hard to follow semantically within large sections of code. These are fairly common constructs in the legacy CFD software.

Very little use was made of the computed GOTO or simple GOTO other than for the implementation of DO..WHILE loops as described above. This meant that the legacy system had reasonably clear execution paths, although some sections of the code were very long.

The naming conventions used in the legacy software followed the standard 6 characters for identifiers and procedure names. Almost all variables were explicitly declared for dimensions and type. The legacy code did not use any implicit typing except for a few loop index counters. The fact that implicit typing was not used made the legacy code easier to reverse engineer but the use of 6 character identifiers made the code quite hard to comprehend initially.

Comments were only consistently applied to subroutine headers where they were used to described the argument list variables and their access modes. The remainder of the source code had only a few scattered comments (typically 1 descriptive comment per 50 lines of code) to explain the algorithms or the purpose of subroutines and functions. Generally the only internal comments were for related code block title headings. This lack of comments meant that legacy code developer assistance and some background research was needed to understand the

reasoning behind some of the algorithms and routines.

Reasonable use was made of named integer parameters for declaring the dimensions of data arrays. This was a considerable help in the assessment of implied aggregate types since arrays that were declared as having the same dimensions were thus likely to be of related storage and hence possibly part of an implied aggregate (collection) type. The following example code fragment (See Figure 3.6.1-1) shows some of the data arrays that could be identified as being related due to the parameters which indicate the array sizes.

```
INTEGER     MAXCEL, MAXFPC
PARAMETER ( MAXCEL = 18500,          MAXFPC = 6 )
INTEGER     CELMAT(1:MAXCEL),        ADJFPO(1:MAXCEL,1:MAXFPC)
REAL        CENTRE(1:MAXCEL,1:3),    CELVOL(1:MAXCEL)
REAL        TEMPER(1:MAXCEL),        H(1:MAXCEL)
REAL        OLDH(1:MAXCEL),          OLDT(1:MAXCEL)
REAL        LASTH(1:MAXCEL),         LASTT(1:MAXCEL)
REAL        U(1:MAXCEL),             OLDU(1:MAXCEL)
REAL        V(1:MAXCEL),             OLDV(1:MAXCEL)
REAL        W(1:MAXCEL),             OLDW(1:MAXCEL)
REAL        P(1:MAXCEL),             OLDP(1:MAXCEL)
```

FIGURE 3.6.1-1 : Identification of aggregate data types.

Fairly extensive use was made of named parameters for storing empirical algorithm numerical value constants. This was good because it tended to clarify the algorithms concerned and indicated that the numerical values had some consistent meaning. These parameters tended to be declared locally and were thus defined many times throughout the entire legacy software system.

One area that the legacy code did fall down on was the frequent use of explicit integer indices to refer to variables within arrays. This was particularly true of the arrays which hold status and control information for each of the solved variables. Access was non intuitive when the source code merely referred to item "n" in an array since the developer then had to cross check to see how the number "n" related to known items.

3.6.2 Data access mechanisms

There was no use of dynamic memory allocation (compiler dependant additions) in the legacy code and no pseudo-allocation techniques were used (for example FORTRAN developers often declare and use huge data arrays that are partitioned and passed down to subroutines as separate arguments). This meant that the legacy code often needed to be modified and recompiled for different problem sizes. This can greatly limit portability of the software and generally means that portions of the code had to be made available in source code form for re-compilation. There was also the problem that the legacy software might not use memory in the most efficient way possible for the particular problem being simulated.

Standard FORTRAN array based storage was used for all data with each array named separately as a separate variable. The legacy code did not use so called "f-array" storage techniques as in, say, Phoenics [CHAM] - This technique declares a huge data array at the main program level and passes down "chunks" of the array to sub-procedures. The use of simple array based storage and the naming conventions used in the legacy code mean that, other than the declared sizes, there was no clear indication of how differently named variables and arrays were related to one other. Another problem that was encountered concerns the difficulty of developing additional functionality that requires new variables. Generally speaking, the addition of a new variable, to solve or calculate, will also require a large number of additional support or storage variables. This additional storage may be needed for temporary storage during calculations, for control of the solution, for reporting of status or for the storage of different historical versions of the particular variable. This is a particular problem because the data dependencies in the legacy code are unclear and it is thus difficult to ascertain how many and of what type the required new variables should be.

There is no use made of COMMON for passing data around the system between procedures. This means that all data is passed around the system as formal parameters in argument lists. Given the high degree of data dependency between most of the numerical routines there are considerable numbers of arguments. All variable names had to be limited to 6 characters to avoid exceeding the continuation line limits for some of the argument lists. The greatest problem posed

by this high degree of data coupling, between subroutines, is that code enhancement which adds extra variables can potentially involve extensive modifications to the majority of the source code. This is clearly undesirable because of the large potential for introducing errors. There is also a problem with large argument lists because of the limited type checking provided by FORTRAN that would allow some variables in a large argument list to be muddled in order without error or warning. The only symptom of such a coding error would be the unpredictable behaviour of the affected subroutine. This is a real and distinct danger that is present in the legacy software. The following source code fragment (See Figure 3.6.2-1) indicates the scale of some argument lists to major routines.

```
 CALL MCSOLV( 3,       CELTYP, NSOLVR, VELERR,
@             LVFRAC, NOFINC, NOFTYP, NOCTYP, WALLSS,
@             NOCELL, BANWID, FPATCH, NOFACE, WKSP,
@             VRMETH, SRELAX, TOLVAL, MAXITR, NUMPAT,
@             BPATCH, NOVARS, PTBYPT, P,      NPATFD, SCHEME, SERROR,
@             CENTRE, AREA,   CELVOL, ADJELE, MAXCFA, CELFAC, DELTAT,
@             OLDLVF, TEMPER, VRELAX, OLDT,   SOLERR, VFALST, ENUT,
@             FTOCEN, NUMMAT, PRPEQS, NEQSFD, MAXEQS, MATPRP, SKINFR,
@             NPRPFD, CELMAT, LASTU,  LASTV,  LASTW,  UAPL,   VAPL,
@             WAPL,   UPG,    VPG,    WPG,    BUOY,   SYSMAT, VARERR,
@             MATINX, U,      V,      W,      TMPSYS, UB,     VB,
@             WB,     UAP,    VAP,    WAP,    OLDU,   OLDV,   OLDW,
@             BDARCY, ISOLID, NORMAL, DENSIT, ENUL,   FACPTS, FACTYP,
@             NOPINF, MAXFPT, TURMOD, ADJFPO, DEBUG,  NUMDBG, XPROD,
@             XYZCRD, NUMPTS, ERRINF )
 RELAXA = SRELAX(1)
 RMETHD = VRMETH(1)
 MITERS = MAXITR(1)
 LTEMP  = DEBUG(1)
 CALL PCSOLV( 3,       CELTYP, LVFRAC,
@             NOFINC, NOFTYP, NOCTYP, NOCELL,
@             BANWID, NOFACE, WKSP,   RMETHD,
@             RELAXA, TOLVAL, MITERS, NUMPAT, BPATCH,
@             SCHEME, LASTP,  B,      CENTRE, AREA,   CELVOL, BUOY,
@             ADJELE, NPATFD, MAXCFA, CELFAC, DELTAT, PCORR,  OLDLVF,
@             TEMPER, OLDP,   OLDT,   NUMMAT, PRPEQS, FTOCEN, ADJFPO,
@             NEQSFD, MAXEQS, MATPRP, NPRPFD, CELMAT, U,      V,
@             W,      UAP,    VAP,    WAP,    UPG,    VPG,    WPG,
@             SYSMAT, MATINX, RESIDU, NORMAL, DENSIT, FACPTS, FACTYP,
@             NOPINF, MAXFPT, LTEMP,  LASTU,  LASTV,  LASTW,  ERRINF )
 IF ( ERRINF .NE. 0 ) STOP
 SERROR(1) = RESIDU
```

FIGURE 3.6.2-1 : Usage of formal function parameters.

3.6.3 Structure

Many of the routines are basically copies of other routines with only slight algorithmic modifications. There is generally little or no consideration given to the isolation and re-use of common code. This is particularly true of the large subroutines that are used to build the coefficient "system" matrix for each variable or the fairly common methods such as simple vector geometry operations. Whilst this approach tends to make the source code very large, and more difficult to maintain, it does have the benefit of fairly optimal speed of execution since there is generally less decision branching and the number of layers of procedure calls is kept to a bare minimum.

```
Legacy code                        is problematic because

RELAXA = SRELAX(5)                 Literal values and simple assignments
RMETHD = VRMETH(5)                 prior to calling a complex numerical
MITERS = MAXITR(5)                 calculation routine.
FALSET = VFALST(5)
CALL HCSOLV(3, RELAXA, ... )       Highly abstracted routine call.
CALL SYSRES( ... )                 Less abstracted utility routine call.
SOLERR(5) = RESIDU
RELAXA = VRELAX(5)
CALL LINRLX( ... )                 Utility routine call.
VARERR(5) = RESIDU
RELAXA = VRELAX(8)                 More low level simple assignments.
RMETHD = VRMETH(8)
CALL CSOLVT( ... )                 Call to highly abstract routine.
IF ( ERRINF .EQ. 0 ) STOP
VARERR(8) = RESIDU
```

FIGURE 3.6.3-1 : Problematic code in the legacy software.

The subroutines in the legacy code have a great deal of clutter around them as if the structure and level of code abstraction has not been completely decided. In the main program there are calls to subroutines that are surrounded by simple assignment statements. This leads to a code that does not have a clear semantic consistency because there are mixed levels of code abstraction. This is clearly evident when one considers the major solution routines, within the top level procedure, that are embedded within simple assignments and calls to simple utility routines. This is mostly due to the evolutionary (research oriented) style of development which tends to incrementally add and modify existing code rather than to be based on a clear and

distinct top-down design. The code fragment shown in the figure (See Figure 3.6.3-1) indicates some of the problems that were found due to the mixed levels of code abstraction in the legacy CFD code. N.B. The "..." is used to indicate many formal procedure arguments. It should be noted that, from the literal values, used to index the control and status arrays in the figure, it is possible to infer that "5" represents the solved variable "ENTHALPY" and that "8" represents the calculated variable "TEMPERATURE" but the meaning of the "3" (used as an argument to the HCSOLV routine) is not apparent at this level.

3.6.4 Optimisation

The legacy system often used very large subroutines with a high degree of inline code and code duplication between many routines. As previously mentioned this can lead to near optimal execution speed at the expense of code clarity, code re-usability and ease of adaptive and perfective maintenance.

3.6.5 Control looping

The subroutines are generally based on looping for all things of a particular type. A typical example is the, geometry related, volume calculation routine which calculates the volumes of all cells before returning. This is true of most of the geometry routines and the majority of the solution calculation routines. The only exceptions are some of the lower level source contribution routines for the calculation of the system matrix coefficients which tend to perform calculations for a single control volume face only. Again this approach does give near-optimal performance but limits the flexibility of the system as a research tool since there are few utility routines that can be used in isolation for an individual object. The usual argument for optimal behaviour in the legacy system is not particularly valid for the geometry routines since they are only currently used during system initialisation. The following code fragment (See Figure 3.6.5-1) shows the implementation of looping for the legacy volume calculation routine that is only able to calculate volumes for all cells at once.

```
  SUBROUTINE VOLUME ( NOCELL, NOFACE, NUMPTS, DIMENS, XYZCRD,
 @                    CELPTS, CELTYP, NOCTYP, NPTCTY, MAXCPT,
 @                    FACPTS, MAXFPT, NOFTYP, FACTYP, NOPINF,
 @                    CELFAC, MAXCFA, NOFINC, CELVOL, CENTRE,
 @                    AREA,   NORMAL, DEBUG,  ADJFPO, ADJELE,
 @                    ERRINF )
C
C Many lines of declarations removed
C
  REAL CELVOL(1:NOCELL), XYZCRD(1:NUMPTS, 1:DIMENS)
C
C Many lines of initialisation removed for clarity
C
  DO 1 I = 1, NOCELL
     CELVOL(I) = 0.0
     DO 2 H = 1, NOFINC(CELTYP(I),0)
C
C Many lines of volume computation removed for clarity
C
        CELVOL(I) = CELVOL(I) + AREA(FACNUM) * DISTAN / 3.0
2    CONTINUE
1 CONTINUE
  RETURN
  END
```

FIGURE 3.6.5-1 : Control loops to be found in the legacy software.

3.6.6 Consistency

Array arguments outside and inside of called routines generally, but not invariably, use the same identifiers but there are many instances of subroutines having arguments that contain a differently declared number of dimensions than in the calling routine. This is a serious flaw in the legacy code since a developer assessing a piece of code in isolation will find it necessary to trace the variable back up through the calling structure to determine the exact nature, context and access mechanism of the variable in question. Also the calling and called naming convention consistency is not guaranteed, within the legacy code, which can lead to semantic development errors caused by name changes. This is a considerable problem as the code is intended to be used and extended by a number of developers who will only have access to the source code itself. The following source code fragment (See Figure 3.6.6-1) demonstrates the change in names and dimensions of some variables between calling and called subroutines.

```
      INTEGER     SCHEME,      NPATFD
      REAL        WALLSS(1:NOFACE),  SYSMAT(1:BANWID, 1:NOCELL)
      REAL        LASTU(1:NOCELL),   LASTV(1:NOCELL)
      REAL        LASTW(1:NOCELL),   P(1:NOCELL)
      REAL        UB(1:NOCELL),      VB(1:NOCELL)
      REAL        WB(1:NOCELL),      ENUL(1:NOCELL)
      REAL        ENUT(1:NOCELL),    SKINFR(1:NOFACE)
      REAL        NORMAL(1:DIMENS, 1:NOCELL, 1:MAXCFA)
      LOGICAL     TURMOD
C
C Declarations removed for clarity
C
      DO 1 ELENUM = 1, NOCELL
C
C Numerical code removed for clarity
C
         DO 2 FACNUM = 1, NOFINC(CELTYP(ELENUM),0)
C
C Numerical code removed for clarity
C
            IF ( ADJNOD .LT. 0 ) THEN
C-----------------------------------------------------------------------
C External Boundary
C-----------------------------------------------------------------------
               PATCH = FPATCH(FACE)
               CALL CBOUND ( PATCH,  NUMPAT,  FAREA,  BPATCH, DIST ,
     @                       NORMAL(1,ELENUM,FACNUM), NPATFD,
     @                       ENUL(ELENUM),            ELDENS,
     @                       ENUT(ELENUM),            SCHEME,
     @                       LASTU(ELENUM),           LASTV(ELENUM),
     @                       LASTW(ELENUM),           SKINFR(FACE),
     @                       WALLSS(FACE),            TURMOD,
     @                       P(ELENUM),
     @                       UB(ELENUM),              VB(ELENUM),
     @                       WB(ELENUM),              SYSMAT(1,ELENUM) )



      SUBROUTINE CBOUND ( PATCH,  NUMPAT, AREA,   BPATCH, DIST,
     @                    NORMAL, NPATFD, ENUL,   DENSIT, ENUT,
     @                    SCHEME, U,      V,      W,      SKINFR,
     @                    WALLSS, P,      TURMOD, UB,     VB,
     @                    WB,     SYSMAT    )

      INTEGER     PATCH,  NUMPAT, NPATFD, SCHEME
      REAL        AREA,   ENUL,   DENSIT, ENUT,   SYSMAT, UB
      REAL        VB,     WB,     U,      V,      W,      SKINFR
      REAL        WALLSS, P
      REAL        NORMAL(1:3),    BPATCH(1:NPATFD,1:NUMPAT)
```

FIGURE 3.6.6-1 : Lack of consistency through parameter lists.

It is worth noting that the "NORMAL" array changes from a 3-D array with sizes of

(1:DIMENSIONS, 1:NOCELL, 1:MAXFINC) to a 1-D array (1:DIMENSIONS) between the calling and called routines. Conversely the "LASTU" array is used as an array called "U" in the subroutine but there is already a "U" of different meaning used in other areas of the code. These changes are likely to lead to confusion and incorrect data access.

The legacy system also exhibits some behaviour that shows that there were a number of code developers working at different times on the system. This is most clearly indicated by a general lack of consistency between some of the routines in terms of naming conventions or structure and purpose. The evolutionary style of development has not helped since there have been no strict development guidelines to adhere to. The only criteria for development has been for the meeting firstly functionality and secondly performance requirements with little or no emphasis on style or maintenance.

3.6.7 Code clarity

Given the lack of comments and the FORTRAN-77 standard restrictions for naming conventions it can be very difficult to follow the source numerical algorithms. This is not helped by the high level of complexity of the algorithms and data structure inter-dependencies due to the unstructured nature of the solution mesh.

The code is mostly unsupported by any comprehensive documentation or algorithm designs and thus serves mostly as its own completed specification.

3.7 Development of a novel nine stage incremental re-engineering methodology

The re-engineering strategy, developed during this investigation, used a nine stage incremental process to restructure the legacy code in FORTRAN-77, to translate to C++, to enforce modern software engineering design principles and to prepare for later perfective and adaptive maintenance. Much of this research has been published in a case study in a journal publication and the interested reader is directed to read the paper [EWER95], included in the Appendices [See Appendix 11.2] for a more complete discussion of the re-engineering process. The flow

diagram (See Figure 3.7-1) indicates the key stages in the re-engineering of the legacy CFD code. The central vertical line indicates the boundary between the FORTRAN-77 and C++ implementations.

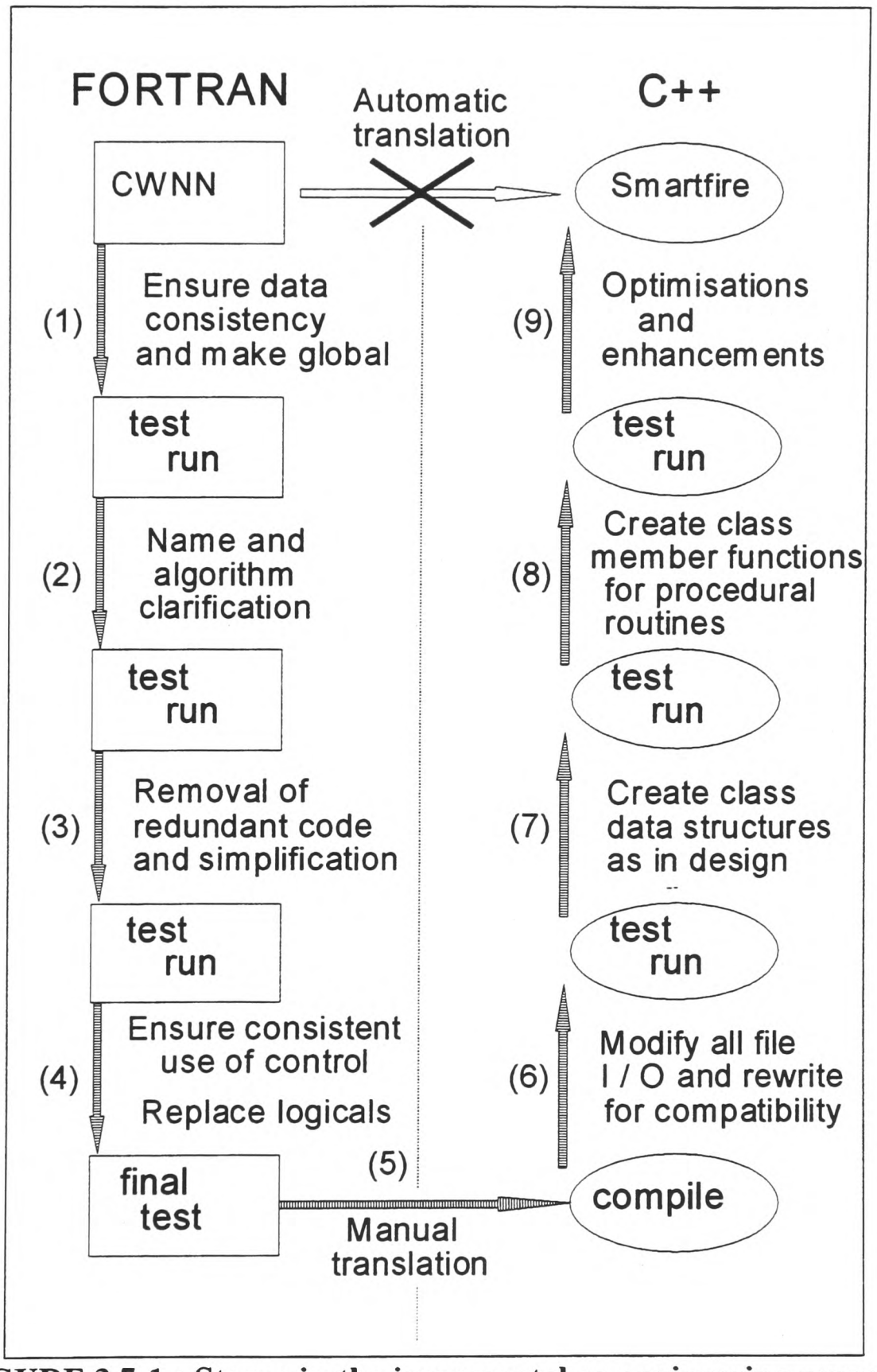


FIGURE 3.7-1 : Stages in the incremental re-engineering process.

Comprehensive regression testing (using a numerical file comparison utility called "Numdiff" [EWER93-2] to check results consistency) after each incremental stage of the re-engineering process ensured functional compatibility between the delivery system and the legacy code. This

regression testing used a sufficiently diverse selection of test cases to ensure that all major code modules were exercised. Clearly it would be inadequate just to test a small sub-section of the code and leave major portions untested.

3.7.1 Stage (1): Ensure data consistency and make all data global

Initially, all of the separate source code files were combined into a single source file with the main program routine as the first routine. This was necessary because the editor, used throughout the re-engineering, had a limit on the allowed number of open files. There was also the problem that a large number of files would lead to a much greater likelihood of missing a modification or translation step. Subsequently, tools such as Visual Studio, with integrated multiple file searching and hypertext browsing, have reduced the problems associated with maintaining and developing multiple file large applications. The legacy software used approximately one source file per sub-routine and navigation was hampered by the use of DOS standard 8.3 file naming and short (6 character) procedure names.

A FORTRAN-77 source code analysis and restructuring tool called SPAG [SPAG93] was initially used to tidy the indentation and to restructure the source code using consistent control constructs. The SPAG tool set also has a global code check utility that was used to generate much needed information about subroutine calling structure and variable usage. SPAG was also configured to set the case of identifiers to indicate variable scope and usage. COMMON variables and PARAMETER statements were completely capitalised whilst local variables used only lower case. Subroutine arguments had an initial capital letter followed by lower case characters. This helped somewhat to locate the appropriate declarations and showed the dependencies of any variable.

```
Legacy FORTRAN code

CALL BUOYAN( RMETHD, ELEMAT, ELEVOL,   TEMPER, .. )

SUBROUTINE BUOYAN( RMETHD, ELEMAT, VOLUME, T, .. )
INTEGER ELEMAT(TOTELE)
REAL    VOLUME(TOTELE), T(TOTELE)
B = TEMPER(I) * ...

is modified to become

INCLUDE 'DATABASE.INC'
CALL BUOYAN( RMETHD, .. )

SUBROUTINE BUOYAN( RMETHD, .. )
INCLUDE 'DATABASE.INC'
B = T(I) * ...

with DATABASE.INC defined as

INTEGER ELEMAT(TOTELE)
REAL ELEVOL(TOTELE), TEMPER(TOTELE)
COMMON /CELL_D/ ELEMAT, ELEVOL, TEMPER
```

FIGURE 3.7.1-1 : Passing data via include files and COMMON.

In order to restructure the software into an object oriented form, the data was grouped into class-like "COMMON" structures. This could not be done if data items changed names in argument lists or were passed around as incomplete array segments. It was therefore necessary to match calling and called routine arguments and rename local variables to match external data items. SPAG was used extensively to document and navigate within the CFD code.

```
Legacy FORTRAN                      becomes

CALL HBOUND( H(ICELL), .. )         CALL HBOUND( ICELL, .. )

SUBROUTINE HBOUND( HVAL, .. )       SUBROUTINE HBOUND( ICELL, .. )
REAL HVAL                           INCLUDE 'DATABASE.INC'
HVAL = HVAL * ...                   INTEGER ICELL
                                    H(ICELL) = H(ICELL) * ...
```

FIGURE 3.7.1-2 : Revised argument passing for COMMON data.

Non-standard FORTRAN include files were used to pass data between routines. These include files used COMMON data declarations to ensure that only one declaration exists for each variable. Any easily identifiable utility routines kept their parameter list arguments intact. In

some instances, during re-engineering, COMMON was used inappropriately for passing data to utility routines. This problem was quite easy to identify because of the necessary introduction of many simple assignment statements (putting data into COMMON) just before the utility routine call. The figure (See Figure 3.7.1-1) demonstrates the passing of data in COMMON blocks.

Data items were grouped into appropriately named COMMON blocks with related items, as they were identified. This identification was facilitated mostly by the declared dimensions of the arrays and the subroutine header information. Tentative groupings were made based on the declared array sizes and these were revised as actual array variable usage was completely identified within the source code. For example arrays with dimensions of (1..NOCELL) indicate cell properties of some sort whilst those dimensioned as (1..NOFACE) were face properties. Many of the single variables were identified as being suitable for COMMON by simple inspection of their usage. Switch control variables tended to be more appropriately passed as arguments to routines. It was preferable to err on the side of caution because of the slight potential for naming conflicts between new COMMON variables and dummy arguments.

Dummy argument names were replaced with direct accesses to the newly defined included COMMON variables. Where subroutines were receiving arguments which were single array elements it was necessary to ensure that the appropriate array index was available within the subroutine. The code fragment (See Figure 3.7.1-2) indicates how array index values were passed instead of the array elements.

3.7.2 Stage (2): Name and algorithm clarification

The FORTRAN-77 standard 6 character identifiers and subroutine names were replaced with longer, lower-case, names that conveyed the functional or conceptual meaning and usage. SPAG was used to automatically rename identifiers, since it prevents and reports any renaming conflicts. Some of the initial name changes are detailed in the figure (See Figure 3.7.2-1).

```
Legacy FORTRAN identifier  -> New identifier name

MCSOLV                     -> solve_momentum
CALGEN                     -> calc_generation_rate
RDINFF                     -> read_inform_file
H                          -> enthalpy
TEMPER                     -> temperature
U                          -> u_velocity
KINETC                     -> kinetic_energy
DISSIP                     -> dissipation_rate
```

FIGURE 3.7.2-1 : Name changes for code clarification.

Many, formerly inline, code sections were moved into new subroutines to highlight their algorithmic meaning at a more appropriate level of abstraction. Passing data by include file and COMMON facilitated this process since extensive re-declarations were no longer necessary. There were two ways to identify inline code. The first was recognition of those instances of code that keep appearing relatively unchanged throughout. An example of repetitive inline code within CWNN was for the calculation of the cell upwind density. This code consists of some 32 lines of source code duplicated in 10 different calculation routines. The second sort of inline code was the use of very large code fragments (100 lines or more) in control constructs such as IF (..) THEN...ENDIF blocks or DO...CONTINUE loops.

```
Legacy FORTRAN indicates that pressure should use the SOR solver

SOLTYP(1) = 5

becomes

INCLUDE 'PARAMS.INC'
solver_type( PRESSURE ) = SOR

with PARAMS.INC defined as

INTEGER PRESSURE, SOR
PARAMETER( PRESSURE = 1, SOR = 5 )
```

FIGURE 3.7.2-2 : Introduction of PARAMETER constants.

Literal numbers that were used to index arrays or used in calculations were globally defined as more meaningful PARAMETER statements in an include file. This file was then included in all routines as indicated in the example code fragment (See Figure 3.7.2-2).

3.7.3 Stage (3): Removal of redundant code and simplification

Code paths and variables that were not required for the current project were removed from the system. It was noted that solidification modelling was not necessary for the target application area, so the corresponding code was completely removed. The solidification code was simple to remove because it was all switched via logical control variables. The extra variable solver also presented no difficulty to removal because it was (like most of the other solvers) simply a copy of an existing solver routine with the data variables changed.

```
Legacy code fragment

MITERS = MAXITR(4)
RELAXA = VRELAX(4)
CALL SORSCH( ... )
SERROR(4) = RESIDU
CALL LINRLX( ... )
VARERR(4) = RESIDU

Equivalent code abstracted

INTEGER VAR_W_VELOCITY
PARAMETER( VAR_W_VELOCITY = 4 )

CALL SORSCH( VAR_W_VELOCITY, ... )
CALL LINRLX( VAR_W_VELOCITY, ... )

N.B. The simple assignment statements have been moved down into the
called subroutines.
```

FIGURE 3.7.3-1 : Re-locating simple assignment statements.

There were many instances where code fragments could be simplified by moving simple executable statements (generally simple assignments) into nearby called routines. This helped to keep the code at the same level of algorithmic complexity and avoided unnecessary clutter as shown in the example code (See Figure 3.7.3-1).

CWNN had many research "hooks" for future use. For example, dummy routine calls and logical variables were provided to allow for the possible future development of mesh-refinement and mesh-adaption. These "hook" locations were noted for location and function and then removed to simplify the overall re-engineering process. Some of these hooks have subsequently been added back into the software for research using SMARTFIRE.

3.7.4 Stage (4): Ensure consistent use of control and logicals

Labelled lines were made to use CONTINUE rather than have executable statements. This helped with the translation to C++ and made it easier to find other loop constructs.

Instances of single line "IF (<expression>) <statement>" were replaced with the equivalent form using "IF (<expression>) THEN <statement> ENDIF" so that subsequent translation to C++ would be facilitated. Instances of "IF (<expression>) GOTO <label>" were left unchanged because these were often part of "do...while" constructs. This identification and replacement was performed later when some of the other clutter was removed.

Loop constructs which used the standard "DO <label> <block> <label> CONTINUE" were changed to a non-standard form as "DO <block> ENDDO" loops which avoided excessive use of continue and labels. The use of "DO...ENDDO" loops also allowed easier recognition of the other uses of "<label> CONTINUE" as in FORTRAN simulated "do...while" loops. Any clearly identifiable "do...while" loops were implemented with the compiler specific non-standard FORTRAN WHILE constructs instead of the usual "IF (<expression>) GOTO <start_label>" as used in the legacy code. SPAG was useful in this respect because it has some automatic re-structuring capabilities supporting non-standard, but widely used, control constructs. The correct indentation of these non-standard FORTRAN extensions is vital for conveying looping structure at a glance. SPAG correctly indents loops and branches during its parsing.

Since C++ does not support a built-in LOGICAL type it was decided that an equivalent, robust replacement should be implemented in the FORTRAN code at this stage. The direct translation to a C++ enumerated type was considered but, because there was no conformal mapping for assignment using the NOT value of a logical, the idea was discarded. LOGICAL variables and comparisons were replaced with integers and integer comparisons respectively. The complexity of replacing the LOGICAL values was significantly reduced by working within the FORTRAN version of the code. This also prevented errors in logic that could occur when too many translation steps had to be performed simultaneously. The replacement of a LOGICAL

sometimes required the introduction of "IF (<expression>)" constructs to assign appropriate "Boolean" values to integer variables. The integer parameters "False" and "True" (representing 0 and 1 respectively) were used throughout the code to match the ultimate C++ implementation.

3.7.5 Stage (5): Translate the legacy FORTRAN to procedural C++

When the above stages had been completed, and the FORTRAN code was still producing consistent simulation results, it was necessary to translate the FORTRAN to procedural C++. This was because there was no appreciable advantage to be gained by further FORTRAN code changes. The serious limitations of the available FORTRAN-to-C translators "f2c" [F2C93] and "for-C" [COBALT93] led to the decision to translate the CFD code to procedural C++ manually. The natural course of action would be to use parsing or compiler writing tools such as "lex" [KERNIGAN88-1] and "yacc" [KERNIGAN88-2] but because of the high learning overheads and non-interactive nature of these utilities, an alternate approach was investigated and ultimately used when it proved to be workable. The tool actually chosen was a powerful programmer's editor with regular expression search and replace facilities, macro record and playback and multiple-file editing capabilities. It should be pointed out that a simple text editor would not be sufficient because of the large syntactic variation that may be encountered in the source code during translation. Even using the facilities provided, great care was needed to plan and perform the macros used to translate the code.

Using the editor facilities, the translation to procedural C++ involved writing a set of individual macros to replace specific code constructs. Again SPAG was used prior to this task so that a globally consistent style and control syntax would persist throughout the source code. This was necessary to enable the searches within the macro replacements to work correctly. (See Figure 3.7.5-1) indicates some of the macro text replacements that were used during the manual translation from FORTRAN-77 to procedural C++. The use of regular expression searches and macro replacements does require that care was taken to perform the replacements in order of most complex to least complex to prevent incorrect matching with parts of other expressions. The main problems are with DO, END and IF which can be part of other keywords like END IF or END DO. There are also potential problems with accidentally matching search expressions

with literal strings or parts of variable names. Using case sensitive searches, after SPAG had been used to consistently set the case of keywords and identifiers, minimised the potential for problems. Clearly these problems would not be present using compiler writing tools (e.g. yacc or lex) because all identifiers are recognised as whole tokens. The program editor was useful in one respect because the regular expression text replacements are interactively controlled. N.B. The ".." and "..." represent code and formal arguments, respectively, not changed by the macros.

```
Legacy FORTRAN        ->  Macro replacement code

ELSEIF (..) THEN      ->  } else if (..){
IF (..) THEN          ->  if (..){
ELSE                  ->  } else {
ENDIF                 ->  }
CALL                  ->  /* CALL REMOVED */
DO I = a, b, c        ->  for ( I=a;I>=min(a,b)&&I<=max(a,b);I+=c ){
DO I = a, b           ->  for ( I=a;I<=b;I++ ){
ENDDO                 ->  }
SUBROUTINE .. (...)   ->  void .. (...){
END                   ->  }
RETURN                ->  return;
PRINT*, ...           ->  cout << ... << endl;
nnn CONTINUE          ->  Label_nnn:
GOTO nnn              ->  goto Label_nnn;
```

FIGURE 3.7.5-1 : Macro replacements.

String variables (i.e. FORTRAN-77 CHARACTER*(n)) were dealt with on an individual basis since the numerical code only had a very limited number of routines which manipulated strings. Literal strings were easily replaced by the ["] delimited versions of C++.

One of the major problems encountered during the translation was the difference in array indexing syntax. FORTRAN style array indexing is very different to C++ style array indexing. It was decided to effect these changes manually (using searches and macros) on a variable by variable basis. Macros were used to change the ( ) indices to [ ] indices, but these could not be used globally because of the complexities of multiple-dimensioned arrays, partial array argument passing and arrays that are used to index other arrays. It was decided to increase the declared array dimension sizes by one, and waste the 0th element, because of the declaration syntax and

limitations of C++. This has an adverse effect on memory used but allows quicker run-time performance. Fortunately most of the FORTRAN arrays had (1:n) indices, but (-m:n) or (m:n) indices were represented by simply adding or subtracting a suitable constant at the declaration and each reference. All of the arrays were initially translated to statically declared (fixed size) C++ arrays and no attempt was made to create class structures at this stage. The potential problems of passing segments of multiple-dimensioned arrays were largely avoided because of the earlier consistency modifications made to the legacy code in stage (1) which meant that data arrays were uniquely defined in COMMON storage.

CWNN [CROFT98] has the traditional batch-mode "INPUT -> PROCESS -> OUTPUT" execution path that is common to most legacy numerical simulation codes and hence, it was not necessary for to completely re-engineer the output routines for the initial compile and run testing. Simple use was made of "cout <<" or "printf()" as appropriate whilst leaving the original commented-out FORTRAN PRINT and WRITE statements for later reference and more thorough and complete translation.

Single item input presented no problem using "fscanf()" or "cin >>" as appropriate. List directed and formatted FORTRAN input were more problematic. The approach adopted was to replace a list directed FORTRAN input with a collection of single item or looped-over fscanf() calls. It was necessary to remove any end-of-line comments from the input files until a permanent "in-code" solution could be implemented. Once again IO translation was facilitated by the limited amount of actual IO performed by the legacy software.

After several trial compilations and minor fixes a clean compilation was obtained. Very small data sets (with between 2 and 100 control volumes) and a C++ debugger were used to check that input files were accurately read in and that the data was appropriately stored in the new structures.

3.7.6 Stage (6): Modify all file I/O and rewrite for compatibility

The first task within the procedural C++ code was to ensure that all of the file input and output

was being performed correctly. This was very easy to check by immediately dumping any item read out to a log file and comparing this with the original data file. It was often necessary to use the "ifstream getline()" function to clear to the end of input lines because of the character based file handling of C++ as opposed to the line based file handling of FORTRAN.

One of the input problem-specification files made use of a script-like command language that presented some difficulties because of potentially multiple command arguments. These problems were overcome by implementing a line parsing routine and corresponding token or phrase extraction utilities to interpret the lines of input and to extract data values as required. This was the only area where new code had to be developed due to the differences between the FORTRAN and C++ languages.

Generally most numerical reading of data was as simple in C++ as it was in FORTRAN however care had to be taken with one of the more obscure FORTRAN formats where numbers can be written with no space separators between them when sign characters are used. If, as with the FORTRAN implementation, the format was known then this difficulty can be overcome by reading the required field width into a character buffer and then extracting the value from the buffer. One other difficulty was with PC C++ compilers which tend to always write float values in double precision exponential format. A writing routine had to be developed for IO to those files which required a pure single precision or non space delimited format for use in other packages.

3.7.7 Stage (7): Implement class objects to replace array structures

The original array structures of the FORTRAN code were very unsatisfactory because there was no explicit grouping and no obvious relationship between many of the variables, apart from the nature of the indexing. The groupings used to make COMMON variables in the legacy code provided a means of collecting data items into structures (C++ classes). This allowed the creation of physically meaningful entities with known attributes. The diagram (See Figure 3.7.7-1) shows some of the legacy code FORTRAN-77 arrays.

This corresponds to the diagram (See Figure 3.7.7-2) of the equivalent C++ classes. Some of the arrays in the legacy CFD code (See Figure 3.7.7-1) contain actual data values (e.g. P, OLDP, AREA and XYZCRD) whilst others contain index values (e.g. CELFAC and FACPTS) that are used to reference data items in other arrays. It should be noted, at this point, that the links between the identified class objects could have been implemented using pointers and arrays of pointers instead of integer indices and arrays of indices. The major problem with using pointers extensively is that this would necessarily introduce pointer de-referencing to access values. This would almost certainly be unfamiliar to many numerical CFD developers. It was decided that the object links be implemented in a form not too dissimilar from the legacy code. Whilst this was often less elegant, than some other techniques, it did have the benefits of consistency and ease of implementation.

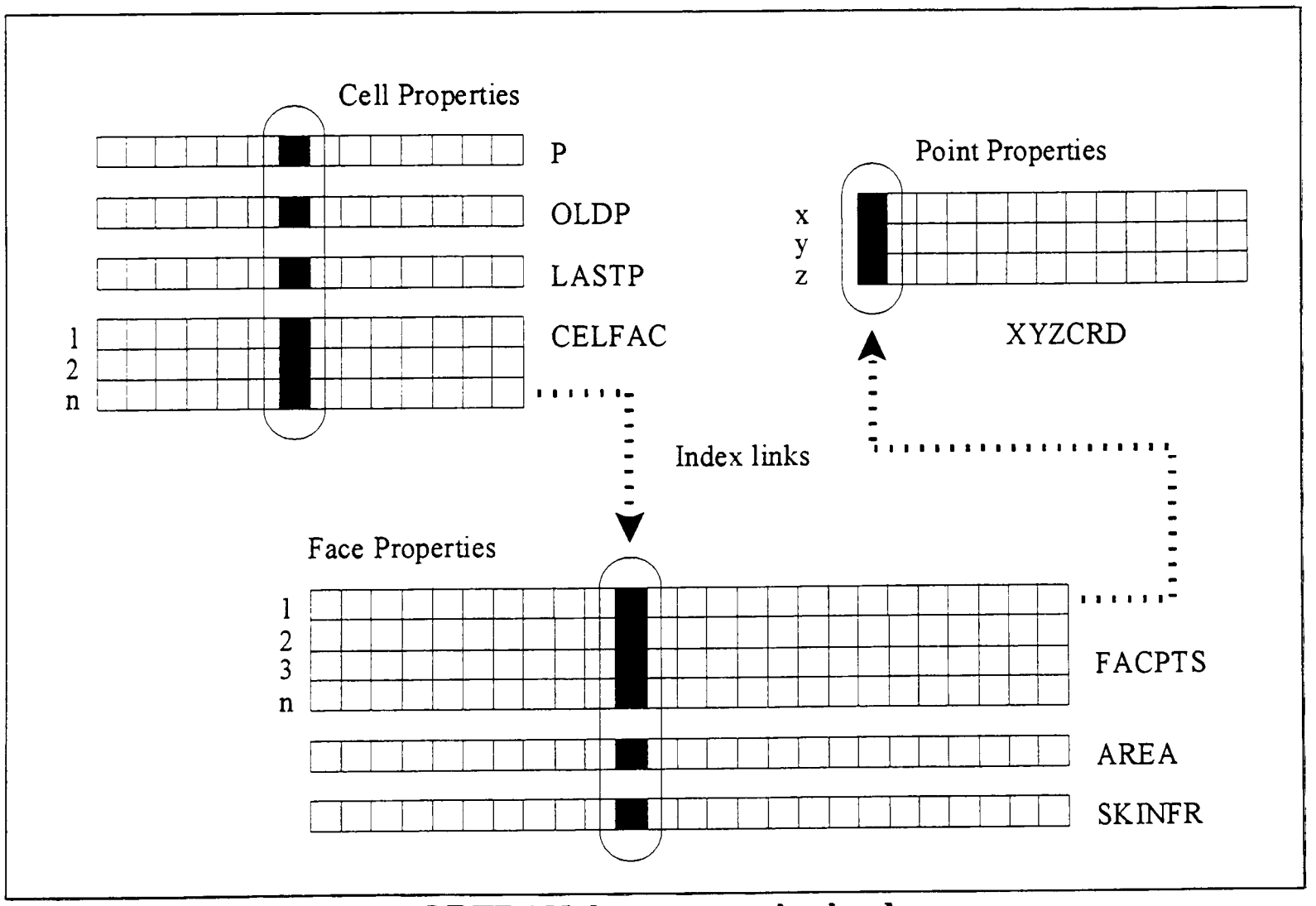


FIGURE 3.7.7-1 : Legacy FORTRAN data storage in simple arrays.

There was a potential problem for the storage of cell properties because of the need to maintain up to four different historical versions of some variables. For example a transient flow simulation needs old time-step, last sweep, previous inner iteration and newest values of pressure and

momentum components. Also variable usage is determined by simulation type. Using explicitly declared cell class attributes for cell properties (e.g. "cell.pressure", "cell.old_pressure", "cell.last_pressure", and "cell.previous_pressure") would always use storage regardless of the actual requirements of simulation. It was decided that simple "data" arrays of properties should be created and then indexed by parameter type identifiers (See Figure 3.7.7-2). The "slots", in the data array, can then be assigned as required by the simulation. This was particularly important, for example, in a simple heat-transfer simulation where the overhead of storing the other flow variables is highly undesirable. This approach also allowed for functional data access with expressive selection arguments, and ease of data monitoring. The argument against using such a method revolves around the fact that it is then possible to confuse the indexing to the slots. In order to prevent inappropriate access to the slots, the indexing mechanisms were made private to the class and embedded within data access functions.

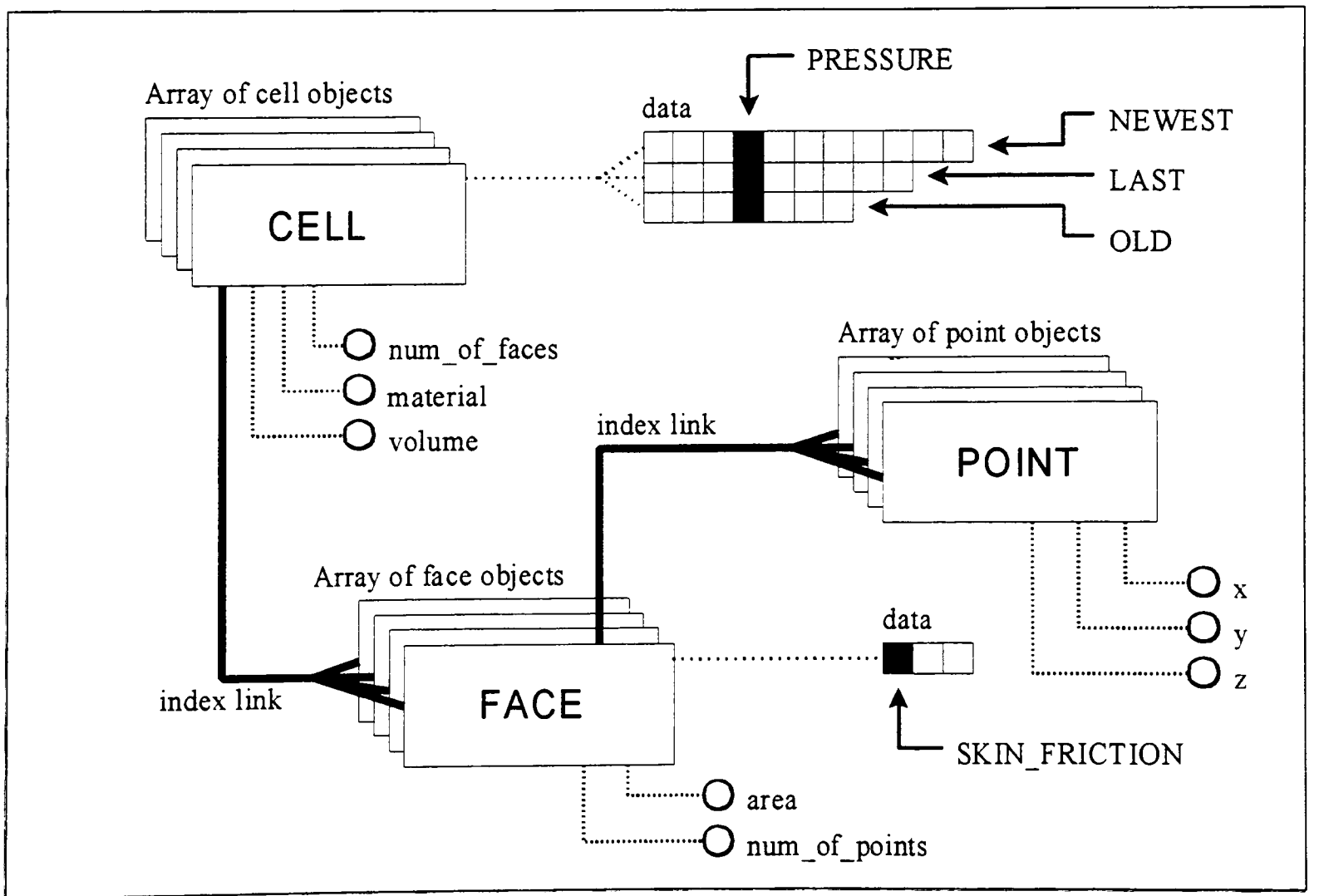


FIGURE 3.7.7-2 : Re-engineered Object Oriented data structures in C++.

3.7.8 Stage (8): Create Class member functions for procedural routines

Many of the procedural calculation routines contained code similar to the fragment in (See Figure 3.7.8-1) which calculates the cell volumes for all cells. The inline code (looping over all cells) often forms a natural class member function for the individual loop objects. The loop code can be abstracted into class utility methods as indicated. The advantage of this approach was that utility routines were created that provided much greater flexibility than was afforded by the original software architecture. Previously the software could only calculate for all objects at once whereas it would be desirable (as in the case of mesh adaption and mesh refinement) to limit the calculation to selected objects.

```
Legacy style routine for all cells

void calculate_all_volumes( void ){
   for ( i = 1; i <= max_cells; i++ ){
      // calculate volume for current cell
      cell[i].volume = ...;
   }
}

Introduce a new method for calculating the volume in a cell

void Cell_Class::calc_volume( void ){
// calculate volume for current cell
   *this.volume = ...;
}

and then the original routine becomes

void calculate_all_volumes( void ){
   for ( i = 1; i <= max_cells; i++ ){
      cell[i].calc_volume();
   }
}
```

FIGURE 3.7.8-1 : Identification of class methods.

The identification of class methods from FORTRAN legacy code was quite straight-forward because numerical codes tend to be optimised for speed rather than for memory usage. This means that developers introduce variables, that are initialised once at program start-up and then persist for the duration of the simulation, to hold values like face areas, volumes and normals

rather than lowering performance by repetitively re-calculating them. This allows methods to be identified in the initialisation stages of the software where these values are first set. The storage of such variables was kept for optimal performance and in some instances extended as other needlessly repetitive calculations were identified and subsequently replaced.

Another area where methods were identified was in routines that were simply "copied" and "modified" versions of other routines. Some code fragments had nearly identical algorithmic structure but used different variables. These routines were relatively simple to gather into a unified general purpose routine that is "parameterised" via function calls to provide the original functionality.

3.7.9 Stage (9): Optimisation and enhancements

The initial class oriented C++ version used statically declared arrays of objects. Dynamically declared arrays were relatively easy to implement provided that the sizes could be determined. In practice this required the use of temporary dynamic storage which is used to hold some of the geometry data whilst allocation sizes are determined. This use of dynamically allocated data structures gave much greater code flexibility without the need to re-compile for larger simulations. It was also possible to implement arrays of pointers to objects so that even individual objects could be created and destroyed independently, at run-time, as required. This could be very important for future methods involving mesh-refinement where more control volumes are created during the running of the program or for parallel processing where each processor could be handling a sub-set of cells.

The debugging of CFD codes has traditionally been a major problem because of limited or non-existent access to the internals of the algorithms and the problems of finding the appropriate values in the segmented and dispersed storage. The initial re-engineered system used direct data access to cell objects but such access cannot be easily controlled or monitored. The approach finally adopted was to use access functions that return data references rather than merely returning data values. Such access functions can then be used on either side of assignment statements to in order to set or get values. The figure (See Figure 3.7.9-1) shows the

implementation of the cell data access function in C++. Since data access was implemented as a function it is possible to plant debug code to monitor the usage of a chosen cell or variable or even to trigger some other analysis code. This is demonstrated by the example debug code to watch for negative temperatures. The optimised version of this access function uses a compiled inline definition (without any debug code) so that it should have little or no greater performance overhead than a direct array access.

```
Enhanced debug facilities for code development via class methods

float & Cell_Class::access( int mode, int var ){
#ifdef __DEBUG_CODE__
   if ( ( mode == NEWEST ) && ( var == TEMPERATURE ) ) {
      if ( data[mode][var] < 0.0 ){
//
// Error negative temperature detected in cell data access
//
      }
   }
#endif
   return data[ mode ][ var ];
}
```

FIGURE 3.7.9-1 : New data access function.

The solvers available in CWNN were all based on whole matrix solving techniques. It was suggested, by a fellow CFD researcher, that a true cell-by-cell solver should be developed for comparison purposes. This has been implemented from some of the re-engineered software components produced during this project but its usefulness is limited because of comparatively poor performance when compared to the "whole-field" solvers.

The implementation of a vector class for normal and displacement vectors has greatly simplified many aspects of the source code. The original FORTRAN code had to loop over all three dimensions for many geometric calculations whereas now, simple vector algebra can be performed. Operator overloading has been used to give the vector algebra a more natural syntax, as found in most reference texts. The figure (See Figure 3.7.9-2) shows the equivalent loops for calculating the cell centres from both the legacy code and the re-engineered code, which uses vector methods. Such instances were relatively easy to find because the loop dimensions go from 1 to 3 (or 1 to DIMENSIONS). It was possible to abstract one and sometimes even two levels

of looping because of the new operators and functions provided for vector algebra. These functions include dot- and cross- product utilities.

```
Original code for cell centre position calculation

DO I = 1, NOCELL
   DO J = 1, 3
      CENTRE(I, J) = 0.0
   ENDDO
   DO J = 1, 3
      DO K = 1, NPTCTY( CELTYP(I) )
         CENTRE(I, J) = CENTRE(I, J) + XYZCRD( CELPTS(I, K), J )
      ENDDO
      CENTRE(I, J) = CENTRE(I, J) / REAL( NPTCTY( CELTYP(I) ) )
   ENDDO
ENDDO

New C++ code using a vector class methods

for ( i=1; i<=num_of_cells; i++ ){
   cell[i].mid.set( 0.0, 0.0, 0.0 );
   for ( j=1; j<=cell[i].num_of_pts; j++ ){
      cell[i].mid = cell[i].mid + point[ cell[i].pt_num[j] ];
   }
   cell[i].mid = cell[i].mid / (float) cell[i].num_of_pts;
}

N.B. This code does not loop for the three directions because the
overloaded vector operators "+" and "/" hide these details
```

FIGURE 3.7.9-2 : Example of using a class for vector algebra.

Any further optimisation and enhancement features, that were identified early in the re-engineering stages, were researched and implemented in this stage. One such example of optimisation was the relocation of loop invariant calculations outside of low level loop constructs.

3.8 Statistics for the software Re-engineering process

The following code statistics provide a crude comparison between the legacy and new systems. This information should be regarded as being of academic interest only and not necessarily typical of any other or similar re-engineering projects.

- The legacy system consisted of 107 source files that contained 158 routines. There were 22,450 Lines-Of-Code (LOC) excluding comments.

- The re-engineered system has 4 source and 13 header files and has 395 routines including class member functions. There are 11,250 LOC in source files and 1,400 LOC in header files.

The project statistics (See Table 3.8-1) indicate approximate durations of the individual stages used during the re-engineering. The final stage (stage 9) has not been included because perfective and adaptive maintenance is ongoing. The project durations are measured in Person-Weeks (PW) where a Person-Week is defined as 5 work days for one system developer.

TABLE 3.8-1 : Duration of the stages of Re-engineering.

Stage	Description	Duration (PW)
-	Background research into un-structured mesh CFD.	6
-	Project planning and learning to use tools.	3
(1)	Ensure data consistency and make data global.	3
(2)	Name and algorithm clarification.	2
(3)	Removal of code and simplification.	3
(4)	Ensure consistent use of control and replace logicals.	2
(5)	Translate from FORTRAN to C++.	2
(6)	File I/O modifications.	3
(7)	Implement data classes to replace arrays.	4
(8)	Create class member functions.	4

It should be noted that stage 9 of the re-engineering is an on-going process and it is therefore impossible to detail the statistics associated with this stage of the development. Further information about run-time performance characteristics and an appraisal of the software re-engineering can be found in the Appendices (See Appendix 11.2).

3.9 Summary of chapter

This chapter discussed the limitations of the legacy software and assessed the issues concerning the re-use of the legacy system in order to create a suitable CFD software framework for research into interactive control techniques. This chapter also described the novel nine stage re-engineering methodology that was developed in order to re-use the legacy software. At the end of the incremental re-engineering process a suitable framework was available for the creation of the prototype interactive CFD system. The following chapter (See Chapter 4) describes how the re-engineered framework was subsequently modified and enhanced to create the interactive CFD engine.

4 Development of a prototype interactive CFD system

4.1 Overview

This chapter discusses the development of the prototype CFD system including the imposition of required design characteristics. The interested reader is directed to the Appendices section where there is more detailed information about the physical implementation of the Geometry Classes [See Appendix 11.5] and the Control Architecture [See Appendix 11.6] that was imposed on the Re-engineered CFD system.

4.2 Important aspects of design

4.2.1 Imposed re-design features

During the planning stages of this research it was known that a suitable application framework for research was required. This influenced the data storage design within the re-engineered system to make use of geometry based objects in a highly intuitive data architecture. This would facilitate CFD research by other researchers as well as significantly simplifying perfective and adaptive maintenance.

Many existing FORTRAN CFD codes use memory tricks or need re-compilation in order to mimic the flexibility of dynamic memory allocation of objects and their internal data. Such was not the case with the prototype system where maximal usage of dynamic memory allocation was planned and implemented. Occasionally this caused problems due to needing access to the data sizes prior to attempting the reading of data but some of the sizes could only be determined by analysing some of the data that had been read. This problem is overcome by reading data in two stages. The first read only determines the sizes whilst the second does the actual data reading after the correct amount of memory has been allocated. Where sizes had to be determined from the data then local temporary dynamically allocated storage was used.

It has always been the intention that the prototype application framework should be available for use in research into CFD algorithms, interactive control and optimisation. This led to the creation of object related methods which are considerably easier to support and maintain.

The direction of future research using the re-engineered prototype was largely unknown but it was likely that the software would be used for mesh refinement at some stage. It was decided to provide the ability to create and destroy objects at run time for continued research purposes. This implied using dynamic arrays of pointers to objects rather than the simpler dynamic arrays of objects because the latter approach would require more time consuming reorganisation of objects in order to allow for run-time creation or destruction of the objects.

One major potential advantage of this form of abstract conceptual design and object oriented implementation is that it would be possible to better exploit parallel or distributed processing platforms than using procedural parallelisation strategies [IEROTHEOU92]. This advantage comes from the fact that it may be impossible to hold all of the data for the solution domain on a single processing node. When data is partitioned between nodes it is found that many parallel implementations of software suffer from a lack of balancing of computational effort between the processing nodes and so it may be necessary to perform load balancing between the nodes. The task of load balancing is greatly simplified if it merely involves sending a single object to another node and updating the member lists rather than having to re-distribute portions of all of the disparate data arrays that would be found in a non object oriented implementation.

When the object oriented viewpoint of geometric objects was considered for implications it was realised that there was a potential for meaningful geometric or solution dependant "groups of cells" as intermediates between the whole domain of cells and each individual cell. It was not known what benefits could be realised using these groups but the group structures were created for partitioning the computational domain into geometric or solution based group collections of cells in order that the potential benefits could be researched. It was anticipated that groups would allow greater focussing of processing effort to areas where it was most beneficial.

The ultimate aim of the project was to investigate the benefits of interactive solution control. If

these benefits proved to be tangible then it was likely that some attempt would be made to automate any successful control strategies. It was realised that this future automation would be greatly facilitated if a consistent form of maintaining and communicating control and status information was created. When the structure of the earlier work on "FLOWES" [KNIGHT91] [PETRIDIS92] [PETRIDIS95] was considered it was natural to impose a so-called "blackboard" architecture that would use control and status objects which could then be used for communications between multiple processes. Essentially a "blackboard" is a structured global memory area which can be written-to and read-from by any of the application processes or sub-tasks. The "blackboard" also incorporates a scheduler to manage the order and frequency of running the processes and tasks. In any event these information and control link objects would greatly facilitate the implementation and data-transfer between the CFD engine and the GUI components.

It has already been discussed that flexible dynamic memory allocation can introduce considerable complexity when it comes to data access mechanisms. Since the prototype system has to be easily comprehensible to researchers and furthermore must support a consistent data view throughout the application it was decided to create data access functions, where necessary, to simplify and standardise the access to an object's internal data. This was particularly problematic where the data inside an object was itself dynamically allocated and thus accessed by an index list which would lead to verbose code to perform data accesses. Fortunately the C++ language allows for functional data access (that hides all of the dynamic data indexing details). This functional data access should have no greater performance overhead than direct array access to static memory since the function call may be generated with inline code - although this may depend on the quality of the C++ compiler and level of optimisation available. The issue can be forced, somewhat, by declaring the function as "inline" but this is still only a recommendation to the compiler rather than an assured behaviour.

Conventional thinking dictates that data should be kept as local as possible in order to prevent unintentional side-effects however in a CFD code this can greatly restrict the flexibility and increase the coding effort to provide new functionality. It was decided to use global data storage mechanisms to allow data to be visible in any routine and to enforce strict access mechanisms

to prevent accidental and unintentional data access. The benefit of this global approach to data storage and data access mechanisms is that data access is always consistent and well known for any and all code modules and allows for future developments with no detrimental effect on existing routines. This type of data storage is also close to the concept of the Blackboard architecture [PETRIDIS93] since data is not passed around the CFD code as arguments, but rather, the Blackboard maintains the data and the rest of the software uses query and update functions to access the data. There are a number of important maintenance and ease-of-comprehension issues associated with having a high degree of consistency between the conceptual design and the physical implementation.

The use of functional data access mechanisms was of some concern due to the potential performance overhead. The overhead of an extra layer of function calls has been mitigated by using the inline directives in the "fast" version of the code and ordinary functional access for the debug version. However, at the present time, the "inline" directive is only a recommendation to the compiler and it does not guarantee that the indicated code is actually generated locally rather than via a function call. Such functional data access mechanisms were necessary to the implementation of a "Blackboard" architecture for the data and the Blackboard access functions.

The consideration of data objects within the CFD software leads naturally to the consideration of object-specific methods. During the re-engineering a library of geometrically related methods, based on any commonly used code fragments that serve some identifiable purpose, have been formed. This is backed by observations [ANGUS94] that have demonstrated considerable ease of maintenance of Object Oriented implementations when compared to purely procedural software.

It was likely that the majority of software users of the initial prototype would be CFD experts and researchers. This had implications for the design of the interface between the GUI and the CFD engine. The GUI is necessarily the main program and the controlling process that repetitively calls the CFD processing segments to progress the solution further. Furthermore, since the scope of the required controls could not be predicted, all of the parametric solution controls had to be made available on the GUI.

4.2.2 HCI design issues

There were two distinct choices for the User Interface (UI) paradigm that could be used with the prototype CFD system.

4.2.2.1 Visual programming interface

A visual programming interface would use visual icons and links to represent data passing through and being modified by the system. Generally the tools could be "opened" and have their behaviour modified using a tool specific menu. Such interfaces have become quite common for image processing systems and, to a lesser extent, for visualisation systems. The problem with a visual programming paradigm for CFD is that it would be inappropriate to the experience of many users. There are often quite high initial learning overheads for visual systems but once the nature of the interface has been learnt then new components and tools are usually quite easy to use. It is generally quite difficult to implement visual interfaces in a portable way and all application tools have to be available and accessible from the interface. There is large scope for inappropriate configuration unless great care is taken for the component linking strategy. It is likely that a CFD visual programming interface would quickly become cluttered with tools and filters due to the complexity of the underlying numerical system and the potential for modifications to that architecture. There are currently very few examples of applications that use the visual programming metaphor so it was difficult to assess the potential effectiveness of this paradigm. Generally visual programming seems to be a very powerful tool when data transformations through a system can be conceived as a linked list of filters and transformation tools (E.g. AVS, Khouros) but the nature of the CFD system suggested that this type of interface be avoided. It is not simple to conceive of one, or more, simple data pipelines because of the complex data interdependencies inherent in CFD computations. A well designed visual programming interface would be a powerful tool for an expert CFD user or a CFD developer but most users would probably never use the majority of the capabilities of the interface and, furthermore, are likely to be confused to such an extent that in-appropriate linkages are formed that break the software.

4.2.2.2 Menu and form filling interface

A menu and form-filling user interface paradigm is much more common and familiar to CFD software users. The interface can be quite restrictive or obscure particularly when too many layers of menus are used thus, effectively, hiding control settings or when settings are inappropriately grouped together using obscure logical linking. Such interfaces are reasonably portable due to the large number of interface development libraries now available. There is generally a low to moderate learning overhead but when extra menus are added then these too need to be learnt. This approach does benefit from a consistent interaction style within application and similar style to many other applications. Generally the interfaces are largely self explanatory and intuitive provided that menus are not overloaded with settings and provided that the items are sufficiently verbose.

4.2.3 Human Computer Interaction issues

4.2.3.1 Multiple modes of data presentation

It has already been mentioned that there were a number of appropriate visualisation techniques for data from CFD codes. The new code interface should support as many modes of data representation as possible so that the user can make the most effective use of the interface. This means that data will be presented as numerical values, graphs and data visualisations within the CFD application UI.

4.2.3.2 The CFD system has a "user-in-charge" interface

Such an interface is reactive to the user. In the case of the CFD code this was implemented as a set of buttons which behave rather like the buttons on a tape recorder that can be used to start or stop the processing as well as to open configuration menus.

4.2.3.3 Visualisation

Visualisation, as previously mentioned, is relatively costly in terms of compute resources particularly for 3D displays. This performance overhead of visualisation has been mitigated by the choice of 2D slice visualisation and the provision of options to limit the frequency of visual updates. Another problem is the question as to whether intermediate (part processed) solution status displays have any meaning. This depends on the context of the display and the interpretation by the user. Any visualisation of a non-converged solution status has NO real physical meaning other than as an indication of the current trends within the data. At the end of a time step or when a converged solution is obtained (for steady state problems) the visualisation is a true representation of the simulated physical behaviour. It was not clear at this stage as to whether other variables could give more meaningful and indicative data and status visualisations, e.g. error values. Any visualisation tools built into the CFD code will serve a dual purpose because they can be used to monitor the intermediate solution status and they can be used as post processing data explorer when the simulation has finished. It is possible that there is a novice user role for pattern recognition for dynamic KBS reasoning. It is envisaged that even if planned attempts at automated pattern recognition fail then it will still be possible for users to visually detect patterns and to thus select appropriate options based on known examples and advice from the UI. In general, visualisation techniques can be costly to develop and to tailor to the underlying data architecture particularly for the unstructured mesh class of codes where there is no regularity or predictability in the layout of the data or the navigation between computational cells.

4.2.3.4 Graphs

Graphs can be highly informative particularly for data trends (continued movement in one direction by similar amounts) and instabilities (oscillatory behaviour). They are inexpensive in terms of compute resources and are generally easy to develop. Graphs may, however, over-emphasise certain solution features inappropriately (e.g. spot value graphs are not very useful unless used with experimental data or a known analytic solution).

4.2.3.5 Control "tinkerface"

The fully interactive nature of the planned CFD code leads to a distinct possibility of corruption of simulation data for uninformed or experimental control modifications by novice users. This meant that the system needed a comprehensive restart capability so that speculative research - that could break a simulation - does not require the user to start again from scratch but instead allows the user to jump back to the last valid situation before the bad control change was made.

It is also important that the control modifications are only applied at meaningful stages of the processing.

4.2.3.6 Minimise data on each form or menu and make menus specific to a task

Confusing UI design can often intimidate the software users to such an extent that it can result in functionality that is hidden or can leave the user with a complex navigational task through layers of menus to reach frequently used options. In order to prevent such problems, with the target development, considerable thought was put into the most appropriate style of interaction grouping for menus and forms. This included having a number of meetings with CFD practitioners to discuss UI design prototypes and using these prototypes to give a "walk-through" of the proposed system. In the formative stages of planning this project, meetings with CFD users were also convened to ask the prospective users about their requirements for the mode of interaction and their needs for data to be available via the User Interface. During the design considerations it was a design priority that no more than two layers of menus would be presented to the user because of the possibility of the user becoming "lost" in deeply layered menus. Conversely, because of the popularity and familiarity that many users have with most Microsoft Windows [MICROSOFT] based software, it was decided to design the style of interaction to be as consistent as possible with common software packages and the native GUI. Most of the guiding principles of "good" User Interface design, that have been employed in this research, can be found in the book by Thimbleby on the subject of User Interface Design issues [THIMBLEBY90] and a collection of papers on the subject [THIMBLEBY97].

4.2.3.7 Choice of portable library

At the start of this investigation there were a large number of user interface and graphical libraries available including X11, motif, MFC, Zinc, XVT, tcl, tkl, wxWin, Phigs, OPEN-GL, GKS, OpenWindows and INTERACTER, to name but a few. Consideration was primarily given to implementation language choice for the target CFD code before the choice of UI development library was made. The secondary consideration was the requirement for a high degree of portability that would include PC compatible machines as well as UNIX based workstations. This latter consideration was thought to be vital for extensive use and acceptance of the CFD code because of the prevalence of PCs within the academic community as well as the common use of PCs in smaller firms and consultancies. Whilst the targeting of the PC platform is necessary for a particular class of user, there is another class of user that needs to run significant problems on powerful workstations since the size and complexity of CFD research cases could preclude the use of PCs. One of the few libraries to meet all of the portability requirements and match the chosen target implementation language was the Zinc interface library [ZINC]. This choice also benefited the research because the Zinc library is itself heavily Object Oriented which ties in with the intended re-development strategy. Zinc also has an elegant event handling architecture that is platform independent and was one of the few libraries to be supplied in fully source code form. The only slight cause for concern was the fact that the Zinc library is a commercial product so future development of the CFD code would depend on the fortunes of the parent company.

4.3 Important aspects of implementation

During this research a number of techniques have been developed and implemented that are worthy of note because of their importance to the application area or because of the potential for continued research that they provide.

4.3.1 Restart database

One of the perceived problems with legacy CFD software is that once a simulation started to go

wrong, at some stage of the processing, then it usually necessary to start the simulation again from scratch with a different set of configuration options. This is clearly very time consuming and prone to error. In the prototype system a restart database is used to store sufficient information to continue a simulation from any saved stage. This restart database maintains the file formats of the usual configuration and set-up files but uses compression techniques to store them compactly. There is also an index table to allow easy selection and database management routines are available to selectively remove certain restarts. The restart database can then be used to regularly save stages of the simulation or can be added to whenever a potentially problematic control action is about to be taken.

4.3.2 Audit trail

Users of CFD codes are starting to be concerned about the reliability and accuracy of simulation data. The prototype system has been written to save ample configuration and status data as well as an audit trail of user control modifications. These various audit files contain all the information about the solution path that gives end users confidence of the final solution results. This is particularly important for safety critical simulations as in, for example, the fire simulation application area where building design and safety issues are paramount. Furthermore the command summary file can also be used to duplicate an earlier simulation exactly be re-imposing the same control actions at the same times.

4.3.3 User defined variables and code

The fact that the prototype system was intended for use in future research has meant that consideration had to be given to facilitating additional user code development. This has lead to the addition of user defined variables in the script file that, once defined, then behave in an identical way to the usual system variables. This means that the same syntax that was used for the usual system variables is also appropriate for additional user variables with no extra learning overhead. User coding links are still under development at this time.

4.3.4 Additional status variables

Traditional CFD codes make use of many temporary variables that may be of significance to the user. The prototype system has been developed to maintain some additional status variables, if required, in order to give a better indication of solution status. This is particularly true of the cell residual variables which store the cell-by-cell residual error for any variable as an additional data field. These status variables can be particularly useful in tracking down problematic parts of the geometry that are causing unpredictable solution errors since it is possible to visualise these residuals in the same way as any other variable and hence locate problem cells.

4.3.5 Finding common structure for momentum and other solved variables

In the original legacy code momentum was solved using a special treatment that was almost a vector handling as opposed to the simple scalar handling of other variables. The prototype system has been reformulated to treat the individual momentum components as simple solved scalars because this allows all of the solved variables to be unified into a common structure where the only differences are in the system matrix coefficient formulation and the source term calculations. This leads to a much cleaner software architecture and allows a consistent and extensible functional access to system matrix calculations to be provided.

4.3.6 Automated saving

Early in this project it was known that a significant amount of research would be needed to validate the implementation and to investigate the benefits of interactive control of the CFD code. Since it was known that much of the CFD research would be quite speculative, whereby a solution is unknown when the simulation is started, it was deemed necessary to provide a suitable support structure that would facilitate this research. The provision of both manual and automated saving facilities allows the configuration of data saving so that regular saves of visualisations, status graphs, data plots, results and restarts can be made. It was assumed that there would be circumstances where the immediate monitoring data and status might suggest one course of control action but that another control action might be more appropriate when

considering a whole history of monitoring data and solution states. This consideration may also be of relevance to the purpose behind a particular simulation and the form that the results will take. This is particularly important for transient simulations where the interest in not so much in any final solution state or data but is rather in the critical phases and changes that occur during the full simulation. With reference to using fire modelling for assessing safety of the built environment, it is insufficient to merely say that at the end of the simulation conditions in a room fire were non-survivable when it would be of more use to determine the first time at which the room fire became truly dangerous.

4.3.7 Debugging facilities

Since the code is intended to be used for algorithm development it was considered vital to provide comprehensive debugging facilities. It is not sufficient to rely solely on compiler provided debugging because the variability of such debuggers is very large and the navigation of complex data structures can be problematic and time consuming. In the prototype system two separate debugging facilities have been provided. The first allows any stored variable to be output in human readable form just after it has been calculated. This output can be limited to a particular simulation time and range of cells. The second facility allows specific data monitoring code to be planted in the data access functions. This allows the values of any data item to be monitored throughout the simulation whenever data is accessed. Of course these techniques do not prevent a developer from using compiler debugging techniques but do provide additional tools to track down hard to find errors that are a quite common occurrence when developing new algorithms.

4.3.8 Automatic self extending arrays

Sometimes it would be inefficient, in terms of memory usage, to allocate arrays to the maximum anticipated size when such arrays may only infrequently use their full extent. Conversely if the length of an array is likely to change quite often then re-allocation is likely to be inefficient in terms of performance because of the book-keeping involved with creating a larger array and moving the existing elements across to it, followed by destruction of the old array space. There

is a middle ground that has been exploited in the prototype system for such arrays that uses arrays that will extend (or contract) automatically by a pre-configured chunk size. This basically means that if an extra element of the array is added but there is no room for it in the array then the array will grow automatically and then the element will be added. If the array was already large enough then the element is simply added without extension and the write pointer is moved on to the next available array slot. Such techniques can be quite elegant in usage since no sizes have to be determined for dynamic memory allocation.

4.3.9 Unstructured visualisation techniques

It is quite complicated and computationally intensive to produce visualisations of data from unstructured meshes. The prototype system uses planar slice 2-D visualisation of an arbitrarily positioned x-, y- or z- plane. The simplest way to handle this situation would be to interpolate the data required for visualisation onto the plane and display it as scattered data. However an alternative treatment was sought because the best quality display possible is required with least possible computational overhead. The adopted method first locates all cell-centre to cell-centre lines that would be cut by the required plane. The intersection points are assigned interpolation weights based on their relative distance from the cut plane and the indices of the neighbouring cells are also stored. These scattered points are then re-meshed into triangles using Delaunay triangulation [FIELD91]. This allows any selected variable to be quickly interpolated to the plane using the interpolation weights on the neighbouring cells data values. The relatively expensive triangulation only needs to be performed once, when the user confirms that a new plane should be used but the mesh of triangles allows high quality visualisation of contour lines or contour fills.

4.3.10 User configured patch and time step modifiers

It was often observed, by the developer, that seemingly simple tasks in other CFD codes resulted in users having to write, compile and link additional user-defined code. This mostly seemed to be due to a lack of foresight on the part of the original code developers. It was clear that more complex scenarios (in the fire modelling application area) that exhibited some degree of realism

would possibly include such features as opening doors, breaking windows and ignition of secondary fires. It was quite simple to define additional control commands to allow the various physical patches to be swapped to alternate definitions at pre-configured times. This greatly enhances the usability of the software without requiring the user to write additional code.

4.3.11 Solution configured patch and time step modifiers

Once it was realised that the user could pre-configure patches to change at given times it was also clear that it might be desirable for the solution to control such events. This is also quite easy to manage since it only requires that data monitoring code be activated and then the boundary patches check with the data monitor to see if it is appropriate to swap to an alternate patch definition rather than checking some pre-configured time for the swap. This functionality again removes the onus from the software user for writing additional source code and rather makes the task a simple configuration method.

4.3.12 Configuration of results saving from sub-regions

Generally, at the end of a simulation, the user will be presented with results files for all of the simulation data for all of the cells of the domain. This does not help the user to comprehend the data when the user may only be interested in smaller sub-regions of the whole simulation domain. In order to provide the user with appropriate data analysis the prototype CFD code has the ability to output results for specified sub-regions. The regions are specified by low and high co-ordinate position and all cells contained within such specified volumes will have their results output.

4.3.13 Tabular data files for volume source variation

Increasingly users of CFD codes are aiming to obtain better and better accuracy from their CFD simulations by using experimental data, wherever possible rather than crude functional approximations. One such area in fire modelling scenarios is for the definition of the heat load that is applied by a given fire in some experimental set-up. Often the actual heat load at any time

can be determined for the experimental set-up and so it is useful if this data can also be used in the CFD code to accurately represent the fire. Again the policy with the prototype CFD system is to limit the need for the user to write source code to extend the CFD capabilities so a general table file handling volume source has been defined which allows the simple use of time varying tabular source data to be read in from a file.

4.3.14 Run-time modification of volume source application region

A recent idea has been to allow the user to simulate fire spread by extending the volume over which the fire load is applied. At present there is no automated means to spread the fire within the prototype system however the first step is to provide a manual method of enlarging the fire volume so that a reliable automated methodology can be found.

4.4 Summary of chapter

This chapter has described the considerations and implementation details that were used to transform the re-engineered CFD framework into a vehicle for interactive control research. Once the new interactive CFD system had been completed it was necessary to verify that the re-engineering process and subsequent prototype developments had left the original functional behaviour, of the legacy CFD system, unchanged. The following chapter (See Chapter 5) describes a sufficiently wide coverage of test cases, and their results, that were used to verify that the whole of the new software framework was consistent with the original legacy software.

5 Prototype system validation

5.1 Incremental testing (functional comparison with legacy FORTRAN code)

At each stage of the reverse engineering and subsequent re-engineering it was deemed to be vital to validate the system functional consistency with the original legacy FORTRAN CFD code to ensure that the current stage of the development maintained the same behaviour as the original. This process was largely automated (in batch mode scripts) at the end of a stage of work so that a known simulation case, that provides full code coverage (i.e. using most, if not all, of the modules and algorithms within the CFD code), was run overnight and the data output files were compared by a numerical differencing utility to ensure reasonable consistency. If any stage of re-engineering or development produced a different set of results then an inspection of the differences was performed to see if they were significant. In the event that the developer was unsure of the significance of any observed differences then a CFD expert was consulted to determine if the new results were acceptable. In general the phrase "in good agreement" means that there were no significant differences in the results.

5.2 Final validations

In order to determine the overall usefulness of the reverse engineering process a sufficiently wide coverage of fire modelling and primitive physics validation test cases were constructed to test (both individually and collectively) all of the sub-models relevant to fire field modelling contained in the re-engineered system. Some of these validation test cases have subsequently been used as the standard validation suite for the SMARTFIRE system. The complete validation report for the key test cases has been included in its entirety in the appendices section of this thesis [See Appendix 11.1] but a summary, of the most important validations, is presented here for convenience. These simulation cases were used to compare SMARTFIRE with several other commercial CFD codes (or experimental data, if available) in order to check the correctness of the results.

The validation case "Two dimensional flow over a backward facing step" was used to validate that the flow and turbulence modules were working correctly. The goal of the case is to develop a fully recirculating flow region behind a sudden expansion in the down-flow direction of the duct. Experimental studies have shown that a parametric solution is obtained that is dependent on the height of the step and the Reynolds number. In the test case, conducted for the validation, the experimental re-attachment point is 7.0 step heights down stream from the step. This re-attachment point is the downstream limit of the re-circulation where the flow at the outer boundary once again follows the dominant flow direction down the duct. In the simulation tests the prototype CFD code gives a re-attachment point 6.0 step heights down stream from the step. This is the same value as the legacy CFD code (and is comparable with other CFD codes which use the K-Epsilon turbulence model). This reduction in re-attachment length is reported in many papers (which have analysed various turbulence models) and is typical of the standard K-Epsilon model used in the prototype and legacy software.

The validation case "Turbulent long duct flow" represents flow along a “long” square section duct such that the flow speed is sufficient to produce a fully turbulent flow. The results from this simulation are in good agreement with those from the legacy software.

The test case "Turbulent Buoyancy flow in a cavity" represents a natural convection scenario where the flow is created by the buoyancy effect of a hot and a cold vertical wall on a fluid. The results were found to be in good agreement with published data.

The test case "Steckler room fire" is a simulation of a fixed heat output fire within a compartment that has a single door. A variety of tests were performed in the actual Steckler experiments but a typical scenario was chosen for the comparisons. The results from SMARTFIRE give good agreement to the published data and are consistent with the results from the other CFD codes.

5.3 Basic implementation validations

A number of basic validation comparisons were also performed to check that the re-engineered CFD system was consistent with the legacy CFD software. The actual simulation results and the detailed set-up configurations, for these simulations, are not particularly relevant to the current discussion and only a brief outline is given of the test cases and their respective results.

The validation case "Diffusion controlled combustion" uses two parallel inlet jets with a jet of fuel and a jet of oxidant into a 2D combustion chamber to test the simple combustion model. The Simple Chemical Reaction Scheme (SCRS) uses a much simplified chemical reaction equation which turns appropriate proportions of fuel and oxidant present within a cell into some product material with the consequent production of heat which is fed into the energy equation. Once again the re-engineered CFD system performed as expected and consistently when compared to the legacy software.

The validation case "Heat bar using multiple materials" uses simple heat transfer along a 2D bar that is constructed of two different materials. This heat transfer is caused by the imposition of an elevated temperature boundary condition at one end of the bar whilst the other end is maintained at some ambient temperature. The results for the temperature profile down the axis of the materials agree with the analytical results. This case reduces, essentially, to a 1 Dimensional heat conduction problem. The results obtained were as expected for both the re-engineered and legacy software.

The validation case "Heat bar using a triangular mesh" also uses a simple heat transfer along a bar but in this case the bar is only constructed of a single material and uses an unstructured mesh of triangular cells. The ends of the bar are maintained at different temperature and heat flux boundary condition combinations and a steady state temperature profile is expected. The unstructured correction terms are used to enhance the solution accuracy for these unstructured mesh cells. The results from the prototype software give good agreement with the results from the legacy software and the analytic solution.

The validation case "Heat bar using a user variable" uses a simple single material bar to check the user variable solver. In the normal case described above the heat variable is solved and

temperatures calculated from the heat content of the cell and the specific heat capacity. In this case, however, a user defined variable is used to represent the heat (i.e. the Enthalpy variable). The results are essentially the same as for the standard heat bar case but have made use of the extra variable solvers.

The validation case "Moving lid cavity" represents an idealised infinite length square cross section box that has a lid that is moving across the top of the box at a uniform rate. The viscosity of the fluid within the box causes momentum from the moving lid to be transferred into the fluid cavity and a fully recirculating flow is developed. The results from this case are in good agreement with the legacy software results and were deemed to be acceptable by an expert CFD user.

The validation case "Natural convection" combines flow, heat transfer and buoyancy for a fluid filled box. In this case one of the vertical walls is maintained at an elevated temperature whilst the opposite wall is maintained at a cooler temperature. The uptake of heat near the heated wall leads to density changes that result buoyancy forces that drive a fully re-circulating flow. Eventually a steady state is reached whereby the uptake of heat from the hot wall is perfectly balanced by the loss of heat to the cold wall. The results from this case are in good agreement to those from the legacy software and the analytic solution presented in journal papers [JONES79] [DAVIS83].

5.4 Interpretation and comments

It is clear from the various validation cases that the re-engineered software is in good agreement with the legacy CFD software. There are, almost inevitably, small differences but these are to be expected due to minor implementation language dependencies. Furthermore the order in which mathematical expressions are evaluated are likely to be different between the two software versions which may explain some of the small discrepancies.

As the validation cases become more complex (the fire field modelling cases have more complex geometries, more extreme rates of heating and many more degrees of freedom) so the agreement

between the re-engineered software and alternate CFD software packages becomes less consistent. It is also observed that the legacy CFD code and the other validation CFD codes do not agree completely and the re-engineered code tends to have results that are within the bounds of the other CFD codes tested. Small differences in boundary condition handling and solution schemes are responsible for most of these differences.

5.5 Summary of chapter

This chapter has demonstrated that the re-engineered, and subsequently the interactive, CFD framework is functionally consistent with the behaviour of the legacy CFD code. It was now possible to use the interactive CFD engine to research the potential benefits of using interactive control. The following chapter (See Chapter 6) describes the test scenarios that were used for research and gives interpretations of how the results demonstrate the benefits of interactive control.

6 Research results

6.1 Overview

In order to investigate the potential effectiveness of user interaction techniques, for Computational Fluid Dynamics (CFD) modelling, it was decided to choose an application area that gave ready access to a number of CFD experts and furthermore was an application area where the accuracy and correct interpretation of results was of a safety critical nature. One such topic of research at the University of Greenwich is Fire Field modelling. This application area is interesting because of the complexity of the geometry used in the simulations and the requirements for the modelling of high rates of heating and thermal radiation [KUMAR91]. Furthermore, many fire simulations have to be performed in transient mode so that the time varying nature of the simulation is revealed. This is partly due to the extremity of the physics being modelled but also due to the types of question asked of the modelling which includes the determination of temperatures and smoke concentrations at certain times in order to research safety issues relating to, for example, fires in compartments and buildings [LEWIS97]. The added advantage of using this application area is that there is considerable "in-house" expertise available for the use of fire field modelling. Of particular importance have been the comparisons [KERRISON94] [BJORKMAN95] of various CFD codes against the experimental work of Steckler [STECKLER82] which give a useful set of validation and comparison data for the modelling of fires within compartments.

The results presented in this section are indicative rather than exhaustive but the intention has been to demonstrate that the traditional techniques used for CFD in general and Fire Modelling in particular have mostly ignored the issues of interactive solution control, to their detriment. Recent questions about the reliability and accuracy of CFD techniques used in ever more critical simulations will, almost inevitably, tend to embrace technologies, such as interactive control and monitoring, in order to give more assurance of solution correctness.

The results presented here demonstrate that even expert CFD users can have significant

difficulties choosing a sufficiently restrictive, but also optimal, set of solution control parameters for a previously unseen simulation scenario where those control parameters are used for the whole of the simulation. Furthermore the results and timings from selected fire simulation scenarios demonstrate that run-time adjustment of control parameters can lead to savings of up to 50% for overall processing time when compared to some "safe" initial set of relaxation parameters used throughout the simulation. Clearly such savings are highly problem specific but the principle of choosing a known "safe" set of control parameters and then adjusting the controls as required by the most up-to-date solution status is highly recommended and likely to be of great benefit to both expert and intermediate users. Ultimately it is anticipated that these experiences of run-time solution control will be automated so that reliable CFD simulation and monitoring is made available to all classes of user from novices to experts.

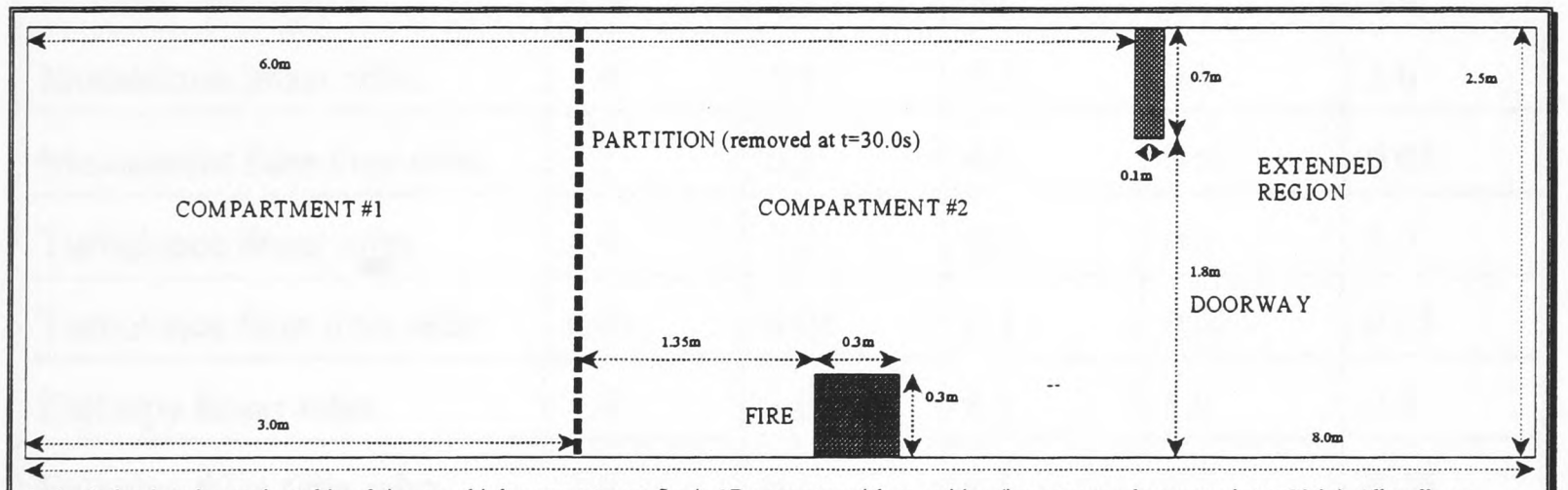
6.2 Indicative test cases

One of the important factors, when considering CFD simulations, is the time required to arrive at the results. This is an easy quantity to measure and gives a reasonable indication of the effectiveness of the interactive control. The problem with attempting to assess the benefits of interactive control, of the solution parameters, is that the effectiveness will be highly dependent on the quality of the initial solution parameters. The factors which determine a good choice of the initial control parameters are prior knowledge of the simulation of similar cases, a reasonable understanding of the behaviour of the particular CFD code in question and an adjustment (based on engineering judgement) to account for the particular simulation being conducted. There are, however, no hard and fast rules to prescribe a suitable set of initial control parameters.

This situation is further complicated by the highly complex nature of CFD simulations which means that the simulation controls required to start a simulation are likely to be too extreme for the later stages of the simulation. There are also potential transient characteristics of the flow solution which typically require even tighter control regimes to prevent the simulation from becoming unstable.

6.2.1 Investigation of initial configuration

In order to determine how good an initial set of control parameters would be selected, by a typical CFD expert, a questionnaire (See Figure 6.2.1-1) was formulated, with the help of a CFD expert user, in order to get some indication of how various CFD researchers would configure a new, previously unseen, simulation specification within the prototype CFD code. The variety of the controls available in the interactive prototype meant that, for ease of description and brevity, some limitations had to be imposed on the range of controls used in the simulation. It was decided to impose a fixed time step size and specified simulated period but to allow the CFD experts free access to choose their own number of iteration sweeps per time step and the linear- and false time step- relaxation values for all of the solved and calculated variables as appropriate. These user specified control parameters were then used in test simulations in order to check their effectiveness.



A simulation is to be conducted involving a multiple compartment fire in 2D geometry with a partition (instantaneously removed at t=30.0s). All walls are assumed adiabatic and the extended region has the side and top set as free (outlet) type boundaries. Front and back surfaces are symmetry planes The case is run for 110 time steps each of duration 1.0s (except for 5 time steps when the partition is removed that use a time-step size of 0.2s to ease the transition). The constant fire output is 200kW. The mesh has regular spacing of 0.1m in both the x and y directions within the two compartments. The air is assumed to be a compressible ideal gas and uses a non-boussinesq approximation for buoyancy. The K-e turbulence model is enabled.

REQUEST: Given the case as described above please can you give your best estimate of the number of sweeps (iterations) per time step and the relaxations needed to give a safely converged solution with the least possible run time. You can use any combination of relaxation values in the table below.
N.B. Linear relaxations should be in the range {0.0001, 1.0} and false time relaxations in the range {0.0001, 10.0} or NONE.

REQUIREMENTS: Each time step should converge to 0.1% of the final maximum of each variable. I.e. the CFD code normalises all variables by the final maximum (already determined) to obtain convergence when all residuals fall below 0.001.

VARIABLE(S)	LINEAR RELAXATION	FALSE TIME RELAXATION
Pressure correction		N/A
Momentum		
Turbulence		
Enthalpy		
Temperature		N/A
Buoyancy		N/A
Density		N/A

Required NUMBER_OF_SWEEPS per time step

FIGURE 6.2.1-1 : Questionnaire used to obtain control selections from CFD experts.

Four sets of control parameters were returned on the questionnaires and these were run as

specified. Suitable monitoring was applied to give an assessment of the appropriateness of each set of initial control parameters. A base set of "safe" control parameters was also used as a comparison for the expert specified cases. This "safe" set of relaxation parameters was arrived at from the Software Developer's experiences of validation and testing of the prototype interactive system when used on similar simulation scenarios.

The CFD users who answered the questionnaire gave the following recommended control specifications for the partitioned room fire simulation.

TABLE 6.2.1-2 : Control regimes taken from questionnaires.

Control Item	Safe Set #1.1	Expert #1.2	Expert #1.3	Expert #1.4	Expert #1.5
Number of sweeps	100	200	100	200	30
Pressure relaxation	0.4	0.6	0.6	0.1	0.8
Momentum linear relax	1.0	1.0	0.2	0.1	1.0
Momentum false time relax	0.1	0.1	0.5	0.01	0.05
Turbulence linear relax	1.0	1.0	0.2	0.1	1.0
Turbulence false time relax	0.01	0.05	0.1	0.01	0.05
Enthalpy linear relax	1.0	1.0	0.2	1.0	0.5
Enthalpy false time relax	1.0	0.1	0.5	0.1	0.05
Temperature relaxation	1.0	1.0	0.5	1.0	1.0
Buoyancy relaxation	1.0	1.0	0.6	1.0	1.0
Density relaxation	0.5	0.8	1.0	0.1	0.8

The simulations where run as specified by the various experts and the run-times and convergence behaviours are shown in the table (See Table 6.2.1-3).

At first glance it appears, from the timings, that the user specified set #1.5 returned the optimal performance because of its shorter run-time, however the simulation results were very poor for that set of initial control parameters because none of the time steps actually converged to a

satisfactory degree due to the low number of configured sweeps per time-step. The only set of acceptable parameters used was from user specified set #1.2 which only failed to converge on two of the time steps and then only by a relatively small factor. User specified set #1.3 was reasonably stable but failed to reach convergence in 25% of the time steps whereas user specified set #1.4 had too much under-relaxation that caused the time steps to do very little useful processing with none of the time steps actually converging.

TABLE 6.2.1-3 : Processing timings for the various control regimes.

Initial control set	Cumulative number of sweeps	Total processing time (seconds)	Non-converged time steps
#1.1 Safe initial control parameters	10750	13641	0 of 110
#1.2 Expert user specified control parameters	7728	9806	2 of 110
#1.3 Expert user specified control parameters	10900	13831	28 of 110
#1.4 Expert user specified control parameters	16329	20720	98 of 110
#1.5 Expert user specified control parameters	3267	4145	110 of 110

The configurations used above were selected by a small, but hopefully quite representative, group of researchers with various degrees of familiarity with the particular CFD engine used but all with considerable familiarity with CFD techniques in general or specific alternate CFD codes. This highlights another problem, which is the unique behaviour of different classes of CFD code to the initial configurations. When the users were informed of the quality of the control specification, that they provided, most were surprised that the settings they would have used in their usual CFD system did not work well in the prototype interactive system. This lack of transferability of set-up knowledge means that users are forced to learn the idiosyncrasies of each new class of code (e.g. staggered mesh, unstructured) and the behaviour due to the particular combinations of approximations, solvers, boundary condition handling and empirical techniques used by each CFD system.

6.2.2 Investigation of adjusting solution control during a simulation

The second investigation used the same test case scenario of a 2D room with a fire and a removable partition. A "safe" set of configuration control parameters was selected for the trial and then a variety of different control strategies were adopted to modify the parameters to attempt to obtain the same ultimate solution in a faster time.

The simulations conducted were as follows:

#2.1 No adjustments to the initial set of relaxation parameters.
#2.2 Manual adjustments applied by an expert user as the simulation progresses.
#2.3 Using the final relaxation values from #2.2 as the initial relaxation values.
#2.4 Automated adjustments applied by a prototype KBS as the simulation progresses.
#2.5 Using the final relaxation values from #2.4 as the initial relaxation values.

The modification strategy used by the expert user was that towards the end of each time step (as determined by convergence) or if a problem was observed then the user could decide to temporarily halt the processing and modify the relaxation parameters in a positive or negative sense based on the convergence graphs and visualisation of the current solution state. The relaxation parameters could only be changed in a positive sense (i.e. a lessening of under-relaxation) by up to 25% of their existing values and, at most, only one stage of removal of under-relaxation could be performed during each time step. This limitation had to be imposed after observations made in preliminary research showed a "run-away" control regime in certain circumstances that kept on increasing or decreasing the relaxation at every control test. It has also been observed that making too large a change in the relaxation values can "kick" the solution so hard that it never regains stability. This is the reason for the 25% change limitation. Whilst these limitations may appear very restrictive it has been observed, from both automated and manual control interventions, that small and gradual changes are much less likely to destabilise the solution whilst still providing the potential for significant optimisation savings. There were no imposed limitations to applying more under-relaxation if some convergence problem was detected. Furthermore there were imposed upper limits to the relaxation values so that some minimal level of under-relaxation was always applied. These restrictions were imposed

to attempt to limit the learning, by the expert, of an optimal set of parameters and applying them (as an alternate initial control set) in the first time step. A description and discussion of the prototype dynamic control KBS and its mode of operation are given in the paper [EWER98] that is included in the appendices. The runs, which used the final relaxation configurations as the initial set-up, were used to determine if the "safe" set of parameters was a particularly non-optimal set of relaxation parameters.

The results obtained for the computational effort for the entire simulation were as follows:

TABLE 6.2.2-1 : Computational effort for various control strategies.

Control strategy	Cumulative number of sweeps	Total processing time (seconds)	Non-converged time steps
#2.1 Safe initial control parameters	10750	13641	0 of 110
#2.2 Expert user adjusted control parameters	5200	6598	0 of 110
#2.3 Final control parameters from #2.2	13940	17689	34 of 110
#2.4 KBS adjusted control parameters	4741	6016	0 of 110
#2.5 Final control parameters from #2.4	5364	6807	7 of 110

This time it is clear from the process timings for #2.2 and #2.4 that there are better sets of relaxation parameters than those used in control set #2.1. However it is also clear from the timings for #2.3 and #2.5 that it is not sufficient to simply apply less under-relaxation from the start of the simulation as this was observed to destabilise some of the time steps such that a converged solution, to some of the time steps, could not be obtained within a reasonable number of sweeps. The prototype KBS used in this investigation provided marginally better control than the one described in the paper [EWER98] because it had subsequently been optimised to incorporate slightly better rules for limiting and applying relaxation modifications and the CFD algorithms had also been improved. The problem with the prototype KBS is that it is somewhat inflexible to alternate simulation scenarios, particularly those with more degrees of freedom,

since it was implemented based on observational experience of the manual control of this 2D partitioned room scenario. The KBS has not performed particularly well on 3D room fire simulations and more research is needed to ensure that the dynamic control KBS is more reliable for general simulations.

In order to check that the controlled path to the solution does not affect the final simulation solution the final results were compared. In this simulation the results for a vertical line of temperatures in the middle of the room were compared for consistency. The graph (See Figure 6.2.2-2) depicts the vertical temperature profile and indicate that there are no significant differences when comparing the results of the different simulations in spite of the vast differences in applied computational effort.

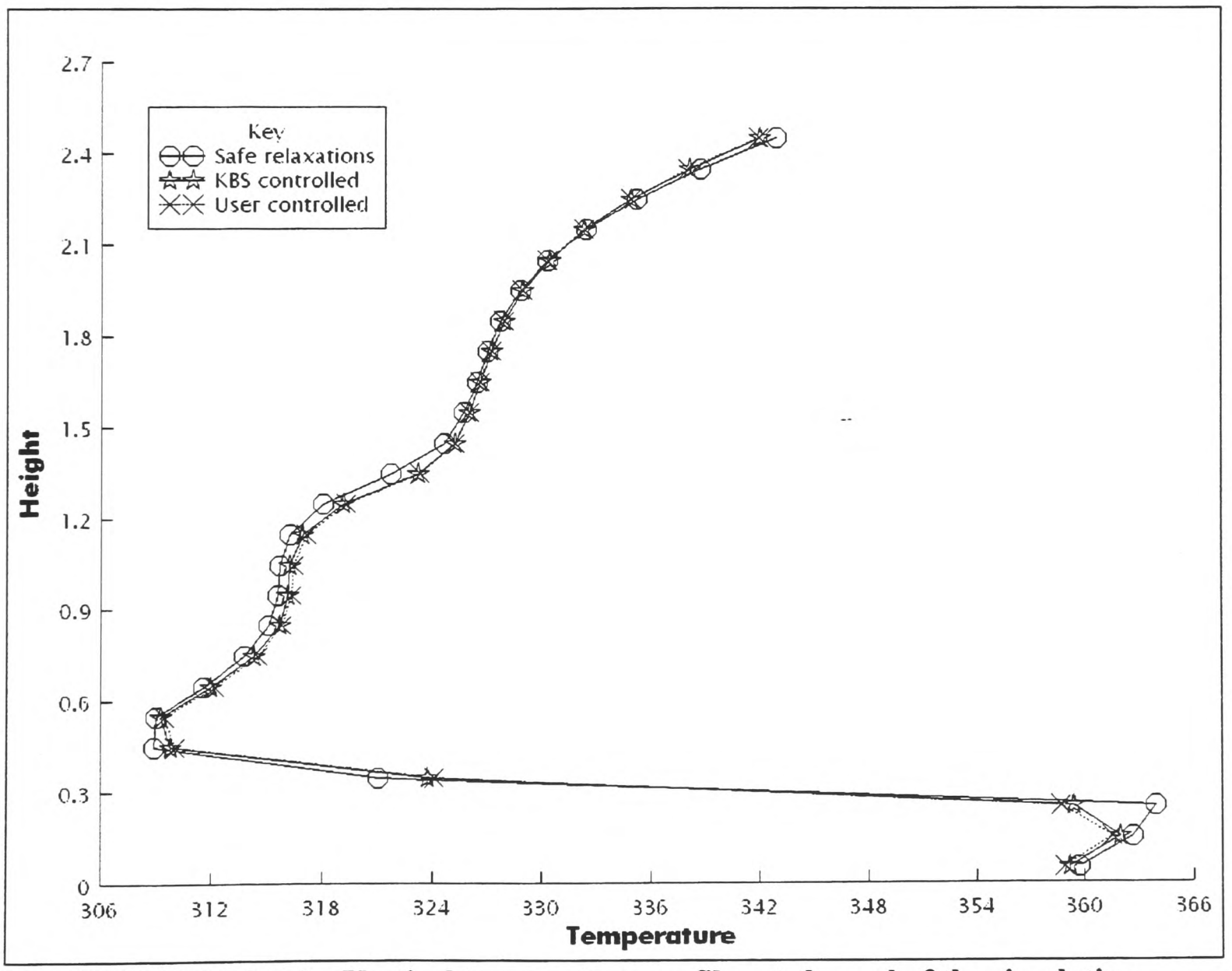


FIGURE 6.2.2-2 : Vertical temperature profiles at the end of the simulation.

6.2.3 Investigation of dynamic control of a more complex fire scenario

The limitations of the prototype KBS do not restrict the expert user from conducting manually controlled investigations on more complex fire scenarios. Another simulation was devised to investigate manual solution control in a 3D fire case.

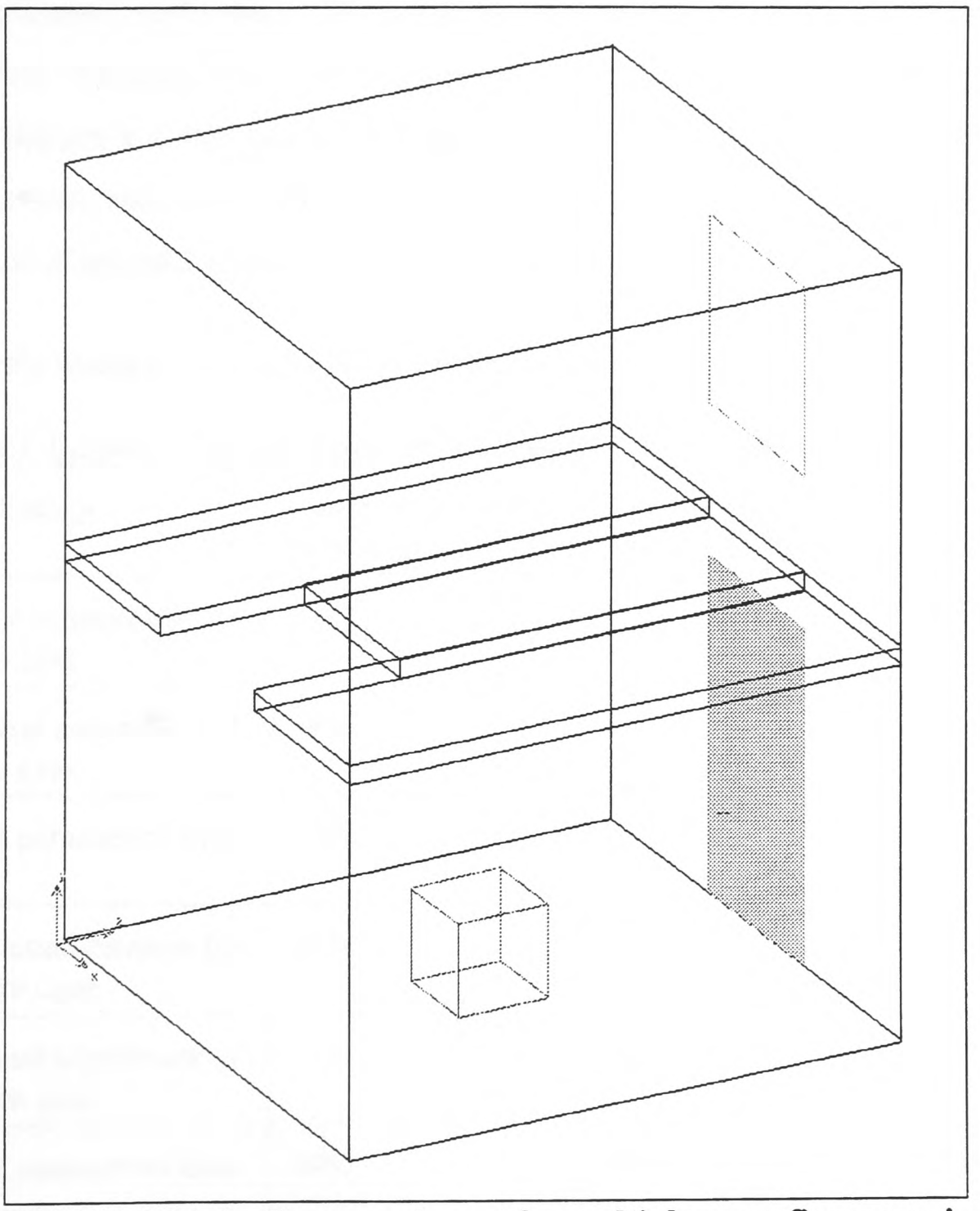


FIGURE 6.2.3-1 : Geometry layout for multiple room fire scenario.

The case investigated was a two storey barn (See Figure 6.2.3-1) that had an open doorway on the ground floor and an open window on the first floor, directly above the door. There was an open ladder hatch between the floors towards the back of the room and a centrally located fire on the ground floor. In this investigation the tests were conducted "blind" so that the expert

CFD user only knew that the simulation would run successfully, with the configured "safe" set-up, but had no indication of how many iterations were required for each time step to converge. Furthermore, only a first attempt at manual control was used for this investigation in order to prevent the user from "learning" the optimal behaviour for the particular scenario in question. The limitations and restrictions for the manual control adjustments were as described above for the 2D partitioned room with a fire simulation. The fact that this simulation was modified "sight-unseen" did mean that a more tentative approach was adopted when applying any relaxation changes but this was considered to be a more realistic use of manual interactive control. The tests were re-run with a fire that had a heat output of four times that of the former case as a test of the modelling of more extreme physics.

The following timings and computational effort measurements were obtained:

TABLE 6.2.3-2 : The processing effort required for various control strategies.

Control strategy	Cumulative number of sweeps	Total processing time (seconds)	Non-converged time steps
#3.1 Safe configuration for 50kW fire case	3464	5577	0 of 100
#3.2 Manual adjustment of 50kW fire case	1834	2953	0 of 100
#3.3 Final parameters from #3.2	1707	2748	0 of 100
#3.4 Safe configuration for 200kW fire case	4448	7161	0 of 100
#3.5 Manual adjustment of 200kW fire case	3155	5080	0 of 100
#3.6 Final parameters from #3.5	2993	4819	0 of 100

In these investigations the savings due to manual control were 47% for the smaller 50kW fire scenario and 29% for the larger 200kW fire scenario. The fact that simulation #3.3 and simulation #3.6 produced marginally better savings without loss of convergence stability implies that the initial "safe" set of relaxation parameters are a little too restrictive but this could not

have been known or predicted prior to the investigation. Whilst this result might seem to suggest that manual control is actually less important than appropriate set-up, it is argued that these results demonstrate that there is still significant potential for solution optimisation from even a safe initial set-up based on observation and manual control. There is also the consideration that the simulation case is quite simple and stable compared to some of the fire modelling research that is actually performed. When a case involves critical events (e.g. window breaking, secondary ignition or flash-over), which may change the flow characteristics and solution behaviour drastically, then it is highly unlikely that a single set of initial safe relaxation parameters will be appropriate and optimal for the entire simulation. In practice, it was observed that this simulation was atypically stable and quite easily approached a steady state solution due to the fact that there was a natural flow path through the building that did not tend to build the often-seen opposing layered flows. It is clear from simulation #3.1 and simulation #3.4 that merely changing the output fire heating rate can greatly influence the amount of processing required to obtain a converged solution. This is intuitively obvious since a higher rate of heating will lead to proportionately faster flows and hence pressures and turbulence will also be more extreme (and hence harder to converge).

The multiple room simulation, from above, was also conducted without any relaxation at all (except for the usual 0.6 linear relaxation on pressure that is generally required by the SIMPLE algorithm [PATANKAR80] for stability). In this case the solution entered a quasi-stable state, during the first time step, where the solution was oscillating at quite high residual errors with no real tendency to either diverge or converge within any time step.

6.3 Assessment of the benefits of interactive control

The problems of selecting an "appropriate" control configuration, faced even by experienced CFD users, vindicate the investigation of interactive control and monitoring as a necessary research program that is needed to obtain a better understanding of the practicalities of CFD simulation and to pave the way for reliable automation of solution control.

It has been observed that there is usually some "acceptable" band of control parameters for each

simulation scenario. When control parameters are chosen outside of this acceptable band then at best the solution will stagnate or oscillate and at worst it will completely corrupt the simulation results so that the only available course of action is to simulate again from scratch. Unfortunately the acceptable band of control parameters is case specific and unknown for each scenario until some stability research has been conducted. The control parameters at the upper edge of the acceptable band are likely to give the fastest possible simulation times, however, these controls are also the most likely to cause data corruption. Simple observation has led to the conclusion that the initial stages of a simulation are the most unstable and hence it is generally the case that greater under-relaxation is required to start the simulation. This indicates that the safest way to proceed is by removing under-relaxation from a sufficiently "safe" set of restrictive initial relaxation values.

This does not, however, address the natural solution or pre-configured events that may occur during a simulation. These "events" can happen at unpredictable times and are generally associated with a significant change to the stability of the solution. Examples of these events are the changing of height of a neutral plane, secondary combustion, flash-over burning, breaking or opening of doors and windows or the change in direction of a geometry constrained fire plume. A pre-configured control strategy for relaxation parameters would be unlikely to meet the relaxation requirements for all of the events that could happen unless a sufficiently restrictive set of relaxation values were chosen. The problem is that a "safe" restrictive set of relaxation parameters are often far from optimal when considering the complete duration of a simulation and hence there would be much associated wastage of computing effort as demonstrated by the investigations in this results section. The only approach that users of traditional CFD codes have been able to use is to attempt to predict the stages of the development of the fire solution and to revise the control configuration between these simulation stages. Such an approach is prone to error since both the duration of the stages of solution development and the required control configuration are unknown. Only considerable experience, of similar simulations, allows expert CFD users to obtain results reliably and optimally.

7 Preliminary investigations into solution optimisation techniques

7.1 Overview

During the course of this PhD research it has been possible to investigate some areas of interest (for the optimisation of the solution process) that became apparent during the re-engineering of the legacy CFD software. Often these investigations were along the lines of feasibility studies to determine if more research would be needed to exploit new features [EWER99-4].

7.2 Preliminary investigations of group solvers

7.2.1 Overview of groups

In traditional Computational Fluid Dynamics (CFD) based fire models [GALEA89], control of the numerical solver applies equally over all of the cells throughout the solution domain. In large geometry cases this can create a significant, and at times limiting, computational overhead. This is particularly true in cases where the fire occupies a relatively small proportion of an otherwise large solution domain for part, or all, of the simulation period. An example of this may be the early stages of fire growth within an airport terminal or a road/rail tunnel. The group solver concept [EWER99-3] attempts to address this problem algorithmically, by providing optimal processing in regions of the domain where and when it is required.

In the group solver concept, the solution domain is split into an arbitrary number of groups-of-cells. A group is defined as a unique collection of cells that can have solver control parameters independent from any other groups in the solution domain. Group solvers can be activated independently for each solved variable. Internally, the group solver makes use of standard numerical "point-by-point" solution methods such as JOR or SOR [CROFT98].

One way in which this may be achieved is by controlling the number of iterations that the solver

performs in the various groups. For instance, the maximum number of iterations in an "Inactive group" will be considerably smaller than the number for an "Active group". As the solution develops, cells can migrate to and from groups, thus receiving more or less computational attention. The overall convergence criteria are still configured as for conventional problems so there should be no significant difference in the quality of the converged solution.

Group solvers are a novel feature of the CFD component introduced during the software re-engineering. In traditional CFD codes, solver type and control apply to all the cells in the solution domain. Group solvers allow the solution domain to be split into a collection of groups-of-cells. A group is defined as a collection of cells that has its own independent control parameters. A group solver is used for a particular variable on a particular sub-region of the domain. The group solver makes use of standard low-level numerical solution matrix solver methods such as JOR or SOR.

There are several different criteria which may be used to determine the cell groupings. "Geometric groups" have membership with cells grouped by geometric location (e.g. a near wall group, a fire group or a "dead" region group). Such geometric groups are intended to keep their cell membership throughout a simulation. Conversely cells may be dynamically assigned to groups whose membership may change during the solution process. These are so called "dynamic membership groups". The membership assignment process is triggered by pre-configured selection criteria which are dependent on the magnitude of particular variables. For each group, there is a lower and upper value of the trigger-variable(s) which define an acceptance band for membership of that group. When the chosen value(s) in a cell comply with the entry criteria, the cell will be transferred to the matching group.

Typically one could define four base groups for dynamic membership, namely: "Active", "Moderate", "Inactive", and "Void". "Active" has the upper value range for flow or heat, "Moderate" has the medium value range and "Inactive" has the lowest value range. "Void" is used for areas in the geometry that are not part of the flow domain and there is a fixed constant value of the variable (e.g. regions that have been meshed for convenience but are not part of the flow domain for all, or part, of the simulation) that does not require iterative re-calculation.

The main purpose of both types of group and the group solvers is to reduce the overall computation time. This is achieved by directing computational effort only to where it is needed. One way in which this is achieved is by controlling the number of iterations the solver implements in the various groups. For instance, the maximum number of iterations performed in the Inactive group will be considerably smaller than the number performed in the Active group. As the solution develops, cells can migrate to and from groups, receiving more or less computational attention. As the overall convergence criteria are set as for conventional problems, there should be no difference in the quality of the converged solution obtained using this technique.

As the prototype CFD code uses a truly unstructured mesh, there are a limited number of reliable and general purpose numerical techniques available to solve the systems of algebraic equations for each of the primary field variables. Structured mesh CFD codes can exploit the structured nature of the data (e.g. using lines or planes) in various solvers to give more efficient solution than for the point-by-point iterative solvers commonly used in unstructured codes. One of the goals of this work has been to investigate and, if possible, exploit reliable techniques that prove to be of benefit to fire modelling within unstructured mesh CFD codes. One such technique, developed by the author, is the concept of group solvers. A conference paper discussing group solvers is included in the Appendices [See Appendix 11.3].

7.2.2 Description of group solvers

Group solvers are a conceptual extension of the simple linear, iterative, algebraic equation solvers usually referred to as Jaccobi Over Relaxation (JOR) or Successive Over Relaxation (SOR) [CROFT98]. At the most basic level these solvers involve the repetitive update of the solution of a property variable within each cell based on the contributions from nearest neighbouring cells, a portion of the previous solution value and the source quantity for each cell. In a CFD context the contributions from neighbouring cells represent the convection and/or diffusion of a physical property throughout the solution domain whilst the source indicates the creation or destruction of the physical property in the considered cell. The distinction between JOR and SOR solvers is that the SOR always uses the most up-to-date versions of the solution

when calculating the next update. This can make the SOR solver less stable than the JOR solver but it does has the significant advantage of spreading the solution much more rapidly than the JOR.

In the typical whole-domain JOR or SOR solver, the solution in each and every cell of the domain is updated repetitively until the difference between successive updates is sufficiently small. Clearly, if the solution domain contains many cells that are far removed from any active flow region or worse are totally de-coupled from the region of interest for a portion of the simulation, then not all of these JOR or SOR calculations are performing any useful advancement of the solution. This is especially true of many of the large complex geometries used in fire field modelling (e.g. whole building simulations).

The group solver concept allows the domain to be partitioned into "geometric" or "logical" (i.e. solution dependant) groups of cells that then use the iterative point-by-point update described above. The difference for the group solvers is that each group can have a unique set of control parameters to configure the maximum number of iterations to perform, the tolerance to use for convergence testing and/or the linear solver relaxation to be used. In this paper, the investigation only concerns the potential benefits of limiting the number of iterations that are used within each group of cells - while maintaining the desired level of convergence.

Since, in an unstructured code, a group does not need to be limited to some pre-configured geometric region it is possible to further extend the group solver techniques by allowing groups to determine their own cell-membership as the solution develops. This has been implemented within SMARTFIRE to allow an arbitrary number of groups which can contain either geometric or solution dependant membership (provided that each cell only exists in one group) and that furthermore the dynamic groups can exchange cells as the simulation solution develops. In practice, the dynamic membership is configured so that each dynamic group has an acceptance range of values which will trigger a non-member cell to be transferred into that group if its property value is within the configured range and that the cell is not already contained in a static "geometric" group.

The implementation of the group SOR solver requires particular care, at the algorithmic level, to ensure that the groups are not de-coupled into JOR connectivity between groups. This scenario is possible if the looping between group-inner-iterations and between groups is mismanaged to give simple external looping for all groups and internally for each group to loop for all configured inner-iterations. There are several possible methods of handling the inner looping which give different updates for cases where groups have different numbers of configured inner iterations. It was decided to interleave the processing between groups without using a simple 1:1 interleave ratio, which would have been easier to implement but possibly less efficient. The more complex interleaving technique causes each group to be visited in turn and performs one (or more) of the inner iterations before moving to the next group. The looping amongst groups continues until each group reaches its configured maximum number of inner-iterations or until convergence is detected.

In order to attain maximal optimisation for cases with truly de-coupled (and hence uninteresting) group regions, it was also necessary to limit processing of such groups so that simple calculated variables are not updated. Mostly there is little difficulty in performing this optimisation because the support variables are generally closely linked in their usage to associated solved variables.

It should be noted that many of the variables in a fire modelling simulation have a definite "directionality" that can be exploited by matching the marching order of the cells within SOR solvers with this direction. The prototype CFD engine has been implemented to use bi-directional marching order for all SOR type solvers, which gave a saving of up to 20% over the usual unidirectional marching order - when used on the simulation case described in this section. All of the timings compare bi-directional group and whole-domain SOR solvers.

7.2.3 Investigation of geometric groups

Several examples were used to investigate the use of geometric group solvers for a partitioned 2-D room fire simulation. The simulation timings, for geometric groups, were all performed

using a 90 MHz Pentium PC with 64MB RAM. The first example involves a single compartment with two doors. Both doors open to the outside and hence involve two extended flow regions. The second example involves a similar compartment in which one door opens to the outside while the other door opens to a second closed compartment. For simplicity, all confining boundaries are assumed to be adiabatic. In both cases a small volumetric fire source of 50 kW is situated in the centre of the fire compartment.

In the first example, one of the doors is open throughout the simulation while the second door is opened 40 seconds into the fire simulation. The solution domain is thus made up of three distinct regions, the first external region outside of the open door, the fire compartment itself and the second external region beyond the closed door. The computational mesh in each region comprises of 8 x 21 cells, 22 x 21 cells and 8 x 21 cells respectively i.e. a total of 798 cells. Using standard CFD solution techniques the solvers operate equally in all of the cells throughout the solution domain, even the cells in the initially dormant external region beyond the closed door. This is clearly a waste of CPU time as nothing of significance occurs in the external region beyond the closed door.

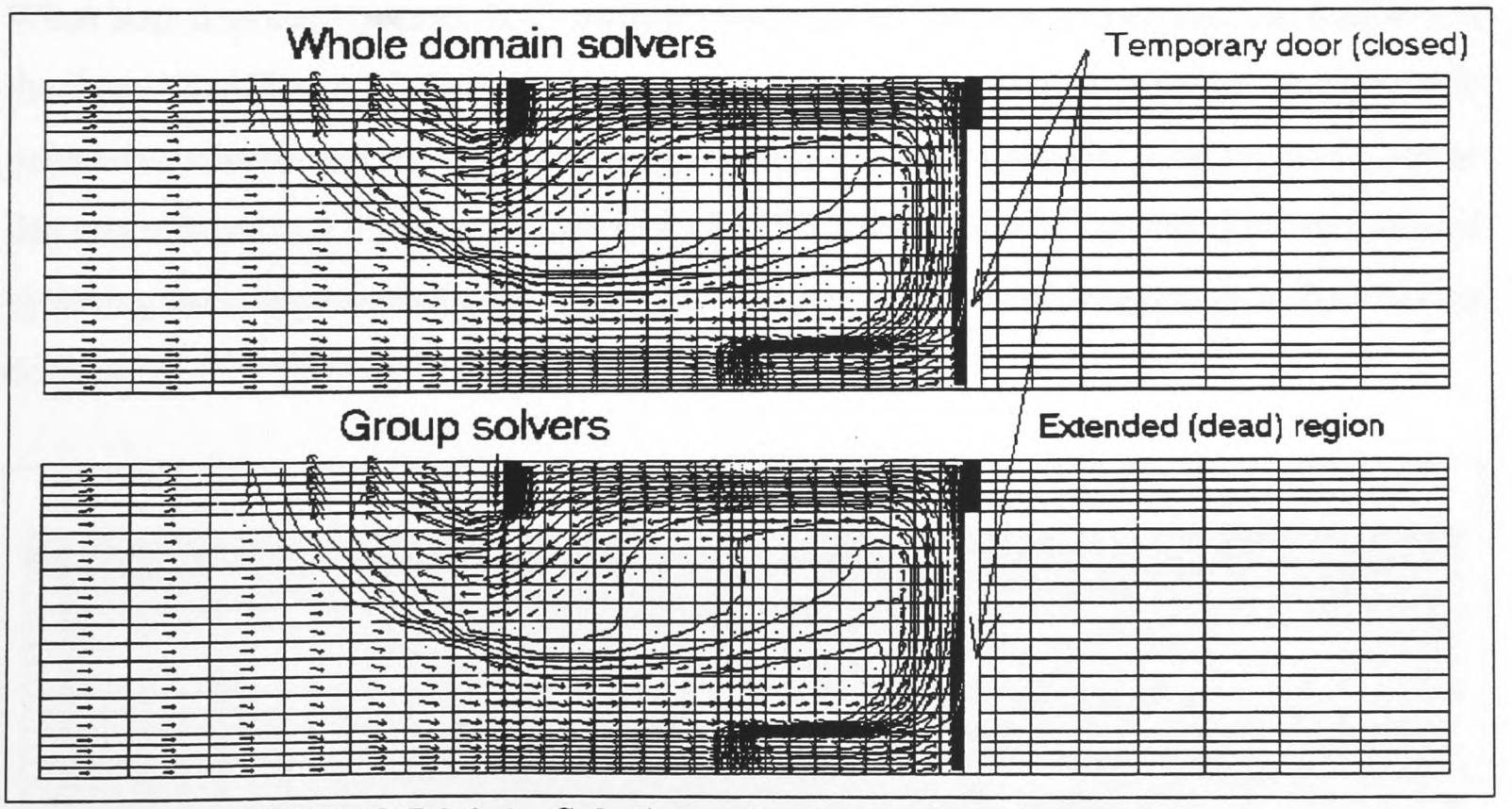


FIGURE 7.2.3-1 : Solution prior to opening of second door obtained using conventional and groups solvers.

Using the group solver, the initially "dead" region is marked as Inactive resulting in the solver spending a minimum amount of effort in this region. When the second door opens after 40 seconds, the Inactive region changes to Active status and the solution domain extends to cover the second extended region. As demonstrated in the figure (See Figure 7.2.3-1), the solution just prior to the second door opening when the group solver is used is identical to the solution when the conventional solver is used and it is concluded that there is no loss of accuracy.

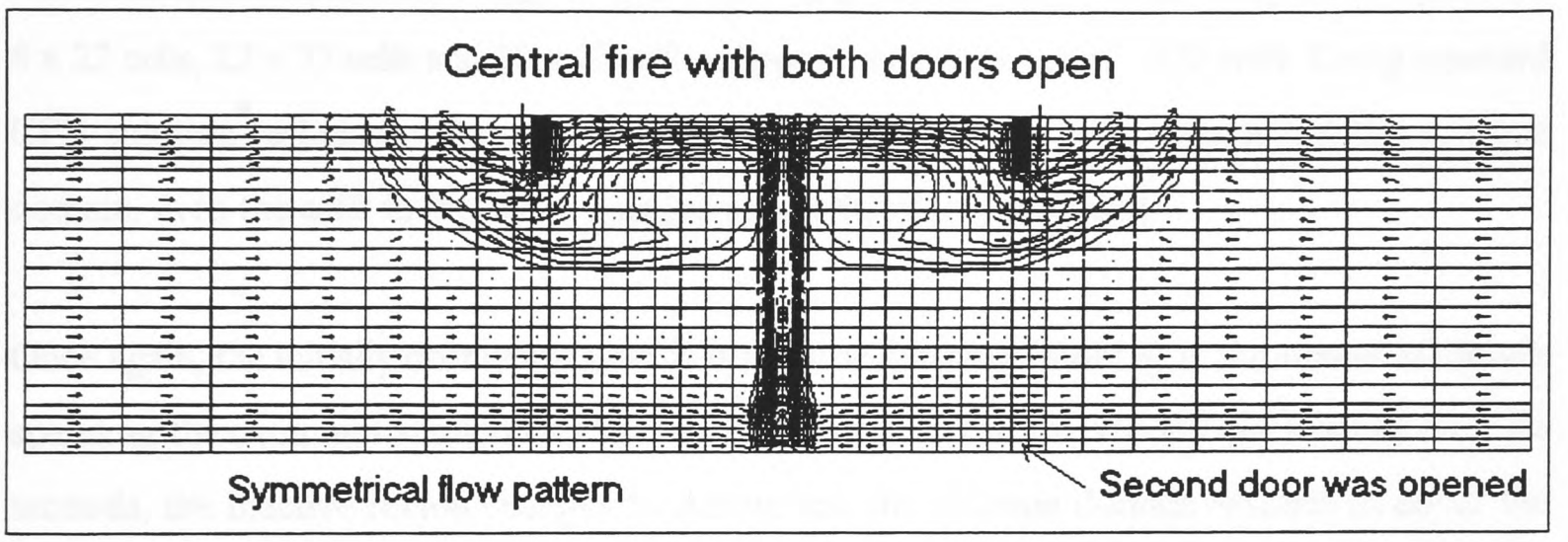


FIGURE 7.2.3-2: Steady-state solution obtained after both doors are opened (example 1).

When both doors are opened, both methods converge to the steady state solution depicted in the figure (See Figure 7.2.3-2). However, using the conventional solver, the run time up to the point where the second door opens was approximately 3.02 hours while using the group solver this was reduced to 2.72 hours, a saving of 10%. While only a modest saving, this was achieved by saving the computational effort over only a comparatively small proportion of the solution domain.

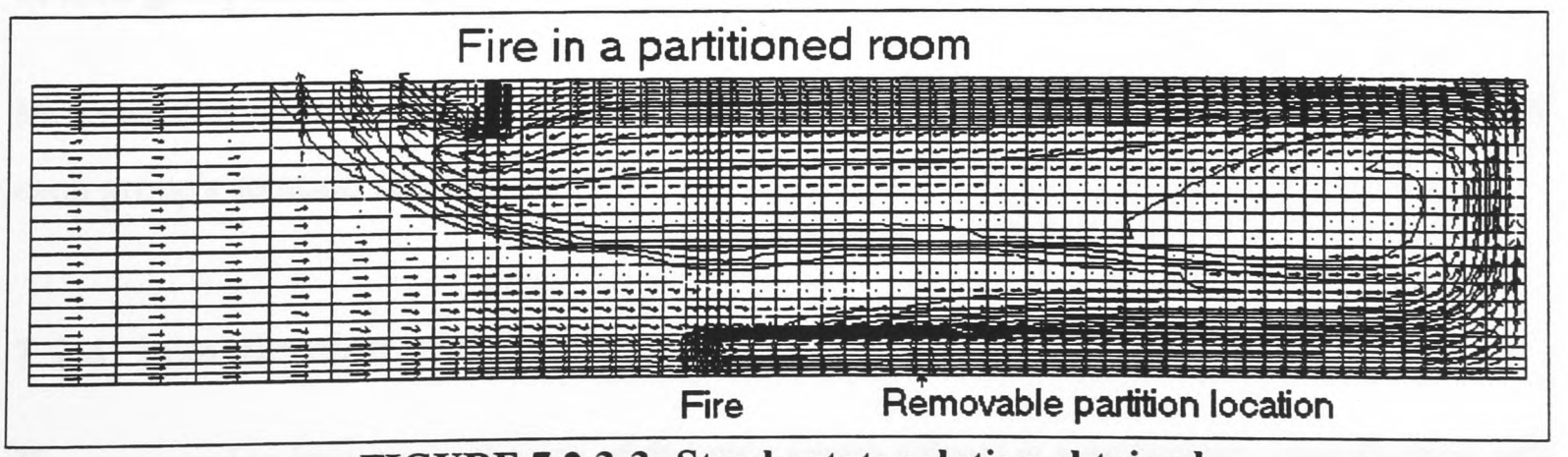


FIGURE 7.2.3-3: Steady-state solution obtained after both doors are opened (example 2).

When Inactive regions occupy a greater proportion of the mesh savings in computational time can be improved significantly.

In the second example, the second door is opened after 40 seconds but rather than opening to the outside it opens into an otherwise sealed compartment. The solution domain again consists of three distinct regions, the external region outside of the open door, the fire compartment and the second, initially sealed, compartment. The computational mesh in each region comprises of 8 x 22 cells, 22 x 22 cells and 30 x 22 cells respectively i.e. a total of 1320 cells. Using standard CFD solution techniques the solver operates equally in all the cells throughout the solution domain, even the cells in the second sealed and isolated compartment.

Once again, the initially dormant region is marked as Inactive resulting in the numerical solver spending the minimum amount of effort in this region. When the second door opens after 40 seconds, the Inactive region changes to Active and the solution domain extends to cover the second compartment. As in the previous case both solution techniques result in identical solutions prior to the opening of the second door. When both doors are opened, both methods converge to the steady state solution depicted in the figure (See Figure 7.2.3-3). However, using the conventional solver, the run time up to the point where the second door opens was approximately 4.40 hours while using the group solver this was reduced to 3.06 hours, a saving of 31%. Thus, by effectively reducing the computational domain by 50%, a saving in computational time of 31% is achieved. This saving is less than might have been expected but it is explained by the need for initialisation and property updates that occur in all cells regardless of their group membership.

More work is needed in this area to determine if there are benefits for the dynamic control of each group's solution controls.

7.2.4 Investigation of dynamic groups

The case used to investigate the use of dynamic group solvers is a preliminary investigation into

fire spread between the floors of a multi-storey building where window sizes are varied to modify the ejected plume behaviour. This case is loosely based on some collaborative research with LPC into fire spread between floors [GLOCKING97]. In the case presented here only the lower (ground) floor room is modelled together with the outer wall of the second and third floors above. In subsequent research it is intended that the upper floor rooms will also be fully modelled with windows that can be broken by the incident heat flux from the ejected spill plume.

In order to investigate the benefits of the group solvers a number of test cases were prepared. The geometry and mesh used in all of the tests was identical and great care was taken to ensure that the mesh was sufficiently refined across the height and width of the window, near the walls of the room and outside and just above the window. These considerations are critical to obtaining a reliable and accurate simulation of the ejected plume.

The geometry (See Figure 7.2.4-1) was set up with room dimensions of 4.0m (x) X 3.4m (y) X 6.0m (z). The centrally located fire is represented as a volumetric heat load which is applied over a volume of 1.0m X 1.2m X 2.0m. The fire uses the so-called "alpha t squared" power curve, which reaches 2.0 MW (using a fast growth rate) in three minutes of simulated time. This is a commonly used growth rate for representing real fires (for example burning furniture) with a volumetric heat output. The window aperture has a size of 2.0m (y) X 2.0m (z) and is centrally located on the high X-face of the room. The exterior wall, above the window, extends for a height of 10.5m vertically. This extended height is intended to allow for the addition of two open rooms above fire room and a further room height to move the free surface boundary sufficiently far away from any upper floor windows that may be used. This positioning of the free surface is necessary to prevent outlet effects from dominating the flow in any critical region of the flow domain where it might change solution.

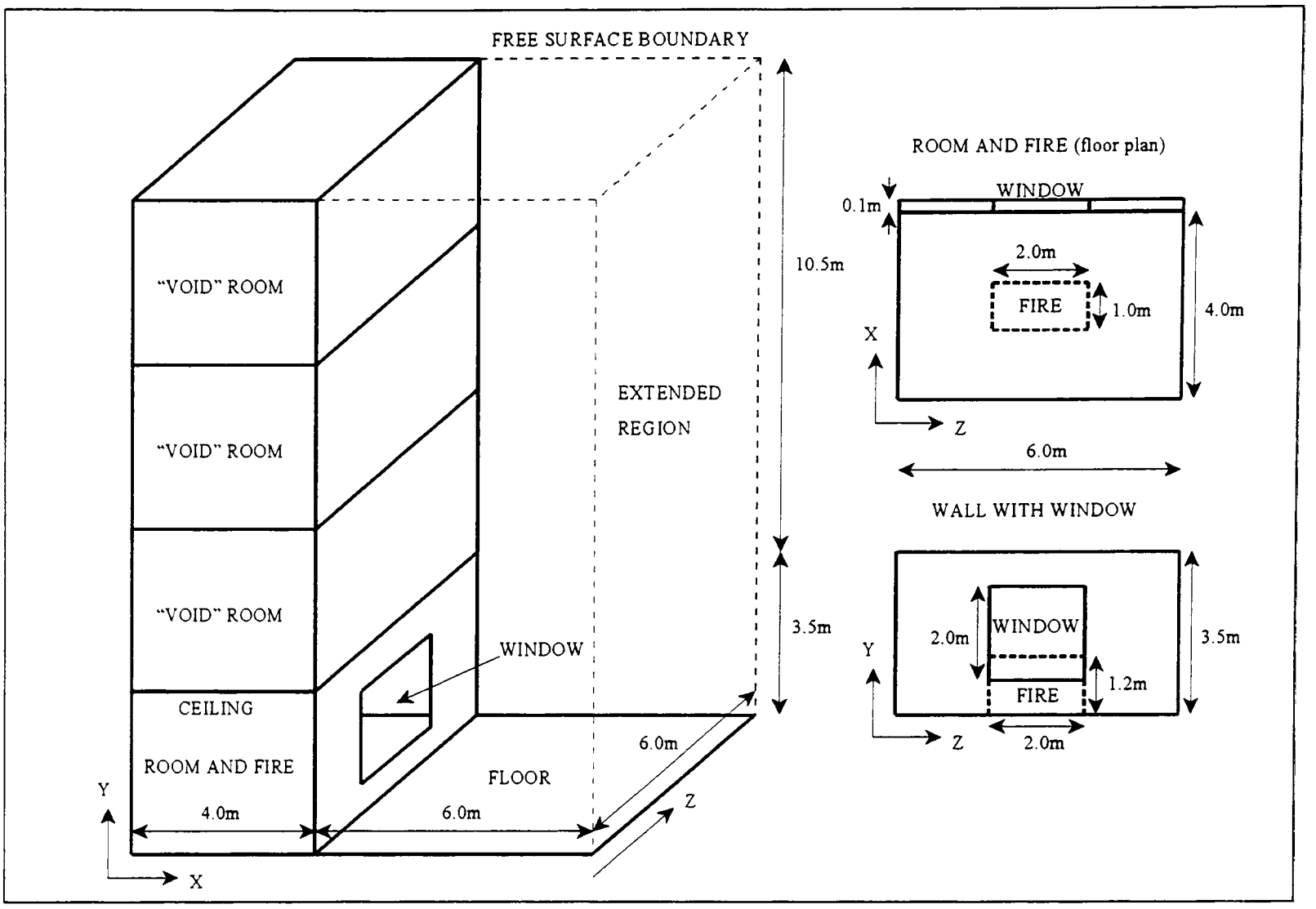


FIGURE 7.2.4-1 : The multi-storey geometry used for the group solver tests.

The extended region beyond the window has the same Z-width as the room and extends for a distance of 6.0m in the X-direction in order to give ample room for the plume ejection. All of the surfaces of the extended region have a free surface boundary condition except for the floor, which is assumed to be solid.

The outside region is assumed to be calm prior to the fire. The walls are assumed to be brick with a thickness of 0.1m.

The mesh used for the simulation consisted of 40,572 cells with NX=36, NY=49 and NZ=23. The number of cells in the geometric regions was as follows: Dead region (non participating rooms above fire compartment i.e. de-coupled region) has 14,260 cells, Fire-room has 8,280 cells and the entire extended region has 18,032 cells (See Figure 7.2.4-2).

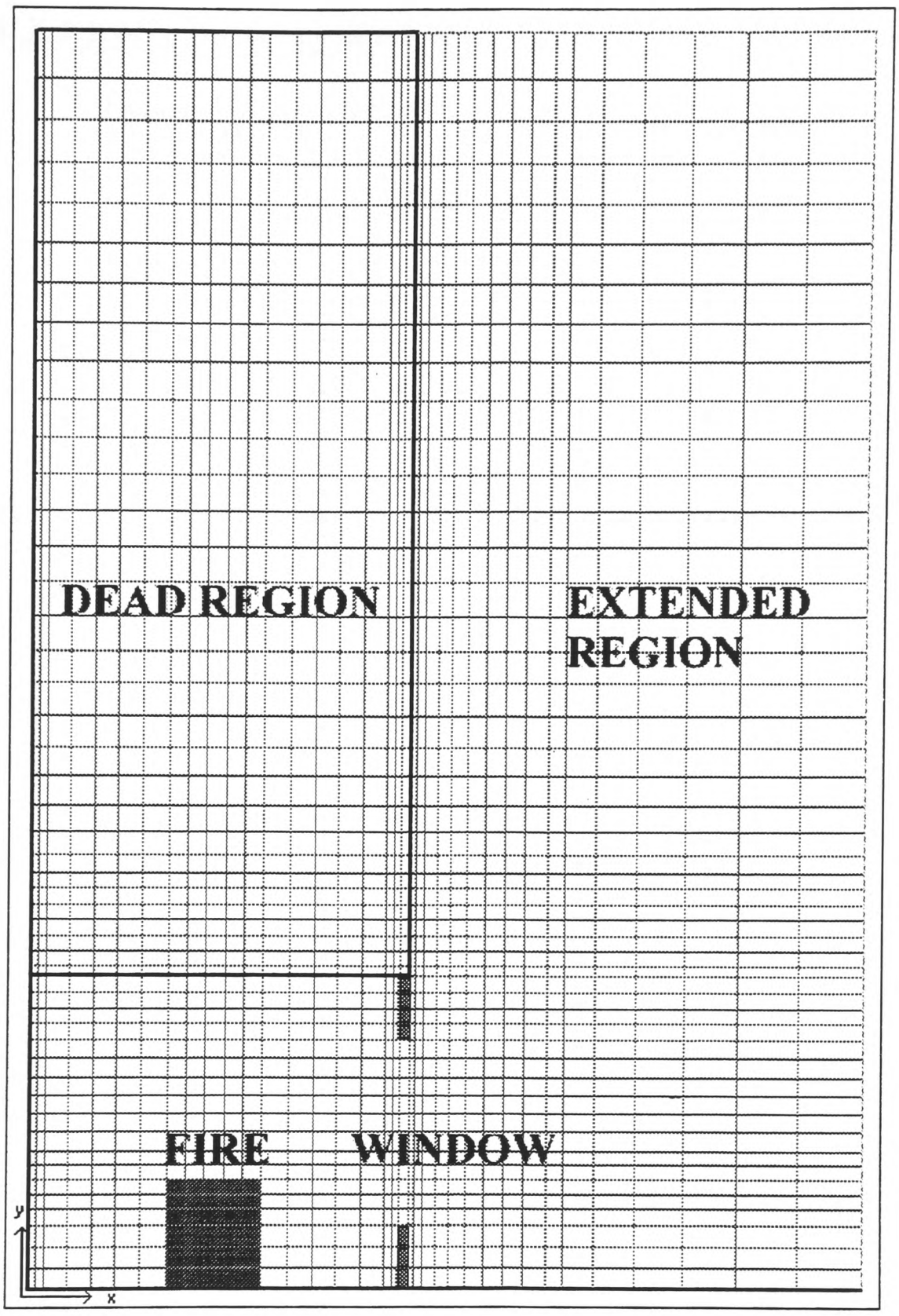


FIGURE 7.2.4-2 : Vertical slice through the domain showing the mesh and the various regions.

The simulation involves buoyancy driven flow with K-Epsilon turbulence model (buoyancy modified) and incorporates the six-flux (enhanced) radiation model as described in the User Manual [EWER99-2]. The entire simulation was configured to perform 90 time steps of 2 second duration. The solver configurations used in the various simulations are summarised in the following table (See Table 7.2.4-3).

TABLE 7.2.4-3 : Summary of solver configurations used in simulations.

Variable(s)	Solver update method	Whole domain iterations	Active group iterations	Calm group iterations	Void group iterations
Pressure	SOR	50	50	12	0
Momentum	SOR	6	6	2	0
Turbulence	SOR	20	20	5	0
Enthalpy	SOR	30	30	8	0
Radiation	SOR	20	20	5	0

Furthermore all solvers were able to terminate their inner iterations if a common convergence level was reached. Each time step was forced to have all normalised variable residuals converged, to 1.0e-03, before the next time step could be started.

For comparison purposes, the following three test cases were simulated:

Case 1: The simulation is configured with all solved variables using the whole domain SOR solvers as specified in the table (See Table 7.2.4-3). For comparison purposes this constitutes the base case. The group solvers are not utilised in this test and so the code is run in a conventional manner.

Case 2: The entire solution domain is configured into two static "geometric" groups, one group configured as a "Void" group and another configured as an "Active" group (See Table 7.2.4-3). The "Void" group contains all of the cells in the de-coupled region above the fire room (i.e. 14,260 cells or 35.2% of the entire cell budget). The "Active" group contains all of the cells that are not in the "Void" group region (i.e. 26,312 cells or 64.8% of the entire cell budget). While the group solvers are activated, group membership remains the same throughout the simulation.

Case 3: The entire solution domain is partitioned into four groups, two are static "geometric"

groups and two are "dynamic" membership groups. The first group is a static group that is configured as a "Void" group which contains all of the cells in the de-coupled region above the fire room (i.e. 14,260 cells or 35.2% of the entire cell budget). The second "static" group is configured as an "Active" group and contains all of the cells in the fire room, those in the window aperture and a small rectangular block of cells that is immediately outside of the window (uses room with 8,280 cells and 2,366 cells from the extended region i.e. 10,646 cells or 26.2% of the entire cell budget). The third group is "dynamic" and "Active" and is configured to determine cell membership from the non-static cells of the extended region. The group membership selection criteria is for absolute cell velocity being greater than 10% of the maximum domain velocity. The fourth "dynamic" group is configured as a "Calm" group and contains extended region cells that have an absolute velocity of less than 10% of the maximum domain velocity. The two active groups share the remaining 15,666 extended region cells or 38.6% of the entire cell budget. Dynamic group membership is updated every 10 sweeps.

For the purposes of this paper, timing comparisons based on the first 50 time steps of each test will be presented. On the test computer (a Pentium II 400MHz with 256MB of RAM) this gave a convenient processing duration that could be run overnight without interruption.

TABLE 7.2.4-4 : Comparison of group solver performance over the three test cases.

Test scenario	CPU time used for 50 time steps	Total number of sweeps used	Percentage time saving over case 1
Case 1 : Whole domain solvers	15h 51m 40s (57,100 seconds)	3095	0.0%
Case 2 : Static groups	11h 43m 36s (42,216 seconds)	3089	26.1%
Case 3: Static and dynamic groups	9h 56m 45s (35,805 seconds)	2919	37.3%

Of primary interest, to this study, are the potential gains in numerical efficiency generated by the use of group solver technology. It should be noted that all three test cases produced practically

identical solutions with the same levels of convergence. A comparison of the run times for the test cases is presented in the table above (See Table 7.2.4-4). Clearly, the group solver has potential for introducing considerable savings in computational time.

The fire dynamics in these test cases proceeded as expected. As the window opening to the fire compartment was considered narrow, a strong plume was ejected from the compartment. As the plume rotated and ascended vertically, it did not attach to the building façade. These results are consistent with earlier modelling work [GALEA96] and with reported experimental observations [YOKOI60].

By 100 seconds, of simulated time, the rising plume outside of the compartment was fully developed and clearly unattached from the building façade. Continuing the simulation beyond this point merely increased the temperature of the fire compartment, the rising plume and the building façade.

The results for temperature displayed in the figure (See Figure 7.2.4-5) were taken at a simulation time of 120 seconds from the whole field SOR simulation in Case 1. Only the results from Case 1 are presented here as the comparable results from Case 2 and Case 3 displayed no apparent differences. Within the solution fields produced by Cases 1-3, maximum temperatures differed by at most, 1 Kelvin in the range of 318 to 914 Kelvin.

In order to verify that the dynamic group solver membership mechanisms were operating as expected, a vertical slice visualisation of group membership was created. This group visualisation (See Figure 7.2.4-6) shows that the "active" dynamic group in the extended region has captured the plume extent correctly.

The static group solvers used in Case 2 demonstrated that, by effectively removing 35.2% of the domain from the computations, a saving of processing time of 4h 8m 4s (or 26.1%) was obtained when compared to the standard whole field SOR solvers processing all cells equally.

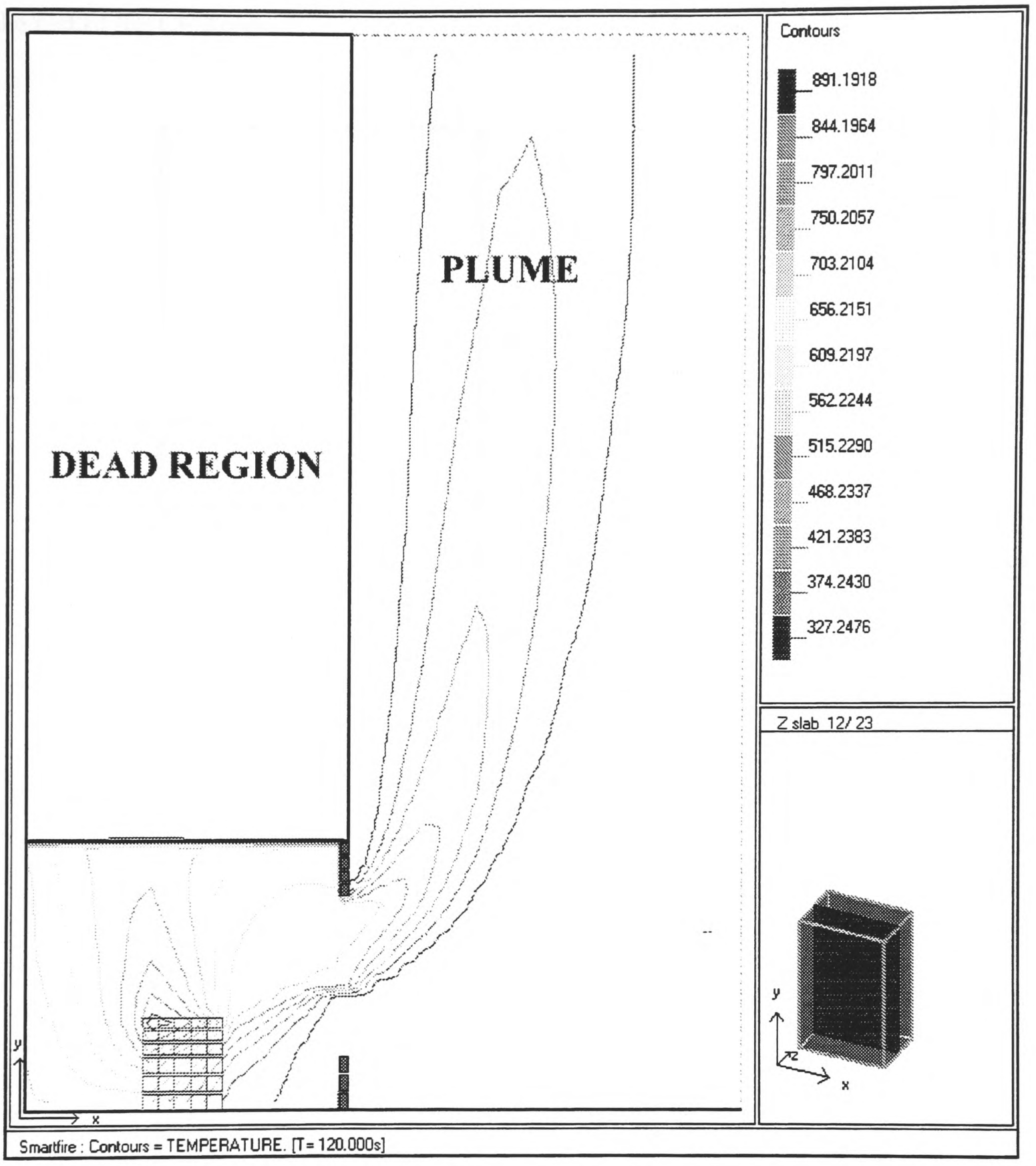


FIGURE 7.2.4-5 : Vertical slice showing room and plume temperatures (K) at 120 seconds.

In effect this indicates that the group solvers were 74.2% efficient at removing the processing overhead of the de-coupled region from the simulation. While a 100% efficiency may be desired, this result was anticipated because there are still many calculations performed in the "de-coupled" region for material properties and simple calculated variables. It is anticipated that this figure can be improved somewhat by increasing the use of "group" activated calculations within

the rest of the CFD code.

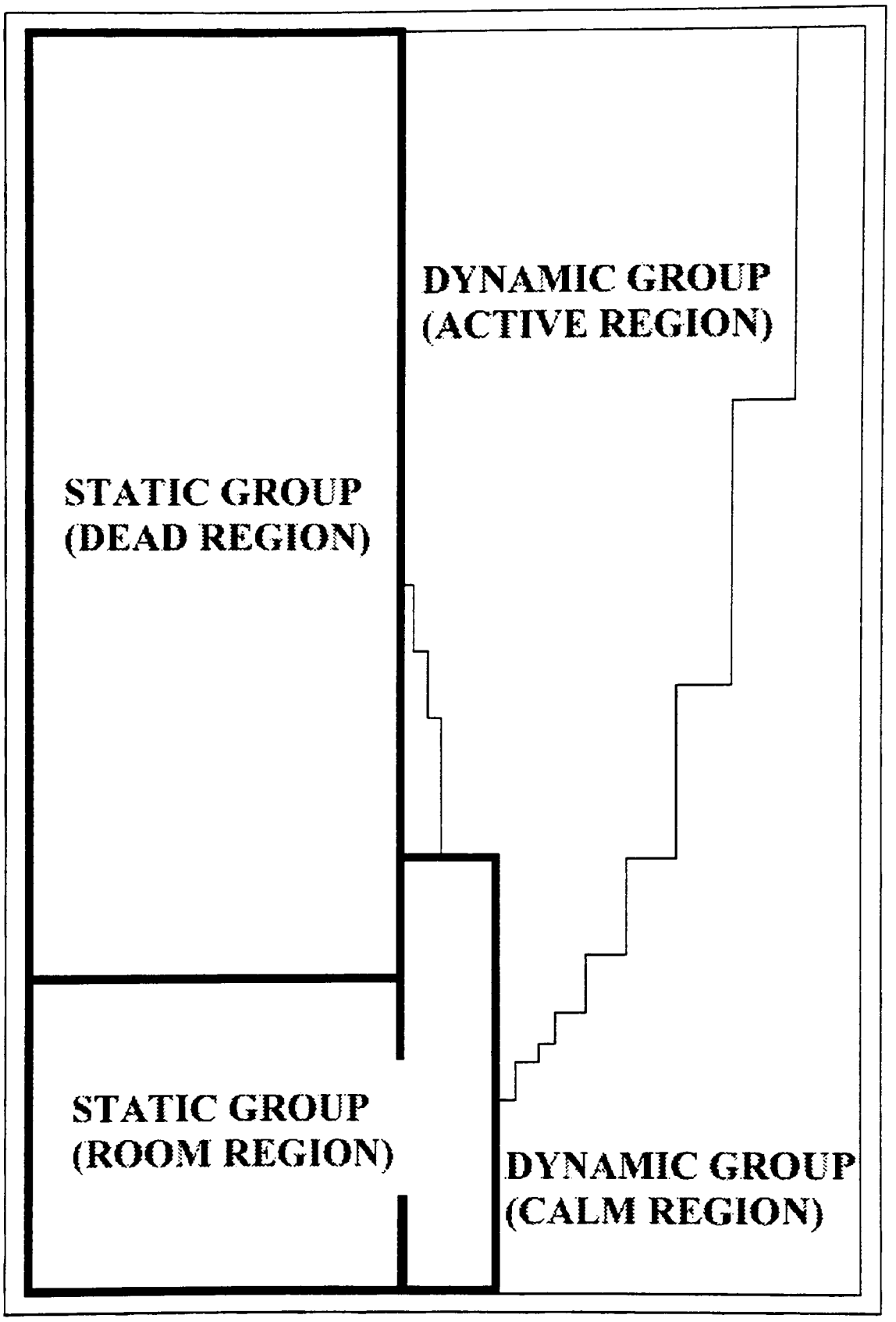


FIGURE 7.2.4-6 : Vertical slice showing static and dynamic group membership at 120 seconds for test case 3.

In Case 3 both static and dynamic groups are used with the majority of the extended region being continuously evaluated for applied processing strategy. In this case an overall processing time saving of 5h 54m 55s (or 37.3%) was achieved when compared to the standard whole field

SOR solvers processing all cells equally. It should be noted that much of this saving is due to the "de-coupled" void group which, as shown in Case 2, saves 26.1% of the processing. The remaining 11.2% saving is due to the optimisation of processing within the extended region which targets less solver processing in cells with relatively low velocity flow. The fact that this saving is comparatively less than for the "de-coupled" region is also anticipated. This can be explained by considering the work performed in the "de-coupled" and dynamic groups. In the "de-coupled" group, it was not necessary to build the system matrix coefficients for the member cells whereas any cell in a solved group that performs one (or more) iterations must build the system matrix coefficients in order to perform any calculation. Building the system matrix coefficients is relatively costly compared to solving the matrix.

The results indicate that there are large potential savings to be gained in the simulation of fire modelling scenarios by the targeting and optimisation of processing effort in fully de-coupled, suitably stratified or geometrically related flows. Furthermore, these savings need not result in compromised accuracy of the final solution. The techniques developed and presented here resulted in considerable run-time savings of up to 37% of processing time. It is anticipated that this figure can be improved significantly when a better understanding of the balancing required between groups and variables is achieved.

As group solvers are a new concept, there was little or no expertise to guide in the optimal selection of number of groups to use, the choice of group membership conditions and the relative amounts of processing used in each group. Furthermore there are a number of remaining group solver control options which were not varied during the test simulations.

It is anticipated that in large scale simulations, which may involve whole buildings, there are likely to be much greater savings possible with intelligent use of group solvers that can target the processing only on the active flow and fire regions until the solution characteristics in other regions become significant.

Current research efforts are directed at gaining a better understanding of when it is appropriate

to use groups and how best to balance the processing between groups in order to obtain optimal convergence and simulation times. Dynamic groups have been shown to give modest performance improvements but more work is needed to determine if there are any further benefits possible due to combined solution monitoring and dynamic knowledge based control of the processing within both the static and dynamic groups. Whilst the use of group solvers increases the complexity of the knowledge based control it is also most likely to provide the most significant savings and most reliable solutions.

7.3 Preliminary investigation of automated dynamic solution control

A prototype dynamic control module was developed to investigate the potential for automating the process of dynamically monitoring and controlling the solution of a particular class of Fire simulations [EWER98]. (See Section 6.6.2 in the Results Chapter 6) The module was quite primitive because it only monitored the local convergence behaviour and then only modified linear and false time step relaxation values.

The production rules that were used to effect these control changes were demonstrated to be quite good for 2-dimensional fire scenarios but not sufficiently flexible to handle more complex 3-dimensional fire scenarios.

It has been established that significant savings in run-times can be achieved when the automated solution control module can fine tune the relaxation parameters for optimal convergence but there is some danger of de-stabilising the solution when compared to using a safe set of initial relaxation parameters. This was demonstrated by the fact that a fairly simple 2-Dimensional scenario gave good savings for optimisation of relaxations but a more complex 3-Dimensional scenario was destabilised by the solution control module. It is predicted that these problems are caused by the lack of monitoring of persistent trends in the solution convergence behaviour. The prototype solution control module was only evaluating convergence trends based on a few preceding simulation sweeps.

8 Conclusions

8.1 Benefits of interactive control

This research has demonstrated significant and tangible benefits for the use of interactive control and monitoring user interaction techniques for use with CFD simulations in the Fire Field Modelling arena. These benefits are not solely limited to performance enhancements because solution reliability, error detection and algorithm development have also been demonstrably improved during the investigations. Although the re-engineered and interactive prototype CFD system runs more slowly than the original legacy code this should be seen in context of the tangible improvements in the overall performance due to time saved by using solution optimisation and the reductions in time wasted in unsuccessful simulations. There is also the added benefit that the prototype CFD system gives more assurance of solution correctness when the path to the solution has been monitored.

The need for interactive control has also been demonstrated by the relatively poor attempts at static simulation configuration obtained from CFD researchers during this investigation. This tends to suggest that some form of interactive control is necessary for CFD experts to be able to transfer their knowledge between CFD codes so that they can use new simulation software correctly and optimally. It is also beneficial for expert CFD users to have immediate access to the intermediate solution status information so that simulation problems can be detected promptly and control actions planned and imposed as required.

One particularly surprising observation made during this research concerned the lack of transferability of set-up knowledge between different CFD codes by expert CFD users who were familiar with another CFD code. This meant that even users who have considerable expertise with using a particular CFD code cannot always easily transfer that knowledge for the optimal or, in some cases, reliable use of another CFD code. Generally speaking, this means that all users are forced to learn the idiosyncrasies of each new class or instance of a CFD code. It is not clear if this problem is quite so marked or critical in other application areas outside of CFD research

or outside of numerical simulation. Clearly the problem is alleviated somewhat by having a good user interface with appropriate monitoring and solution statistics that allow the user to check that the CFD code is behaving predictably, acceptably and optimally.

The implementation of reliable simulation control automation relies on the availability of high quality expertise and knowledge obtained from Expert users controlling real simulations. Formerly this knowledge has not been available because traditional approaches to CFD used batch mode processing techniques followed by post-processing results analysis. It has been demonstrated that this often leads to highly non-optimal simulation strategies and errors in the simulation are found (if detected at all) after a simulation has been run completely. Knowledge about reliable solution control methods will only be obtained through the use of interactive control techniques by CFD experts and further research in this area. As a corollary, it should be noted that an interactive control and monitoring interface is a significant benefit for teaching purposes. When used as a teaching tool, an interactive CFD code gives trainee Fire Field modellers a much better understanding of the internal processes and limitations of CFD software and is thus likely to enhance their productivity and accuracy of simulation

The current trend for the use of Fire Field modelling by building designers, fire regulatory authorities and others who often have quite limited CFD experience (or in the case of Firemen who have recently started to investigate Fire Modelling techniques, almost no CFD experience) which means that it is more important than ever for reliable, informative and (whenever possible) automated simulation software to be made available to ensure that simulations that are conducted for risk assessment, design planning and performance analysis are conducted correctly and that the results are only used or presented if they are reliable.

8.2 Benefits of incremental reverse engineering

The reverse engineering methodology developed during this study has wider implications when one considers the huge amount of extant software that was written in older computer languages and often without the benefit of modern software engineering practices. The success of the methodology lies in the fact that the software being re-engineered is never far away from a state

that can be compiled and re-validated for functional consistency with the original legacy software. Clearly it is necessary that the validation cases selected should provide suitably wide function coverage but, with this proviso, it is possible to impose many modern principles of software design and ease of maintenance without an overwhelming expenditure of human resources.

In practice, it would have been more efficient to use more automation (i.e. translators or compiler writing tools) during the translation phase of the re-engineering. It is also not easy to predict how well the techniques used in this investigation would be transferable to other application areas but the principles of maintaining absolute functional consistency do make a great deal of sense no matter what the origin and purpose of the legacy code.

8.3 CFD Research benefits of using an open architecture and Object Oriented development techniques

Many of the enhancements, capabilities and possibilities for further research have only been practically made possible due to the open architecture imposed on the re-engineered CFD software and the use of Object Oriented implementation techniques. This does not mean that traditional implementation techniques (i.e. procedural, structured coding) make such enhancements impossible but there is a point where procedural implementations with their poor code clarity, primitive data passing, simple data storage mechanisms and monolithic software architecture tend to hamper further development. A well-structured Object Oriented architecture can give considerable assurance that the methods for one type of object will not interfere with other types of object and their respective implementations. This was a definite benefit with regards to the implementation and development of the visualisation, knowledge based control and group solver functionality.

The use of Object Oriented development techniques have also been observed to promote ease of adaptive and perfective maintenance. The fact that new algorithms can be implemented without having a knock on effect through the rest of the software considerably simplifies the scope of any new developments and helps to ensure implementation correctness.

8.4 CFD Research problems caused by the use of Object Oriented development techniques

There were, almost inevitably, some problems facing the use of Object Oriented development techniques with CFD research.

The first observed problem was that there were multiple clear conceptual or real world objects that could be used as the basis of the data architecture. The eventual object abstraction that was adopted came down to a simple preference for a favourite object hierarchy. The re-engineering development described in this research may have gone very differently if, at the key stage of formulating an object hierarchy, it had been decided that data vectors and system matrices were the best choice of conceptual objects.

The second observed problem is that Object Orientation tends to make the developer write more and simpler functions than would have been present in the original procedural code. The problem here is that adding additional layers of function calls at a low level of looping within the numerical code is likely to be accompanied by a considerable performance overhead for the additional calls. Matters would be even worse if dynamic function resolution (i.e. late-binding or run-time) is used because of Object Oriented inheritance and overloaded functions. This is typically seen where one type of object is a sub-type of a parent object type and both object types have identical methods which means that the compiler cannot resolve which method to call and it must be evaluated at run-time. The re-engineering described in this investigation tried to limit this increase in the number of layers of function calls, but it was not completely successful. The re-engineered system has, on average, at least one extra layer of function calls at a relatively low level of the code and this has quite a large impact on run-time performance. The development has managed to keep late-binding to very high levels of the code where it will have negligible effect on the overall performance.

The third observed problem is that the Object Oriented language used for this research (i.e. C++) and most other object oriented languages are somewhat slower than the FORTRAN language. This is partly due to the difficulties of optimisation in C++ and the simplicity of FORTRAN.

There is also the performance impact, which has not been evaluated, of the organisation of the large amounts of simulation data in memory and the memory address "jumping" required by the processor to perform the calculations. The original FORTRAN code had long arrays of data whereas the Object Oriented C++ version of the code has cell objects (with complete sets of internal data) that are consecutive and adjacent in memory. It is not clear what overhead this has in terms of the jumping necessary when addressing data and the performance hits that this will cause due to the required frequency of cache updates. Unfortunately there was insufficient time to perform an analysis of the impact of the techniques used for data storage.

A final problem facing the prototype system is that of user acceptance. The fact that the system has been written in object oriented C++ will mean little to the majority of users who will only ever make use of the executable form of the software to perform their simulations. Conversely users and developers who need to write additional routines may find the conceptual structure of the Object Oriented version of a CFD code too alien to their experiences of procedural development. Every effort has been made to mitigate these problems by avoiding the more obscure syntax, constructs and mechanisms available in C and C++ but some of the idiosyncrasies remain. Possibly the worst of these is the slightly strange syntax needed to access any cell data value which in the re-engineered system appears as

data = cell[cell_index]->access(NEWEST,TEMPERATURE);

whereas the original legacy code would have used the simpler syntax

data = TEMPER(cell_index)

but, it is argued that, the benefits of the re-engineered system described in this investigation far outweigh this fairly minor inconvenience.

9 Further work

9.1 Overview

During these investigations there were many instances of research areas that needed investigation or that could be potentially beneficial but these could not be pursued due to time and resource constraints. This section describes areas of research that need further work.

9.2 Dynamic solution control

Further work is needed for better determination of the current solution status (possibly involving more historical solution status information) so that the control rules can be fired in a more reliable way. Also the actions that can be performed need to be extended to cover more of the capabilities of the CFD software. It is possible that the Dynamic control module would benefit from initially using a "Zone model" to determine a "quick and dirty" solution prior to performing the simulation and to use this data to apply changes to the number of iterations, tolerances, solver types and time step size as required.

Furthermore the dynamic control module needs to be "aware" of transitional effects such as the re-direction of plumes or flow jets by geometry constraints or the change in height of neutral plane because these changes can greatly effect solution stability and often require special handling. There is also the potential for user or solution defined transitional effects such as the breaking of windows, opening of doors and secondary fire ignition or fire spread that will also require specialised control handling.

9.3 Visualisation

Visualisation needs to be extended to 3-D for improved run-time assessment of the solution status. This should not be regularly updated, as it is in the current 2-D slice visualiser, because

of the performance overhead that would be incurred for 3-D display formation. A more global approach to investigating the solution data, at any stage of the processing, is needed so that solution features are not missed. Quality data visualisation using contour nets and vectors are recommended though it may prove necessary to use transparencies and cutaways in order to prevent foreground data from obscuring background data. Streamlines can also be used to give visual meaning to flows.

The fire field modelling area is likely to benefit from recent visualisation techniques such as "fogs" and Virtual Reality in order to help bridge the gap between the fire field modeller and designers or non-CFD specialists who need to comprehend and make use of the simulation data.

9.4 Pattern matching for KBS control and status reporting

It is anticipated that some means can be found to detect recirculations and other flow formation events in order to fine tune the simulation controls to give reliable but still optimal numerical handling. This may need to implemented, at least initially, with simple status monitors and questioning of the computer user who will be used as a non-specialist expert for the visual classification of a data visualisation. This would be used where numerical or other programmatic means of classification of simulation features are non trivial or prove to be unworkable.

Work is already progressing into more reliable and comprehensive techniques for solution status reporting based on the analysis of residual graphs. This new type of analysis attempts to classify convergence trends and behaviour over time and over successive sweeps so that qualitative decisions can be made about how the simulation can be optimised and to monitor for potential problems so that they can be handled before ever becoming critical (i.e. solution threatening).

9.5 Enhanced physics and numerical methods

There are a number of diverse techniques that need to be added to the prototype system in order

to provide a comprehensive tool for fire field modelling scenarios. The most important of these features are toxicity modelling, pyrolysis and solid fuel combustion [COX95], flash-over modelling [JIA97], discrete transfer model for radiation, fire spread modelling and secondary ignition. These features are by no means trivial and some are at the forefront of current research [JIA99].

9.6 Exploitation of parallel processing architectures

It is interesting to note that the Object Oriented data structures used in the prototype system could be used to provide a fairly simple exploitation of parallel processing architectures. Since each cell is a complete and separate entity and the fact that group solvers and dynamic group membership have already be demonstrated as workable, it is possible that a parallel implementation of the code could be formulated with relatively little additional programming. There are several areas that would need to be addressed i.e. the addition of halo cell regions about each group and the scheduling of inter-processor data updates. Where this implementation may really benefit is from the potential for dynamic assessment and handling of load balancing as under-utilised parallel processes could have additional cells passed across to them in order to make maximal use of all processors. The data structures of the prototype system minimise the amount of book-keeping and simplify the data access so that parallel implementation is likely to be greatly simplified.

Whilst such exploitation of parallel or distributed architectures is not new [LUKSCH98], it is anticipated that the Object Oriented data structures within the prototype system will greatly facilitate the parallelisation of the software.

9.7 Interactive control expertise

Throughout the current research there has been a lack of high quality expertise about how to manage the interactive control of CFD codes. This is mostly due to the fact that CFD

practitioners have never really had the tools to enable them to perform reliable run-time optimisation of the simulation controls. In order to address this limitation it is predicted that case based data relating to simulation status, reliable control modifications, problematic simulation features and solution reliability need to be collected and analysed in order to facilitate future research.

9.8 Validation and fine-tuning of algorithms

Although the prototype software is being used more frequently from real world simulations [WANG99] there is still considerable scope for validation of all aspects of the algorithms and numerical behaviour in diverse simulation cases.

There is also a need to continually analyse and fine-tune the algorithms within any CFD code in order to best represent each case or application area.

Another potential problem is that it is not known how sensitive CFD codes are to all of their input or pre-configured parameters. It is likely to prove beneficial to perform a comprehensive parametric sensitivity study to determine how critical all of the various input parameters and so called "algorithm constants" are. This is particularly true of the Fire Field Modelling application area where many properties are assumed to be constant over the whole range of temperatures and conditions experienced during a simulation but this is quite an unrealistic assumption for quantities like the specific heat capacity of air.

9.9 Latest Research

The interested reader is advised to check out the University of Greenwich Web pages [GREENWICH] in order to check the current advances in research within the Fire Safety Engineering Group and to see the current status of the Smartfire system.

10 References

[N.B. All references are in chronological order.]

1. [YOKOI60] Yokoi S., "Study on the prevention of fire-spread caused by hot upward current." Report of the Building Research Institute, 1960.

2. [JONES79] Jones I., "A Comparison Problem for Numerical Methods in Fluid Dynamics, The 'Double Glazing' Problem", Numerical Methods in Thermal Problems, Proc. of the First International Conf., pp 338 - 348, 1979.

3. [PATANKAR80] Pantakar S., "Numerical Heat Transfer and Fluid Flow", Intertext Books, McGraw Hill, New York, 1980.

4. [SPALDING81] Spalding D., "A General Purpose computer Program For Multi-Dimensional One- and Two- Phase Flow", Mathematics and Computers in Simulations, North Holland (IMACS), Vol. XXIII, 267, 1981.

5. [STECKLER82] Steckler K., Quintiere J. and Rinkinen W., "Flow Induced By Fire in a Compartment", NBSIR 82-2520, National Bureau of Standards, Washington, 1982.

6. [DAVIS83] De Vahl Davis G. and Jones I., "Natural Convection in a Square Cavity: A Comparison Exercise", International Journal for Numerical Methods in Fluids, Vol. 3, pp 227 - 248, 1983.

7. [KNIGHT87] Knight B., Cross M., and Edwards D., "Software Design for Numerical Software", Reliability and Robustness of Engineering Software, Ed. Brebbia C., and Keramidas G., pp 121 - 136, 1987.

8. [KERNIGAN88-1] Kernigan, B. & Wilson, B., "lex - a lexical analysis tool", The C

programming language, Pub. Prentice-Hall, 1988.

9. [KERNIGAN88-2] Kernigan, B. & Wilson, B., "yacc - yet another compiler compiler", The C programming language, Pub. Prentice-Hall, 1988.

10. [WILLIAMS88] Williams A., "The Development of an Intelligent Interface to a Computational Fluid Dynamics Flow-Solver Code", Computers and Structures, No. 1/2, pp 431 - 438, 1988.

11. [GALEA89] Galea E., "On the field modelling approach to the simulation of enclosure fires", Journal of Fire Protection Engineering, vol. 1 (1), 1989, pp 11 - 22.

12. [EASYFLOW90] EasyFlow: CHAM (Concentration, Heat and Momentum) North America: EasyFlow Reference Manual, 1990.

13. [THIMBLEBY90] Thimbleby H., "User Interface Design", Publisher: Addison Wesley 1990, ISBN: 0201416182.

14. [ANGUS91] Angus I., and Stolzy J., "Experiences converting an application from FORTRAN to C++ : Beyond f2c", C++ at work conference, November 1991.

15. [BYRNE91] Byrne E., "Software Reverse Engineering: A Case Study", Software Practice and Experience, Vol. 21 (12), pp 1349 - 1364, December 1991.

16. [FIELD91] Field D., "A generic Delaunay triangulation algorithm for finite element meshes, Advances in Engineering Software, Vol. 13, No. 5/6, pp 263 - 272, September 1991.

17. [FLOW3D91] FLOW3D Release 2.3.3 Reference Guide, CFD Dept., AEA Harwell, UK, February 1991.

18. [JAMBUNATHAN91] Jambunathan K., Lai E., Hartle S., and Button B., "Development of an Intelligent Front-End for a Computational Fluid Dynamics Package", Artificial Intelligence in Engineering, 1991, Vol. 6, No. 1, pp 27 - 35.

19. [KNIGHT91] Knight B., and Petridis M., " A Design For Reliable CFD Software", Reliability and Robustness of Engineering Software II, Ed. Brebbia C., Ferrante A., pp 3 - 17, Elsevier, 1991.

20. [KUMAR91] Kumar S., Gupta A. and Cox G., "Effects of Thermal Radiation on the Fluid Dynamics of Compartment Fires", Fire Safety Science - Proc. of the Third International Symp., pp 345 - 354, 1991.

21. [DUBOIS-PELERIN92] Dubois-Pelerin Y., Zimmermann T. and Bomme P., "Object-oriented finite element programming: II. A prototype program in Smalltalk", Computer Methods in Applied Mechanics and Engineering 98, pp 361 - 397, 1992.

22. [IEROTHEOU92] Ierotheou C., and Galea E., "A Fire Field Model implemented in a Parallel Computing Environment", International Journal for Numerical Methods in Fluids, Vol. 14, Issue 2, pp 175 - 187, Jan 1992.

23. [PETRIDIS92] Petridis M., and Knight B., "FLOWES: An Intelligent CFD System", Engineering Applications of Artificial Intelligence, Vol. 5(1), pp 51 - 58, 1992.

24. [SCATENI92] Scateni R., "Towards integrated object-oriented computational fluid dynamics environments: Interactive Domain Editor", Conf. Proc. 3rd Eurographics Workshop on Visualisation in Scientific Computing, pp 83 - 98, 27 - 29 April 1992.

25. [SPALDING92] Spalding D., "The expert-system CFD code; problems and partial solutions", International High-Tech Forum, Basel, 27th May 1992.

26. [COBALT93] for-C translation software, Cobalt Blue Inc., 11585 Jones Bridge Rd., Ste 420 - 306 Alpharetta, GA 30005, USA.

27. [DUBOIS-PELERIN93] Dubois-Pelerin Y. and Zimmermann T., "Object-oriented finite element programming: III. An efficient implementation in C++", Computer Methods in Applied Mechanics and Engineering 108, pp 165 - 183, 1993.

28. [EDWARDS93] Edwards H., and Dr. Hayes L., "Visual Programming of Iterative Methods", Conf. Proc. 1st Annual OON-SKI '93 (Object Oriented Numerics) Conference, pp 163 - 170, 1993.

29. [EWER93-1] Ewer J., Petridis M., Cowell D., and Knight B., "An Intelligent User Interface for Computational Fluid Dynamics Software", Proceedings of AIENG '93, pp 77 - 92, 1993.

30. [EWER93-2] Ewer J., Numdiff - A numerical file differencing utility, 1993, The University of Greenwich, School of Computing and Mathematical Science, Maritime Greenwich Campus, Greenwich, London.

31. [F2C93] f2c is a freeware Fortran to C translation utility. Available via the GNU Free Software Foundation, 1993, Original code is © AT&T, Lucent Technologies and Bellcore.

32. [PETRIDIS93] Petridis M., and Knight B., "A Blackboard Approach for the Integration of an Intelligent Knowledge Based System into Engineering Software", Knowledge Based Systems for Civil and Structural Engineering, Ed. Topping B., pp 49 - 56, Civil-Comp Press, 1993.

33. [SPAG93] SPAG - A Fortran tool for restructuring spaghetti code. Part of plusFORT, 1993, Salford Software Ltd., Adelphi House, Adelphi Street, Salford, M3 6EN.

34. [TWORZYDLO93] Tworzydlo W. and Oden J., "Towards an automated environment in computational mechanics", Computer Methods in Applied Mechanics and Engineering 104, pp 87 - 143, 1993.

35. [AFZAL94] Afzal M., Cross M., "GASFLO - airflow distribution evaluation tool for ducting systems of pellet induration processes", Applied Mathematical Modelling, Vol. 18, pp 408 - 414, 1994.

36. [ANGUS94] Angus I., and Curtis W., "From Fortran to Object Orientation: Experiences with a Production Flutter Analysis Code", Conf. Proc. 2nd Annual OON-SKI '94 (Object Oriented Numerics) Conference, pp 174-180, 1994.

37. [BANERJEE94] Banerjee D., Morley C., and Smith W., "The design and implementation of the Cortex visualisation system", Conf. Proc. Visualization '94 (Cat. No. 94CH35707), pp 265 - 272, 17 - 21 October 1994.

38. [CROSS94] Cross M., Chow P., Ewer J., et al., "PHYSICA - A Software Environment for the Modelling of Multi-physics Phenomena", 1994, Internal publication at the University of Greenwich, CNMPA, London, SE18 6PF.

39. [KERRISON94] Kerrison L., Mawhinney N., Galea E., Hoffmann N. and Patel M., "A Comparison of Two Fire Field Models With Experimental Room Fire Data", Fire Safety Science - Proc. of the Fourth Intl. Symp., Ottawa, Canada, 13-17 July 1994, pp 161-172.

40. [PARSONS94] Parsons R., and Quinlan D., "A++ / P++ Array Classes for Architecture Independent Finite Difference Computations", Conf. Proc. 2nd Annual OON-SKI '94 (Object Oriented Numerics) Conference, pp 408 - 418, 1994.

41. [UPHAM94] Upham D., "FUNGI: Finite-difference Using a Nearly Graphic Interface", Conf. Proc. 2nd Annual OON-SKI '94 (Object Oriented Numerics) Conference, Poster Session, pp 464 - 467, 1994.

42. [BAILEY95] Bailey C., Ewer J., et al., "An Object Oriented Approach to Computational Mechanics - Physica", 1995, SEL-HPC Short Course delivered by The University of Greenwich, CNMPA, London, SE18 6PF.

43. [BJORKMAN95] Bjorkman J., Keski-Rahkonen O., and Lewis M., "First Simulations of the Steckler Room Fire Experiment by using SOPHIE", Conf. Proc., First European Symp. On Fire Safety Science, Zurich, Switzerland, 21 - 23 August 1995.

44. [COX95] Combustion Fundamentals of Fire, Editor: Cox G., Academic Press, 1995.

45. [EWER95] Ewer J., Knight B. and Cowell D., "Case Study: An Incremental Approach to Re-engineering a Legacy FORTRAN Computational Fluid Dynamics Code in C++", Advances in Engineering Software, Vol. 22, pp 153 - 168, 1995.

46. [PETRIDIS95] Ph.D. Thesis: "Integrating an Intelligent Knowledge Based System into CFD Software", Petridis M., 1995, The University of Greenwich, School of Computing and Mathematical Science, UK.

47. [BRAND96] Van Den Brand M., Klint P., and Verhoef C., "Reverse engineering and system renovation - an annotated bibliography", Technical Report P9603, University of Amsterdam, Programming Research Group, 1996, ACM Software Engineering Notes.

48. [GALEA96] Galea E., Berhane D. and Hoffmann N., "CFD Analysis of Fire Plumes Emerging from Windows with External Protrusions in high-rise Buildings", Proc. Interflam 96, Cambridge, UK, pp 835 - 839, March 1996.

49. [PETRIDIS96] Petridis M., and Knight B., "The Integration of an Intelligent Knowledge Based System into Engineering Software using the Blackboard Structure", Advances in Engineering Software, Vol. 25, pp 141 - 147, 1996.

50. [TAYLOR96] Taylor S., Galea E., Patel M., Petridis M., Knight B. and Ewer J., "SMARTFIRE: An Intelligent Fire Field Model", Proc. Interflam 96, Cambridge, UK, March 1996, pp 671 - 680.

51. [GLOCKING97] Dr Glocking J., Annable K., and Campbell S., "Fire spread in multi-storey buildings : Fire break out from heavyweight unglazed curtain wall system - Run 007", Document TE88932-43, Confidential Report, 1997, The Loss Prevention Council, Melrose Ave., Borehamwood, Herts., UK, WD6 2BJ.

52. [HUME97] Hume B., "Development of a User-Friendly Interface for a Fire Model", UK Home Office FRDG, Central Fire Brigades Advisory Council Joint Committee on Fire Research, Research Report No. 77, 1997.

53. [JIA97] Jia F., Galea E., and Patel M., "The prediction of Fire Propagation in Enclosed Fires", Fire Safety Science - Proc. 5th International Symposium, 1997, pp 439 - 450.

54. [LEWIS97] Lewis M., Moss M. and Rubini P., "CFD Modelling of Combustion and Heat Transfer in Compartment Fires", Fire Safety Science, Proc. of the Fifth Int. Symp., Ed: Hasemi Y., pp 463 - 474, 1997.

55. [TAYLOR97-1] Taylor S., Petridis M., Knight B., Ewer J., Galea E. and Patel M., "SMARTFIRE: An Integrated CFD code and expert system for fire field modelling", Fire Safety Science, Proceedings of the 5th Int. Symp., Ed: Hasemi Y., 1997, pp 1285 - 1296.

56. [TAYLOR97-2] Ph.D. Thesis: "An Investigation into Automation of Fire Field Modelling Techniques", Taylor S., September 1997, The University of Greenwich, School of Computing and Mathematical Science, UK.

57. [THIMBLEBY97] "People and Computers XII", Editors: Thimbleby H., O'Conaill B. and Thomas P., August 1997, Publisher: Springer-Verlag Berlin and Heidelberg GmbH & Co. KG, ISBN: 3540761721.

58. [CROFT98] Ph.D. Thesis: "Unstructured Mesh - Finite Volume Algorithms for Swirling, Turbulent, Reacting Flows", Croft T., June 1998, The University of Greenwich, School of Computing and Mathematical Science, UK.

59. [EWER98] Ewer J., Galea E., Knight B., Patel M., Janes D., Petridis M., "Fire Field Modelling using the SMARTFIRE Automated Dynamic Solution Control Environment", CMS Press, Paper Number 98/IM/41, ISBN 1899991387, London, 1998.

60. [LUKSCH98] Luksch P., Maier U., Rathmayer S., et al., "Software Engineering in parallel and distributed scientific computing: a case study from industrial practice", Proc. International Symp. on Software Eng. for Parallel and Distributed Systems (Cat. No. 98EX155), pp 187 - 197, 20 - 21 April 1998.

61. [EWER99-1] Ewer J., Galea E., Patel M., Taylor S., Knight B. and Petridis M., "SMARTFIRE: An Intelligent CFD Based Fire Model", Journal of Fire Protection Engineering, Vol. 10, No. 1, pp 13 - 27, 1999.

62. [EWER99-2] Galea E., Knight B., Patel M., Ewer J., Petridis M., and Taylor S., "SMARTFIRE V2.01 build 365, User Guide and Technical Manual", Smartfire CD and bound manual, 1999.

63. [EWER99-3] Ewer J., Galea E., Patel M. and Knight B., "The Development and Application of Group Solvers in the SMARTFIRE Fire Field Model", Proc. Interflam 99, Edinburgh, UK, June/July 1999, Vol. 2, pp 939 - 950.

63. [EWER99-4] Ewer J., Galea E., Patel M. and Knight B., "Enhancing the Numerical Performance of Fire Field Models", CMS Press, Paper No. 99/IM/52, ISBN No. 1 899991 53 0, 1999.

64. [JIA99] Ph.D. Thesis: "The Simulation of Fire Growth and Spread within Enclosures using an Integrated CFD Fire Field Model", Jia F., October 1999, The University of Greenwich, School of Computing and Mathematical Science, UK.

65. [WANG99] Wang Z., Jia F., Galea E., Patel M. and Ewer J., "Simulating One of the CIB W14 Round Robin Test Cases using the SMARTFIRE Fire Field Model", December 1999, To be submitted to Fire Safety Journal.

66. [BERGEY] Bergey J., Smith D., Tilley S., Weiderman N., and Woods S., "Why Reengineering Projects Fail", Technical Report CMU/SEI-99-TR-010, ESC-TR-99-010, Carnegie Mellon Software Engineering Institute, Pittsburg, PA 15213 - 3890.

67. [CHAM] PHOENICS and PHOENICS VR: CHAM (Concentration, Heat and Momentum) Ltd., Bakery House, Wimbledon Village, London, SW19 5AU, UK.

68. [GREENWICH] www: http://fseg.gre.ac.uk/

69. [MICROSOFT] Microsoft Windows 95, 98 and NT, Copyright © Microsoft Corporation, One Microsoft Way, Redmond, Washington 98052-6399 U.S.A.

70. [NETCFD] "netCFD: A Ninf CFD component for Global Computing, and Java applet GUI", Parallel & Distributed System Performance TRC Laboratory Real World Computing Partnership, JAPAN.

71. [WEBSTER] Webster B., "Pitfalls of Object-Oriented Development", M&T Books, New York, ISBN 1-558513-97-3.

72. [ZINC] Zinc Application Framework, Zinc Software Incorporated, 405 South 100 East, Pleasant Grove, Utah, 84064, U.S.A.

11 Appendices

This appendix section gives more detailed or background information which was not appropriate for direct inclusion in the main thesis as it would tend to clutter the text.

11.1 SMARTFIRE VERIFICATION AND VALIDATION REPORT by Ewer J., Jia F. and Grandison A.

SMARTFIRE VERIFICATION AND VALIDATION REPORT

By J. Ewer, A. Grandison and F. Jia.

1 Introduction

The test cases presented in this document serve to verify and validate the SMARTFIRE CFD fire modelling software. The report is split into four sections. Section 2 contains test cases concerned with verifying that the basic physics within SMARTFIRE has been correctly implemented. Section 3 is used to compare SMARTFIRE predictions against data derived from fire experiments and data generated by other fire models. Finally, section **Error! Reference source not found.** provides detailed information concerning the problem set-up (e.g. meshes used in the test cases) and additional detailed information to allow other users to reproduce the SMARTFIRE results.

Validation is not a "once and forget" task. It is an on-going activity that both code developers and users should be involved with. It is expected that this report will grow in time as more test cases are developed and the capabilities of SMARTFIRE expand. SMARTFIRE users are encouraged to develop other test cases and to report their findings to the code developers. When reporting verification/validation results please ensure that complete details of the SMARTFIRE set-up are provided along with your SMARTFIRE predictions and expected results for comparison purposes. Please report validation cases by email to smartfire@fseg.gre.ac.uk

Unless otherwise stated the following material properties are used in the test cases: -

PROPERTY	AIR	COMMON BRICK
CONDUCTIVITY CONSTANT	0.02622	0.69
VISCOSITY CONSTANT	1.6e-005	1e+010
NATURAL STATE	Gas	Solid
THERMAL EXPANSION	0.003292	0
DENSITY	Use Ideal gas law	1600
Molecular weight	29.35	N/A
Specific heat constant	1045.78	840

Unless otherwise stated, the CFD codes used in this document are: -

SMARTFIRE v2.01 build 369, produced by FSEG of the University of Greenwich,
PHOENICS v2.1.3, produced by CHAM Ltd of the U.K., and
CFDS-FLOW3D v2.3.2, produced by AEA Technology of the U.K.

2 General Physics Validation Cases

2.1 Basic physics verification of SMARTFIRE — Convective term.

This test examines whether the convective term in SMARTFIRE is functioning correctly. The tests involve a simple 2D fluid flow in a box. The fluid uniformly enters the box from an inlet and leaves via an outlet located opposite to the inlet (see Figure 1). The fluid temperature is uniform. The test was repeated in the three co-ordinate directions (x, y, z) in the positive and negative directions. This leads to six test cases which should all produce identical results. All these tests are further repeated with heat transfer and also with heat transfer and buoyancy. Due to the use of the symmetry planes, the flow leaving the geometry should exit uniformly with the same velocity with which it entered and possess the same velocity throughout the domain.

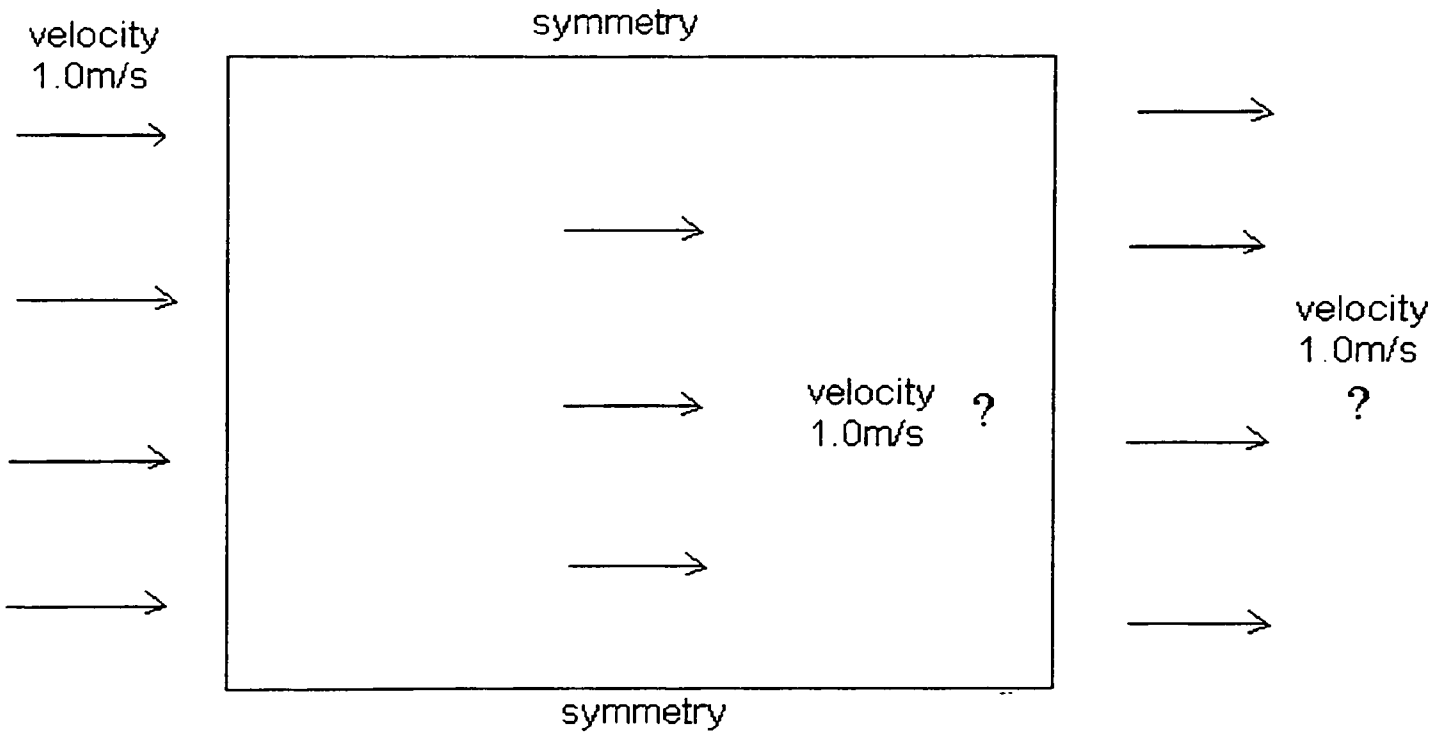


Figure 1 - Geometry for 2D flow case

2.1.1 SMARTFIRE results

a) flow only, no heat transfer and buoyancy

maximum velocity : 1.0 m/s, minimum velocity: 1.0 m/s. ✓

b) flow and heat transfer but no buoyancy

maximum velocity : 1.0 m/s, minimum velocity: 1.0 m/s. ✓

c) flow, heat transfer and buoyancy

maximum velocity : 1.0 m/s, minimum velocity: 1.0 m/s. ✓

2.2 *Radiation verification in 2D cavity without flow.*

These tests were designed and carried out to examine the implementation of the SMARTFIRE six-flux radiation model. The cases concern radiation within a cavity with hot and cold walls and a uniform temperature distribution within the media. Scattering is neglected. The geometry used in the test cases is presented below in Figure 2.

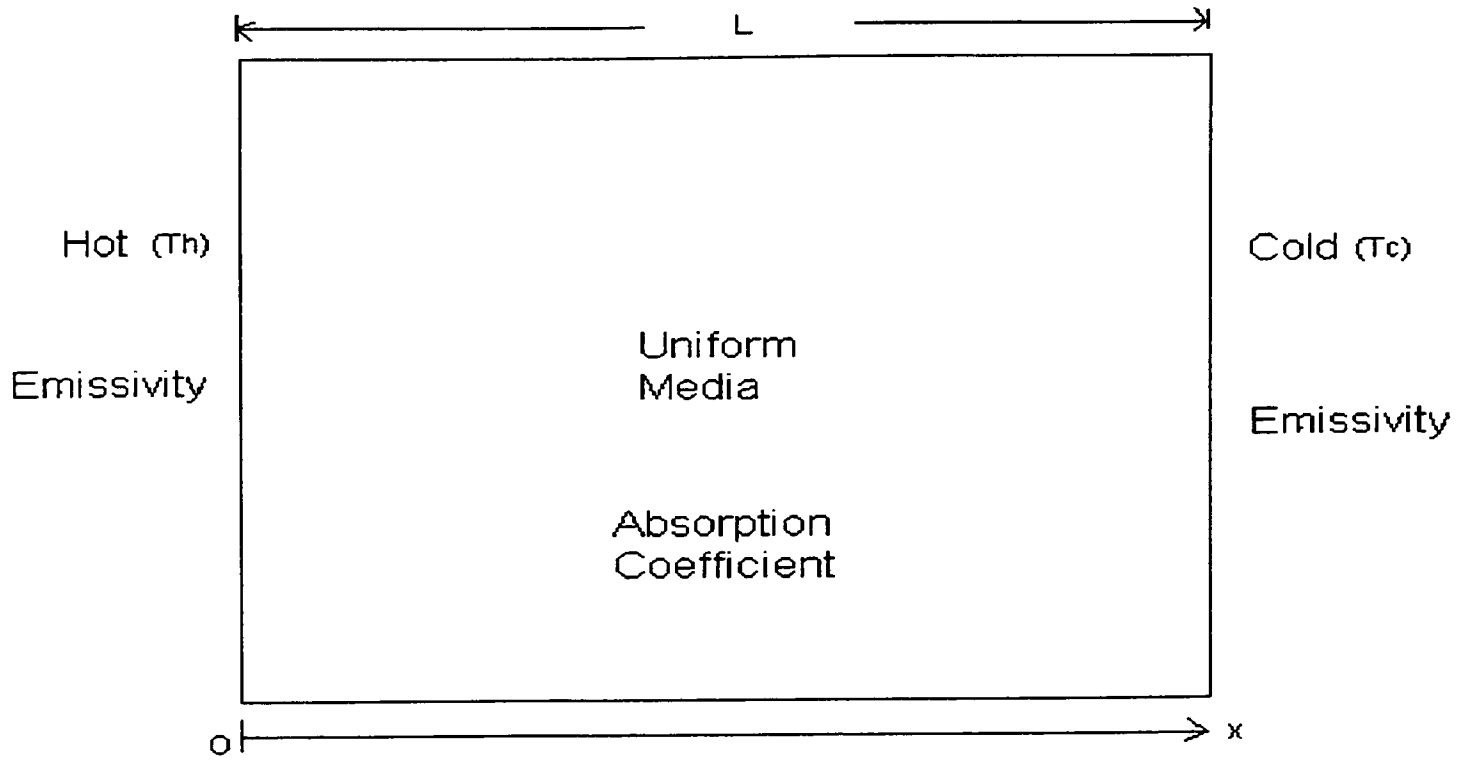


Figure 2 - 2D verification of SMARTFIRE six-flux radiation model.

For this scenario it is possible to determine an exact analytical radiation flux based on the six-flux model formulation. Let T_h, T_c, T_m, ε_h, ε_c and α denote the hot wall temperature, the cold wall temperature, the media temperature, the hot wall emissivity, the cold wall emissivity and the absorption coefficient of the media respectively. Then the exact analytical solution for the radiation fluxes as determined by the six-flux model is:

The radiation flux along the negative x direction, F^- is :

$$F^- = D\exp(-\alpha(L-x)) + \sigma(1-\exp(-\alpha(L-x)))\, T_m^4$$

Where,

$$D = (1-\varepsilon_c)C + \varepsilon_c \sigma\, T_c^4$$

$$C = B/A$$

$$B = \sigma\exp(-\alpha L)\,[\varepsilon_h\, T_h^4 + (1-\varepsilon_h)\,(1-\exp(-\alpha L))T_m^4 + (1-\varepsilon_h)\,\varepsilon_c\exp(-\alpha L)\, T_c^4] + \sigma(1-\exp(-\alpha L))T_m^4$$

$$A = 1-(1-\varepsilon_h)(1-\varepsilon_c)\exp(-2\alpha L)$$

The radiation flux along the positive x direction, F^+ is :

$F^{+} = D\exp(-\alpha x) + \sigma(1-\exp(-\alpha x))\ T_m^{\ 4}$

Where,

$D = (1-\varepsilon_h)C + \varepsilon_h \sigma\ T_h^{\ 4}$

$C = B/A$

$B = \sigma\exp(-\alpha L)\ [\varepsilon_c\ T_c^{\ 4} + (1-\varepsilon_c)\ (1-\exp(-\alpha L))T_m^{\ 4} + (1-\varepsilon_c)\ \varepsilon_h\ \exp(-\alpha L)\ T_h^{\ 4}]$
$+ \sigma(1-\exp(-\alpha L))T_m^{\ 4}$

$A = 1-(1-\varepsilon_h)(1-\varepsilon_c)\exp(-2\alpha L)$

where σ is the Stefan-Boltzmann constant.

The test was repeated for various values of emissivity, absorption coefficient.

2.2.1 SMARTFIRE results

The model predictions are compared with the exact solutions. In these comparisons, the temperatures of the hot wall, the cold wall and the media are 774 K, 304 K and 574 K respectively. The length of the cavity L is 1m. The cell size is 0.0222m.

a) Absorption coefficient of the media is zero, emissivity is 1.0.

If $\varepsilon_{all} = 1.0$ and $\alpha = 0.0$ then

$F^{-} = \sigma T_c^{\ 4}$

$F^{+} = \sigma T_h^{\ 4}$

The results are tabulated below (Table 1)

Table 1 - Theoretical and SMARTFIRE results for radiation fluxes when $\varepsilon_{all} = 1.0$ and $\alpha = 0.0$

Flux	Theoretical	SMARTFIRE
F^{-}	4.8423E+02	4.826421E+02
F^{+}	2.0348E+04	2.032182E+04

Maximum relative error < 1%.

b) Absorption coefficient of the media is zero, emissivity is 0.7.

If $\varepsilon = 0.7$ and $\alpha = 0.0$ then

$$F^{-} = \varepsilon\sigma T_c^{\ 4} + \frac{(1-\varepsilon)\sigma(\varepsilon T_h^{\ 4} + (1-\varepsilon)\varepsilon T_c^{\ 4})}{1-(1-\varepsilon)(1-\varepsilon)}$$

$$F^+ = \varepsilon\sigma T_h^4 + \frac{(1-\varepsilon)\sigma(\varepsilon T_c^4 + (1-\varepsilon)\varepsilon T_h^4)}{1-(1-\varepsilon)(1-\varepsilon)}$$

The results are tabulated below (Table 2).

Table 2 - Theoretical and SMARTFIRE results for radiation fluxes when $\varepsilon_{all} = 0.7$ and $\alpha = 0.0$

Flux	Theoretical	SMARTFIRE
F^+	1.59518E+04	1.574355E+04
F^-	5.060E+03	5.060915E+03

Maximum relative error < 1%.

c) Absorption coefficient of the media is one, emissivity is 1.0.

As there is now absorption from the media, this leads to the fluxes being dependent on displacement. The results are illustrated below in Figure 3 and Figure 4.

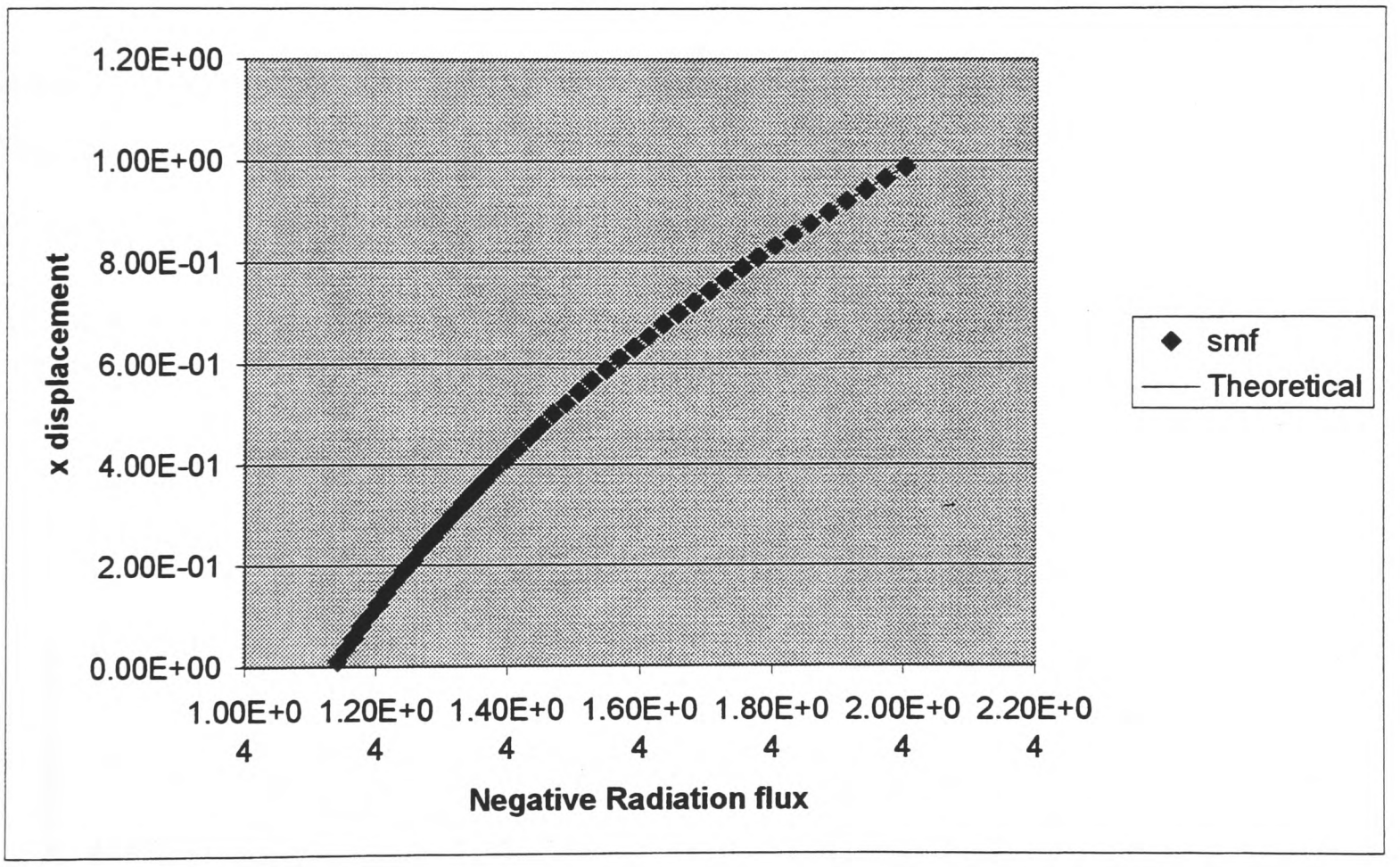


Figure 3 - Theoretical and SMARTFIRE results for negative radiation fluxes when $\varepsilon_{all} = 1.0$ and $\alpha = 1.0$

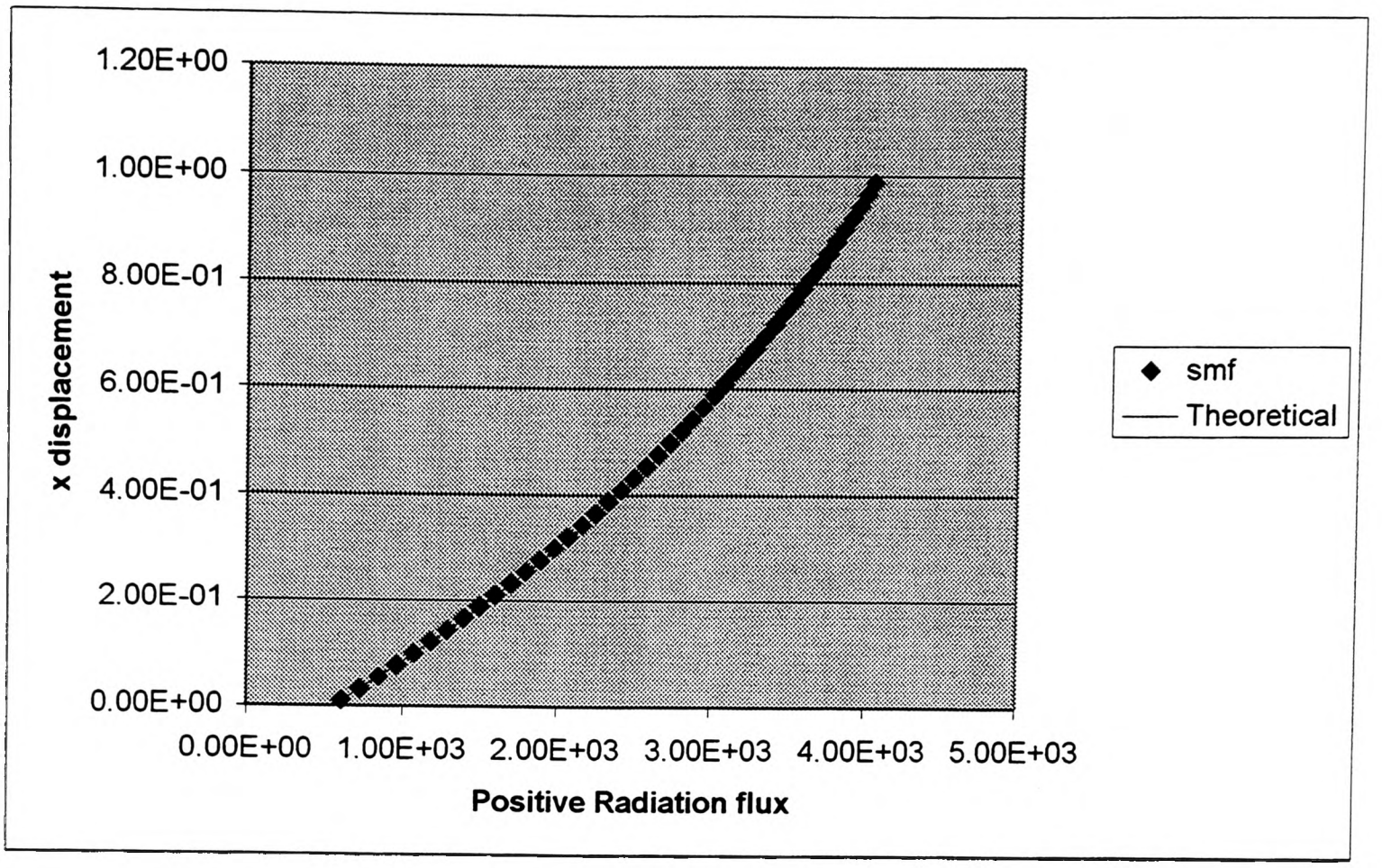


Figure 4 - Theoretical and SMARTFIRE results for positive radiation fluxes when $\varepsilon_{all} = 1.0$ and $\alpha = 1.0$

Maximum relative error < 1%.

d) Absorption coefficient of the media is one, emissivity is 0.7.

As there is now absorption from the media, this leads to the fluxes being dependent on displacement. The results are illustrated below in Figure 5 and Figure 6.

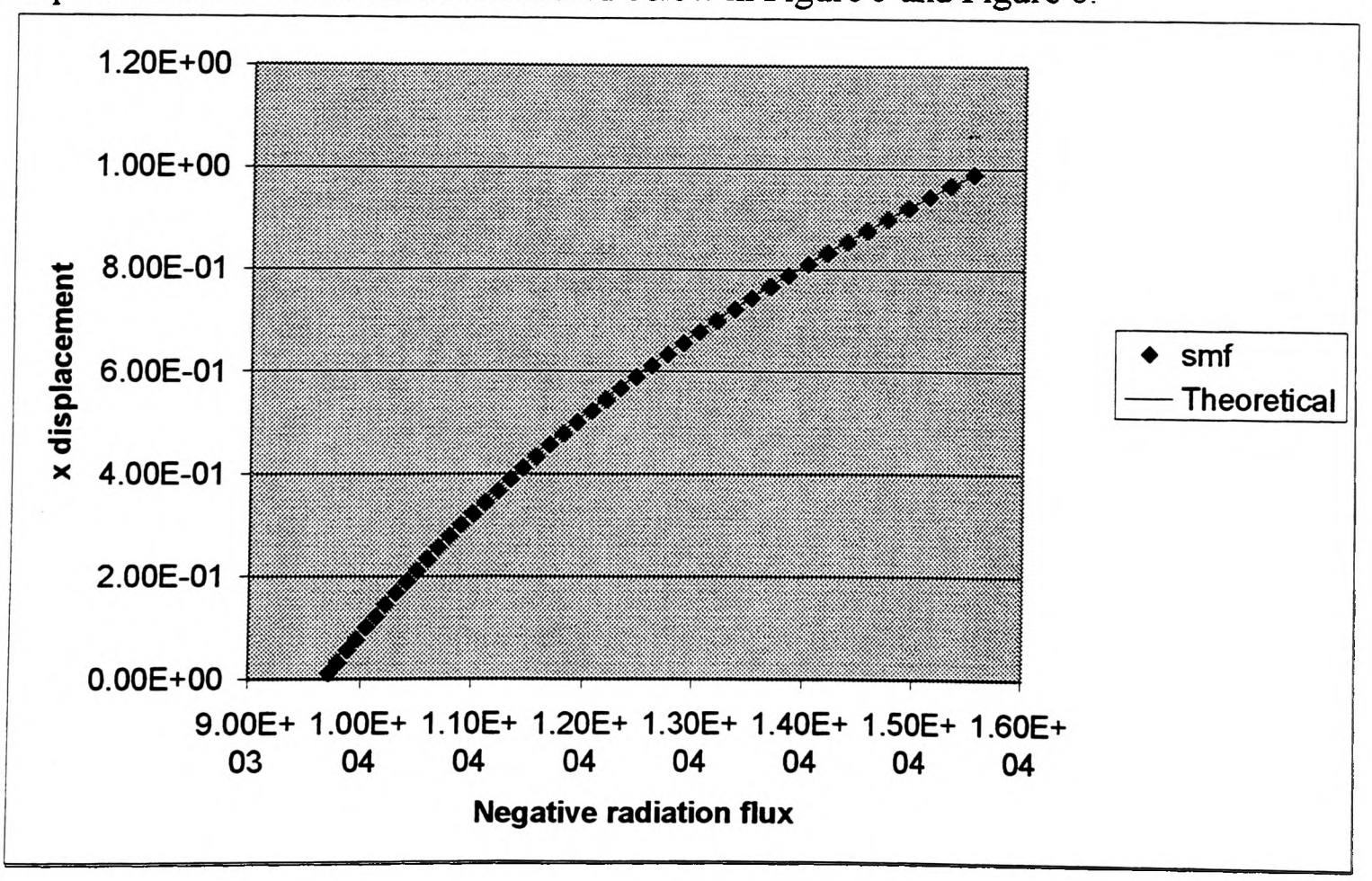


Figure 5 - Theoretical and SMARTFIRE results for negative radiation flux when $\varepsilon_{all} = 0.7$ and $\alpha = 1.0$

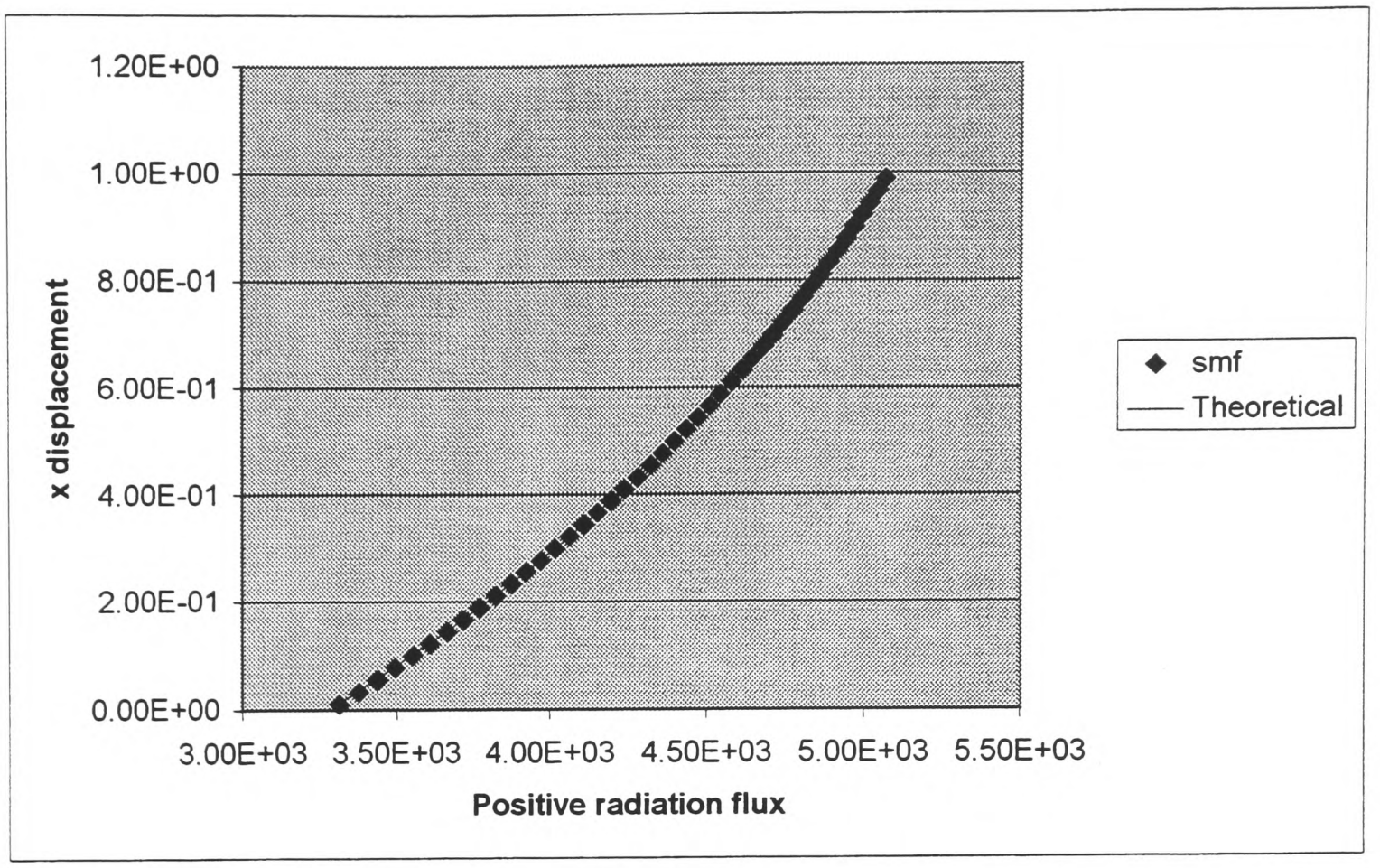


Figure 6 - Theoretical and SMARTFIRE results for the positive radiation flux when $\varepsilon_{all} = 0.7$ and $\alpha = 1.0$

Maximum relative error < 1%.

The results from this test suggest that the six-flux model has been correctly implemented

2.3 Symmetry boundary condition test

This case is intended to test if the symmetry function works correctly. The case involves flow expansion from a small duct into a larger duct. The configuration is shown in Figure 7 below. The case was simulated using the whole flow domain and then repeated using a symmetry boundary condition along the central axis.

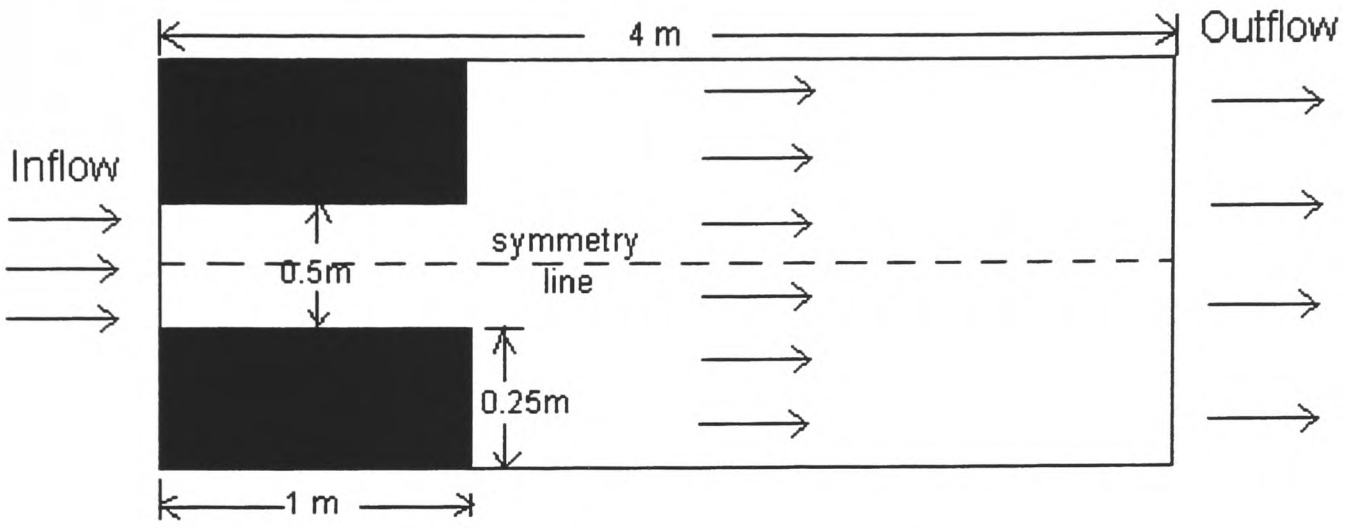


Figure 7 - Expanding duct with symmetry line indicated

2.3.1 SMARTFIRE results

The flow fields of the symmetry case (Figure 8) and whole field case (Figure 9) systems are plotted below. The velocity profile at the outlet is plotted in Figure 10.

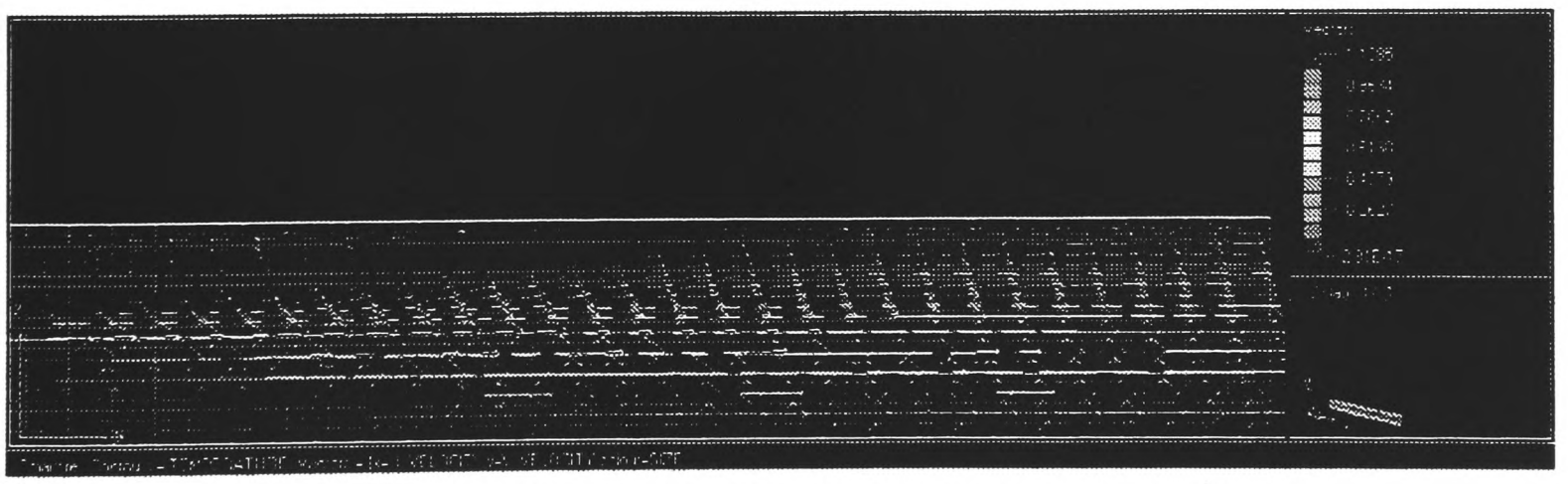


Figure 8 -The flow field producted by the simulation using the half system

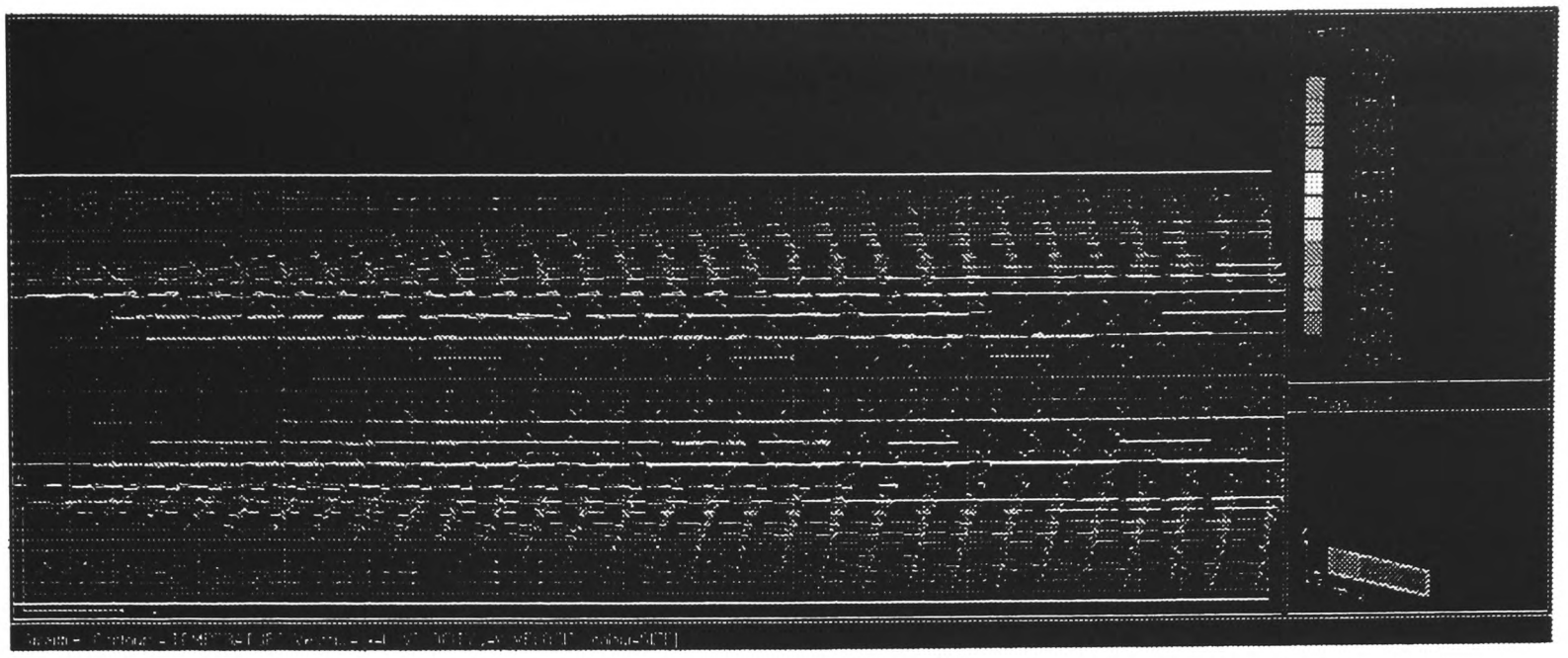


Figure 9 - The flow field produced by the simulation using the whole system

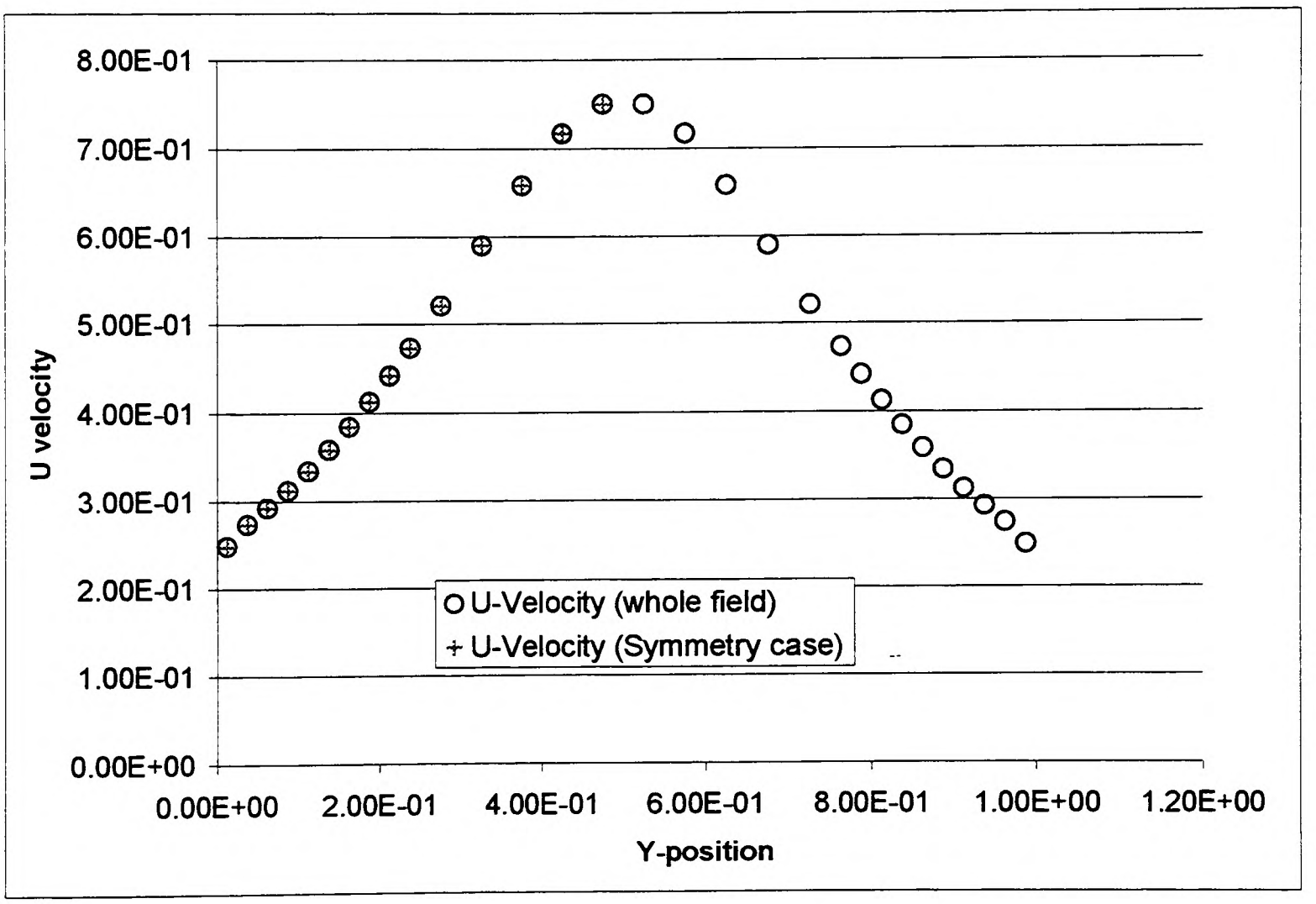


Figure 10 - Comparison of the whole field and symmetry cases at the outlet

The results suggest that the symmetry condition within SMARTFIRE functions as intended for isothermal flows.

2.4 *Two-dimensional turbulent flow over a backward facing step.*

This test examines the SMARTFIRE κ-ε turbulence model. The comparison is carried out between SMARTFIRE, PHOENICS and FLOW3D. The same mesh (60×50) is used for all the CFD codes. The flow properties and boundary conditions are described below with the configuration shown in Figure 11:

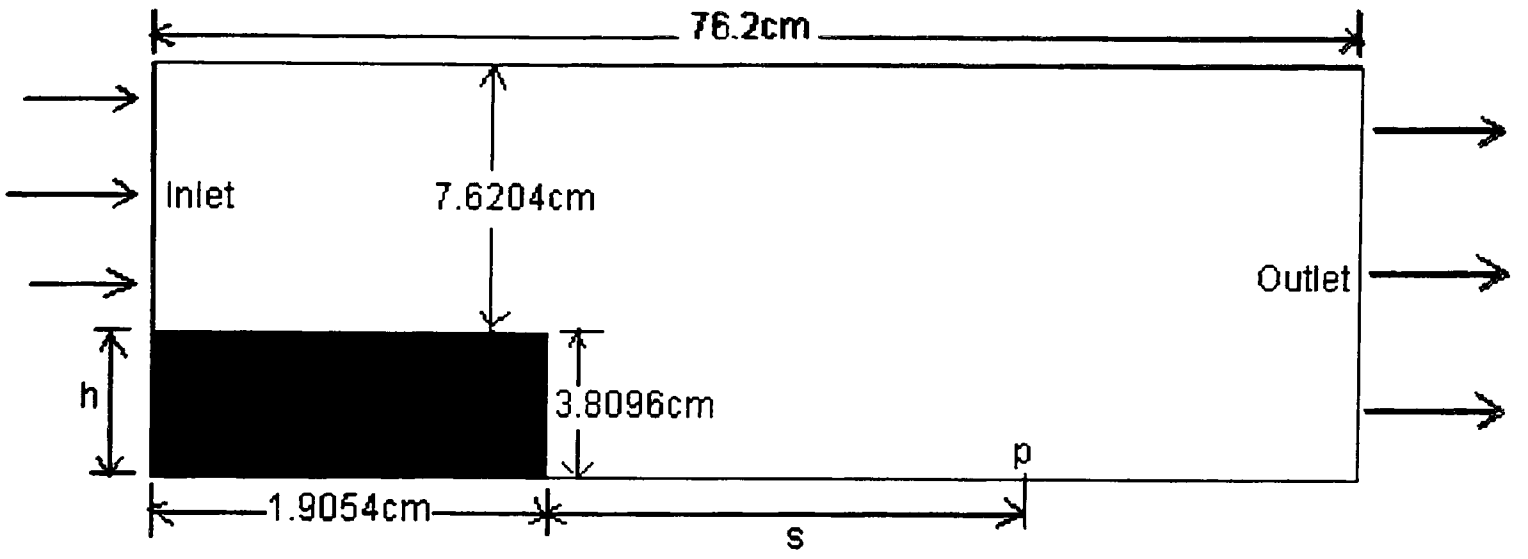


Figure 11 - Backward facing step configuration

Fluid properties

Density: 1 kg/m^3,
Viscosity: 1.101E-5kg/ms.

Boundary conditions

At the inlet

velocity:13.0 m/s,
kinetic energy:0.7605 m^2/s^2,
dissipation rate:31.78 m^2/s^3.

There is no heat transfer in this problem. It should be noted that SMARTFIRE and FLOW3D use the same wall function formulation while PHOENICS uses a different wall function formulation.

2.4.1 Results

Reattachment point

The reattachment point is the downstream location in the x direction where there is no longer any flow re-circulation due to the backward facing step.

In Figure 11 the reattachment point is denoted by P and the distance from the step to point P is s. The ratio of s to the height of the step (h) predicted by the following CFD codes are SMARTFIRE: 5.70; PHOENICS: 6.55; FLOW3D: 5.16. Experimental results indicates that the ratio is 7.1 [1, 2]

All three codes predict values for the reattachment point that are similar and all codes under-predict the correct value. The SMARTFIRE prediction falls between that of PHOENICS and FLOW3D. It is expected that these values will improve with further mesh refinement.

Velocity profiles

In addition to the reattachment distance, it is also important to compare the prediction of the velocity profile at several locations within the duct. In this case, the SMARTFIRE, PHOENICS and FLOW3D generated U velocity profile at the outlet and 0.285 m from the inlet are compared. U velocity profiles for this case at the two different positions are presented below in Figure 12 and Figure 13.

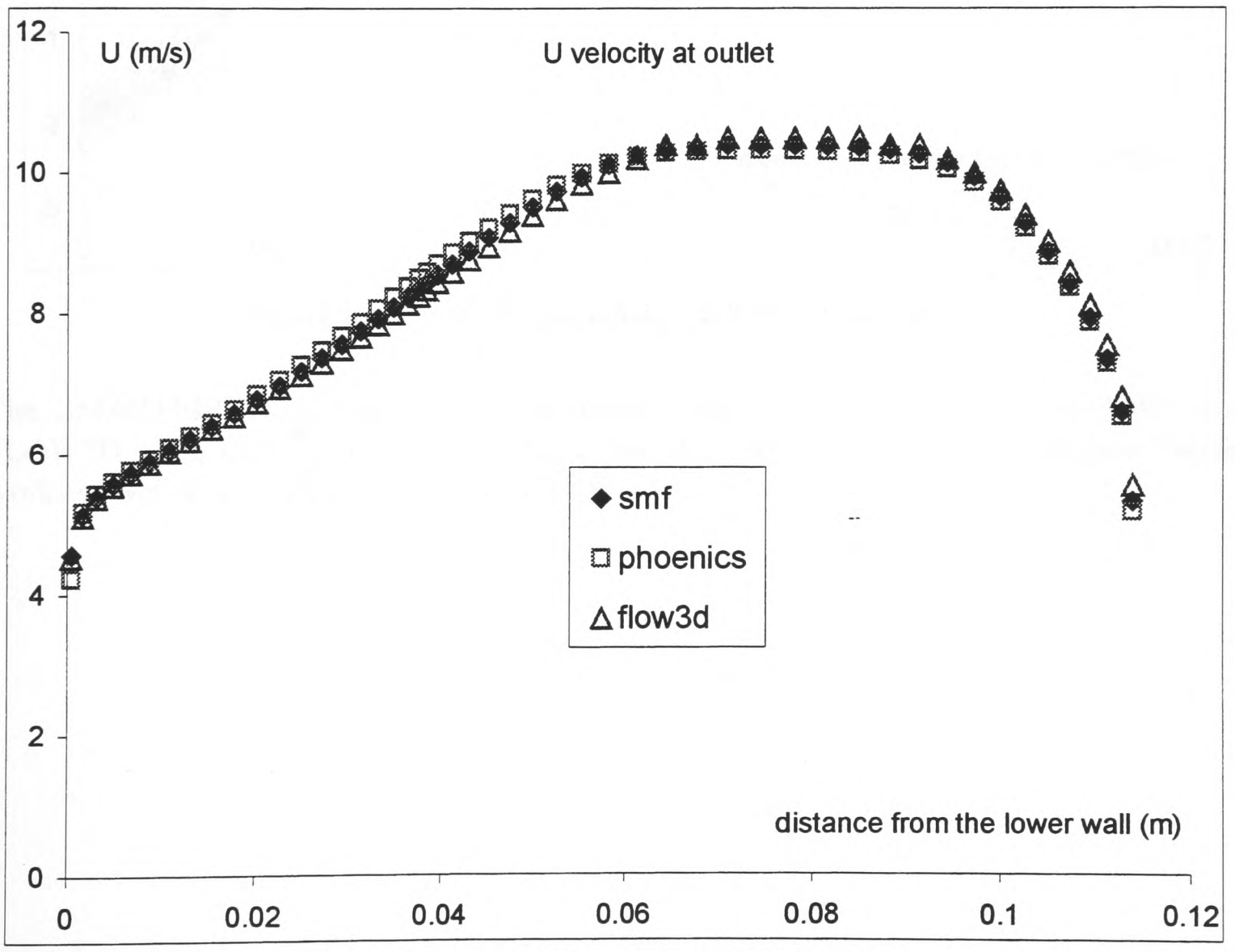


Figure 12 - U velocity against height at the outlet.

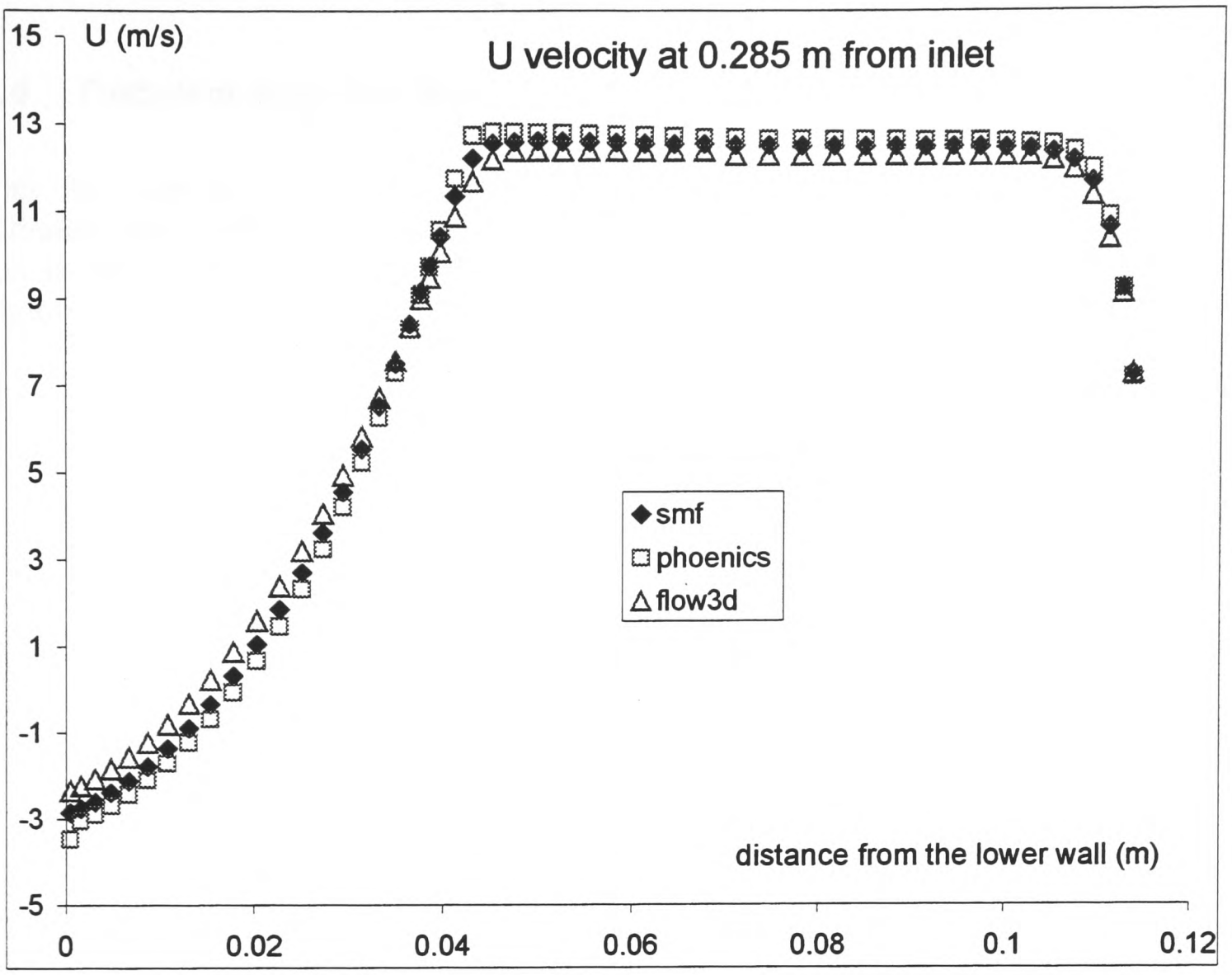


Figure 13 - U velocity against height at 0.285m from the inlet

The SMARTFIRE velocity profiles are very close to the profiles of PHOENICS and FLOW3D. The comparison demonstrates that the SMARTFIRE κ-ε turbulence model works as well as either PHOENICS or FLOW3D.

2.5 Turbulent long duct flow.

This test case examines the SMARTFIRE κ-ε turbulence model in conjunction with turbulent heat transfer. This case has been well investigated with PHOENICS in the past and is part of the PHOENICS test case library. The geometry of the case is depicted in Figure 14.

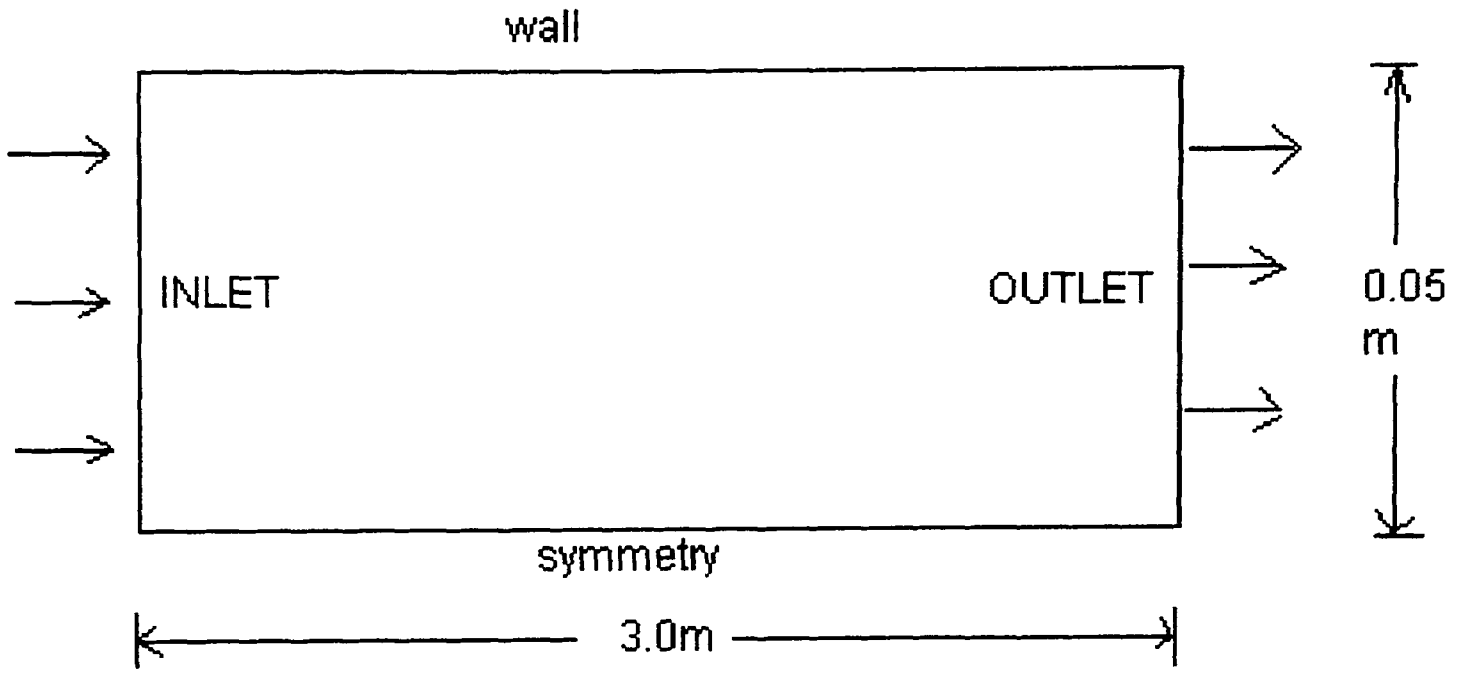


Figure 14 - Turbulent long duct flow configuration

Fluid properties

Conductivity: 0.07179 (W/mK)
Density: 1 (kg/m^3)
Viscosity: 5e-5 (kg/ms)
Specific heat: 1005 (J/kgK)

Inlet Conditions

Velocity : 50 m/s
Turbulent kinetic energy: 11.25 (m^2/s^2)
Dissipation rate: 1378.0 (m^2/s^3)
Enthalpy: 10 (J/kg)

Wall Condition

Fixed enthalpy value :(1 J/kg).

No buoyancy is used in this problem.

The 2 dimensional mesh is non-uniformly distributed and the cell budget is 600 (20×30).

2.5.1 Results

The results from SMARTFIRE are compared with those from PHOENICS. The comparison includes the enthalpy and velocity profiles across the duct at the outlet.

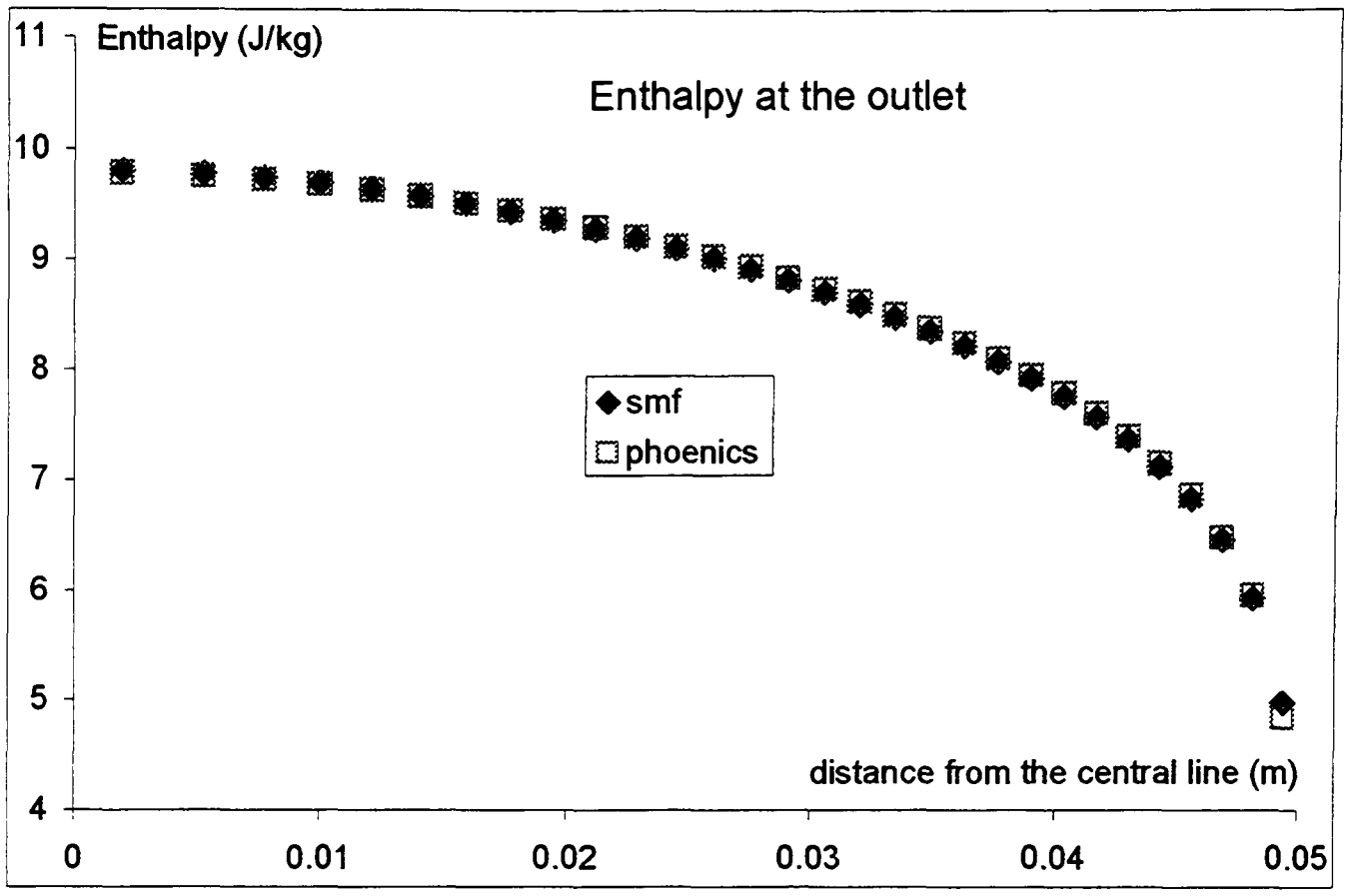


Figure 15 - Enthalpy plotted against distance from duct axis at the outlet.

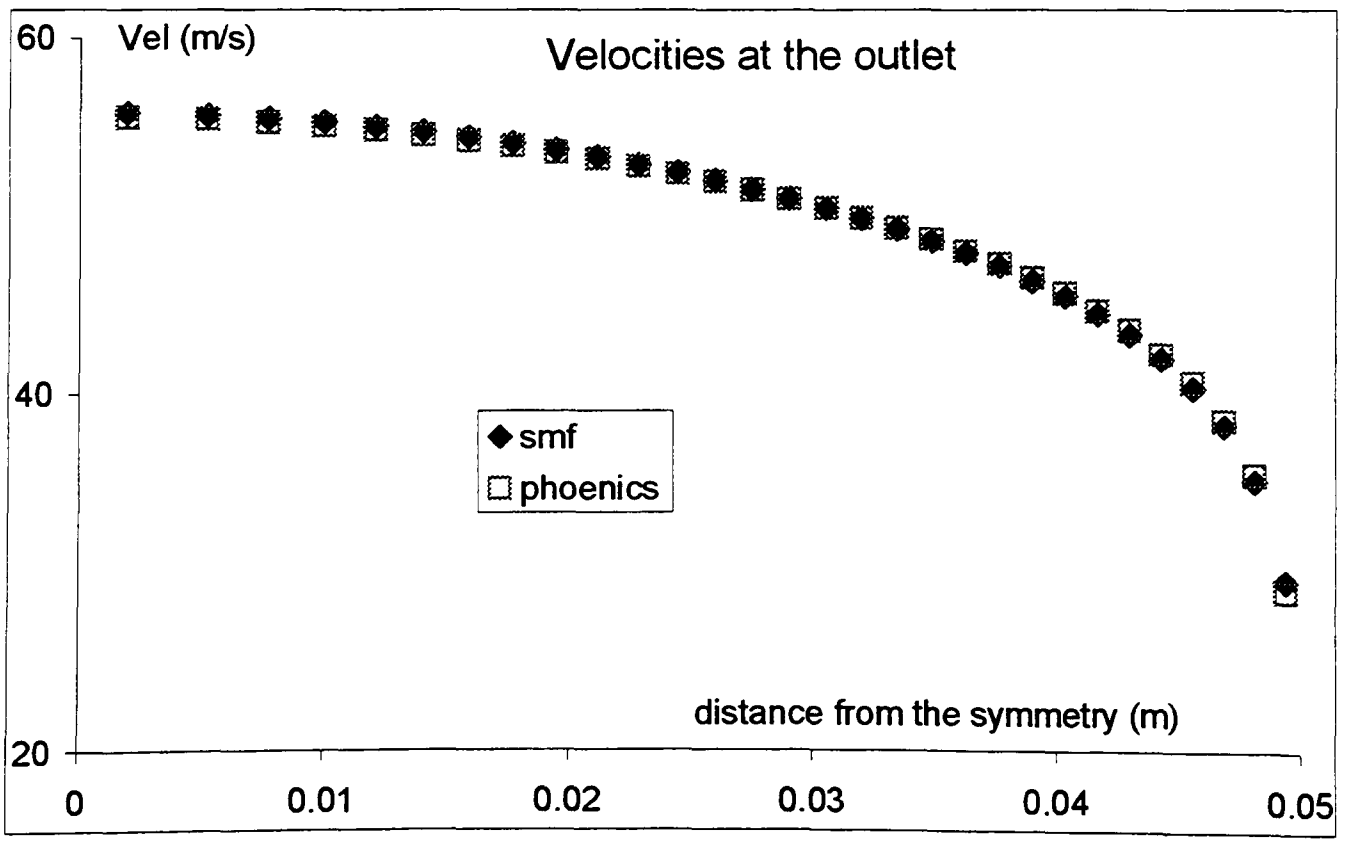


Figure 16 - Velocities plotted against distance from the duct axis at the outlet.

The results indicate that the turbulent heat transfer function of SMARTFIRE and PHOENICS produce similar results.

2.6 Turbulent buoyancy flow in a cavity.

This test case examines the turbulence model, turbulent heat transfer and buoyancy model. The test case is a standard test case which has been used by a number of other investigators [3] and forms part of the PHOENICS test library. The turbulence model used by SMARTFIRE is based on the model of Launder and Spalding [4].

The geometry used for this case is depicted in Figure 17 below.

Fluid properties

conductivity is 2.852158e-02 (W/mK)
density is 1.071 (kg/m^3)
specific heat is 1.008e+03 (J/kgK)
viscosity is 2.0383e-05 (kg/ms)
thermal expansion is 3.029385e-03 (1/K).

Boundary conditions

hot wall (t_h): constant temperature (353.0 K)
cold wall(t_c): constant 307.2 (K).
The other walls are adiabatic.

The cell budget is 14400(120×120) with non-uniformly distributed mesh.

The Boussinesq approximation is used to model the buoyancy. As the flow lies in the low Mach number region (i.e. subsonic) and there is a small temperature difference between the walls, then the Boussinesq approximation is extremely good and therefore the use of a fully turbulent treatment is unnecessary.

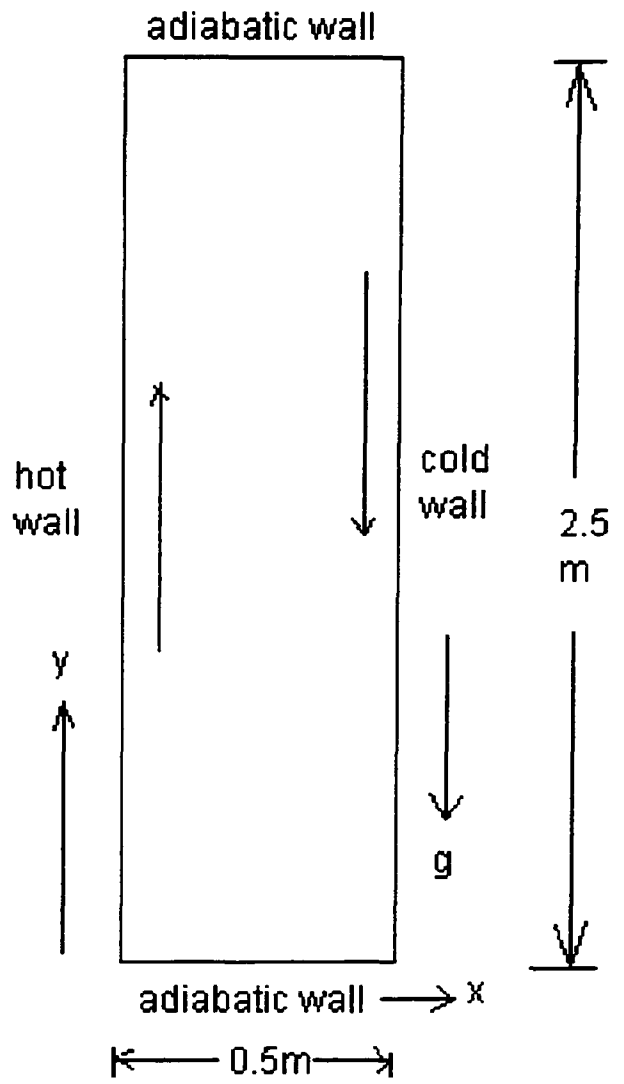


Figure 17 – Configuration for buoyancy flow in a duct

2.6.1 Results

In this test case, SMARTFIRE predictions for the vertical velocity profile (mid way up the test cell), temperature along the vertical centre line, turbulent fluctuations and turbulent viscosity are compared with published experimental data. The results produced by SMARTFIRE (Figure 19) when compared with the results published in the reference [3] (Figure 18, Figure 20, Figure 22 and Figure 24) demonstrate that the SMARTFIRE results are very close to the published results. In Figure 18, Figure 20, Figure 22 and Figure 24; 'LB' stands for the turbulence model of Lam and Bremhorst [5], 'PRESENT' stands for the turbulence model devised by Davidson from which the figures shown here are taken [3], and 'EXP' stands for experimental data obtained by Cheesewright et al [6]. It should be noted that the LB [5] and Davidson [3] turbulence models are more advanced than the current SMARTFIRE turbulence model [4].

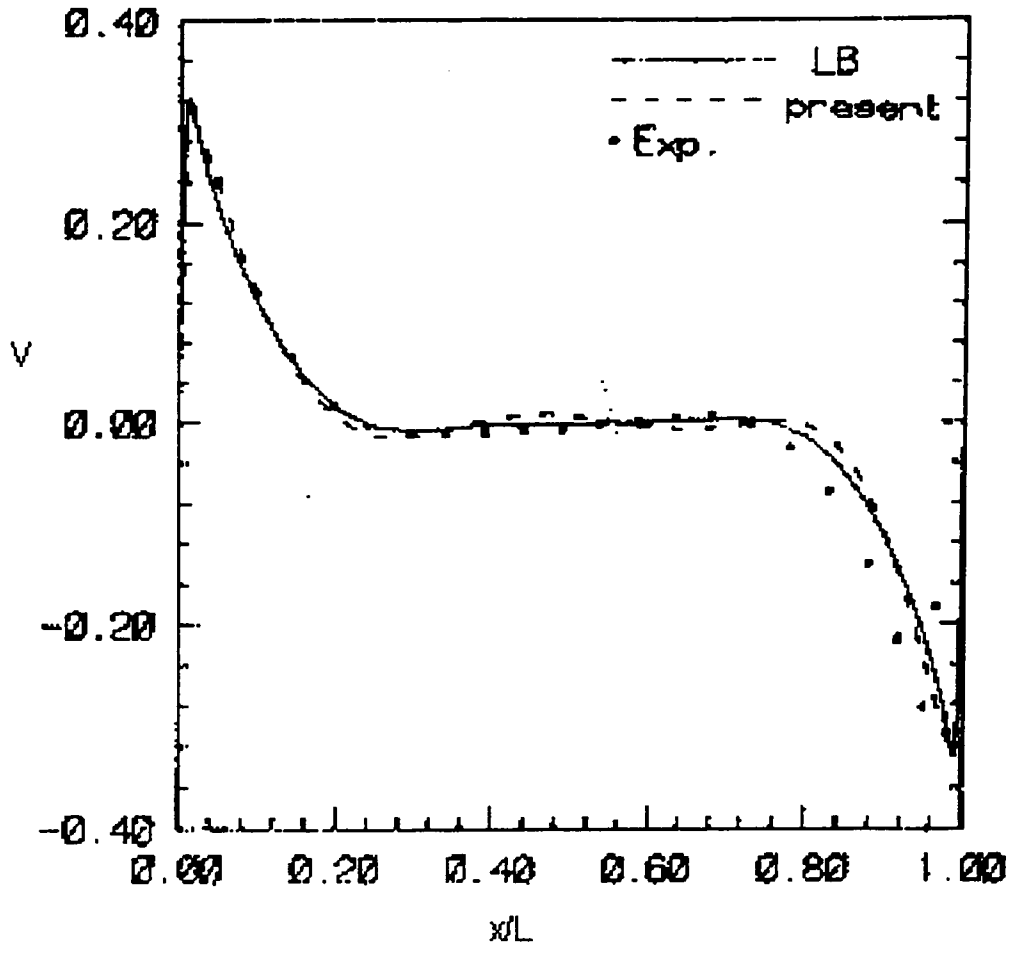


Figure 18 - Published [3] velocity (y direction) profiles at y/H = 0.5

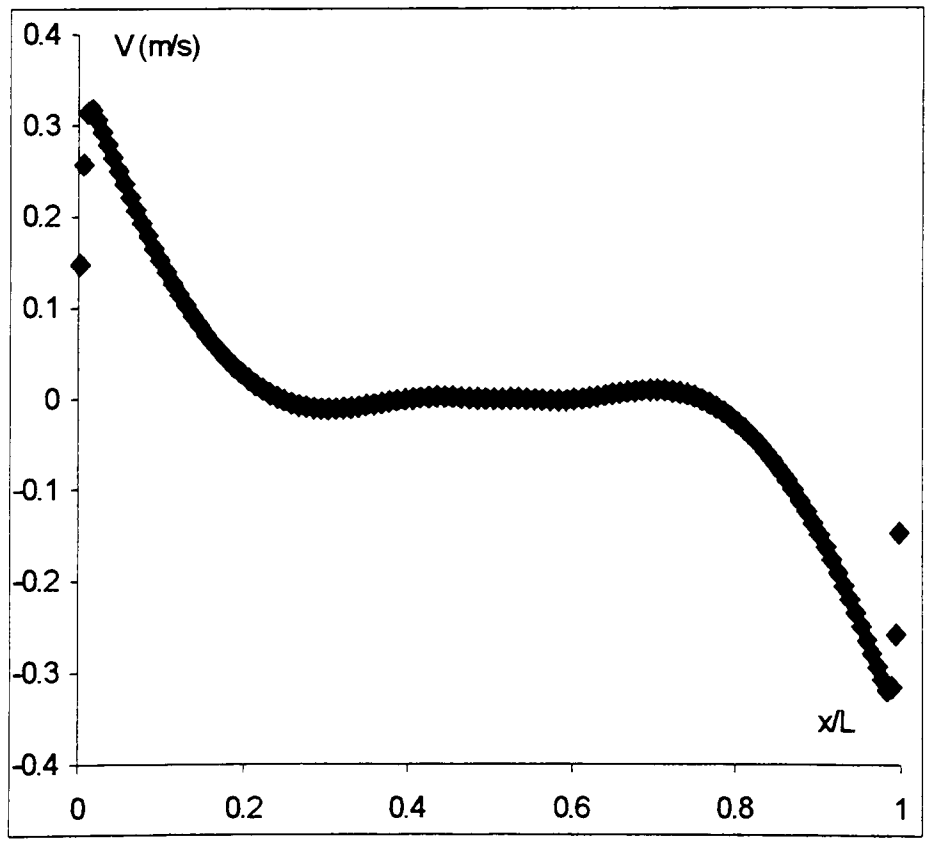


Figure 19 - Predicted velocity (y direction) profiles at y/H = 0.5

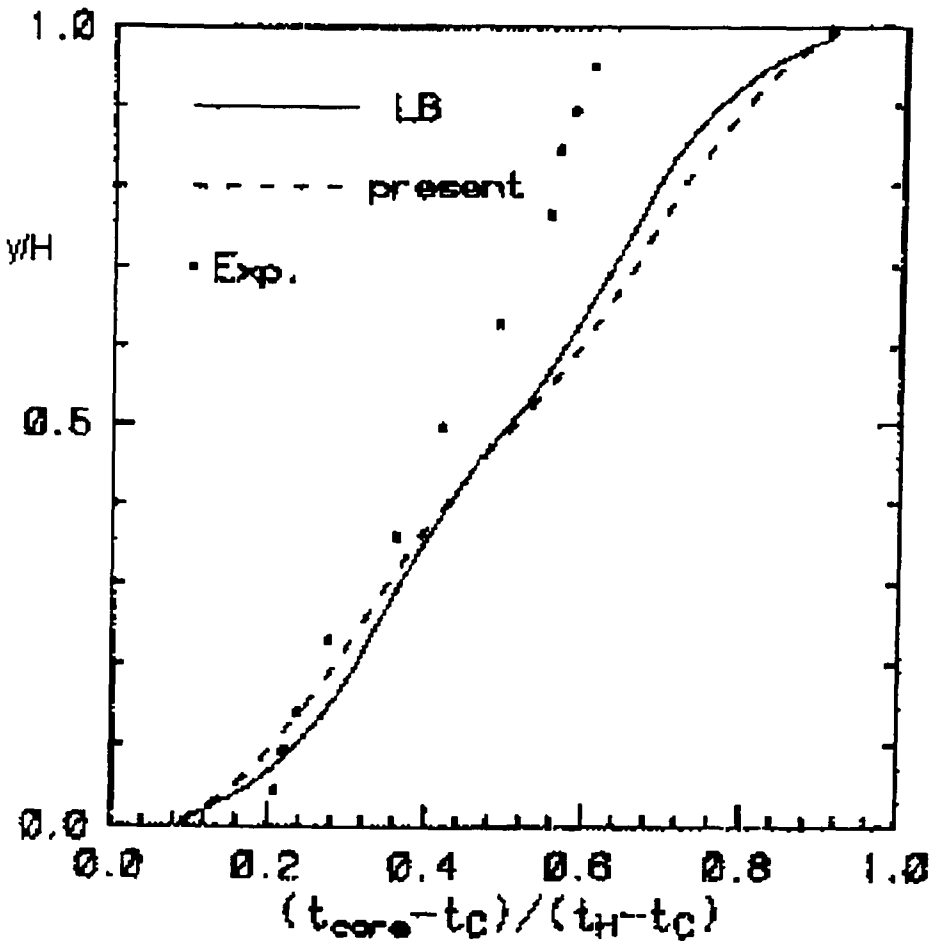


Figure 20 - Published [3] local temperatures along the vertical central line.

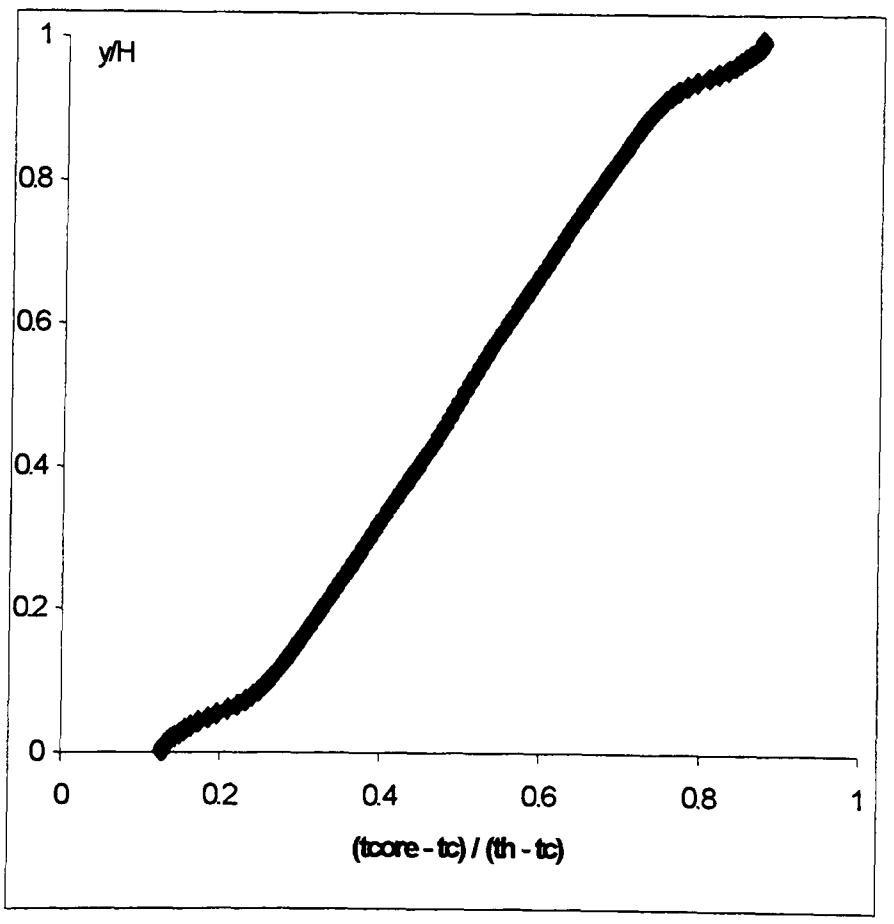


Figure 21 – SMARTFIRE predicted local temperatures along the vertical central line.

Note: T_c and T_h represent temperatures at the cold and hot wall respectively.

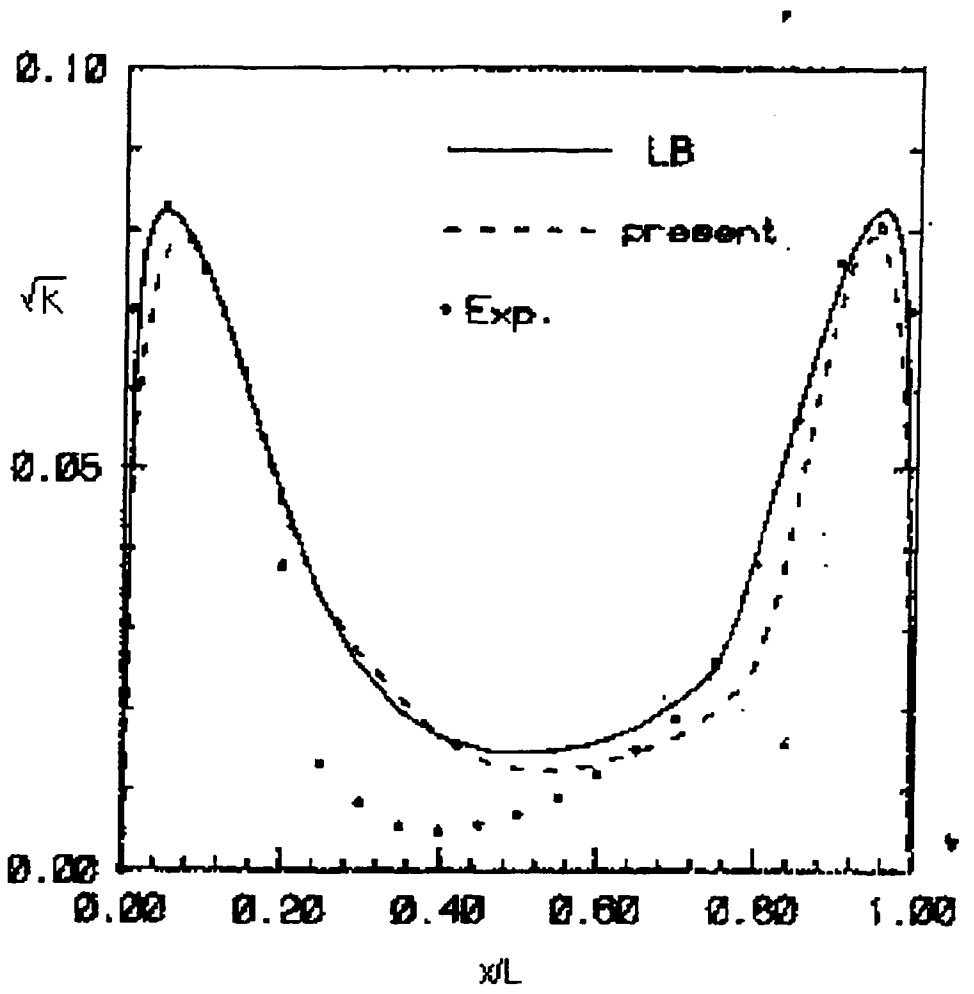


Figure 22 - Published [3] turbulent fluctuations, $\sqrt{k}$, at y/H = 0.5

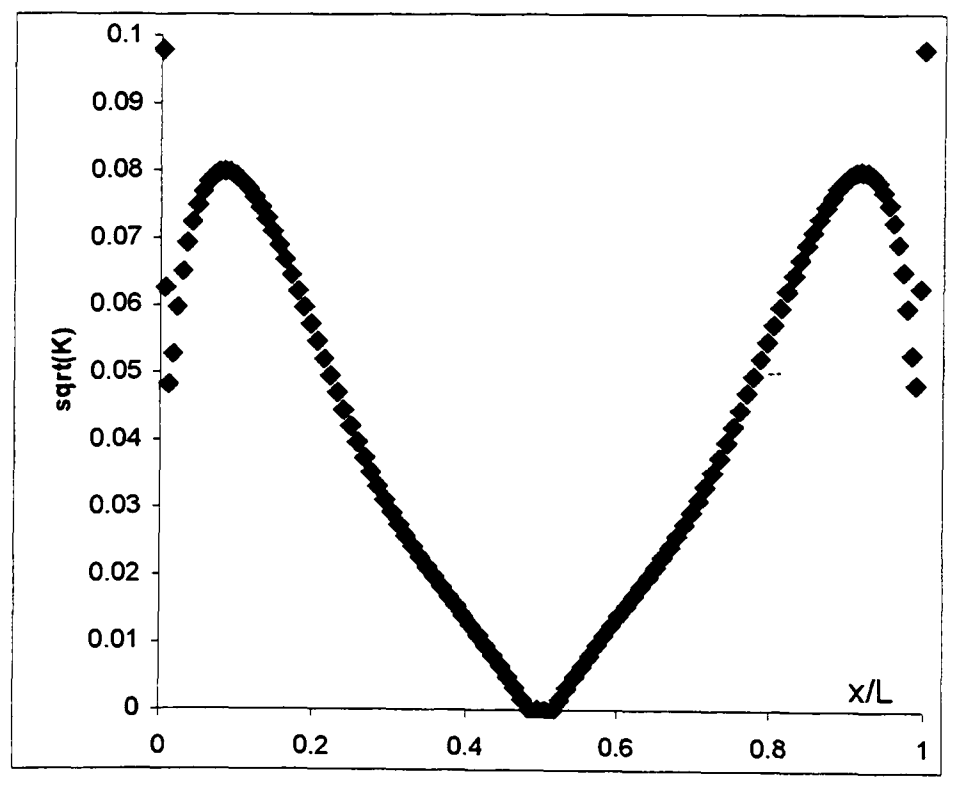


Figure 23 – SMARTFIRE predicted turbulent fluctuations, $\sqrt{k}$, at y/H = 0.5

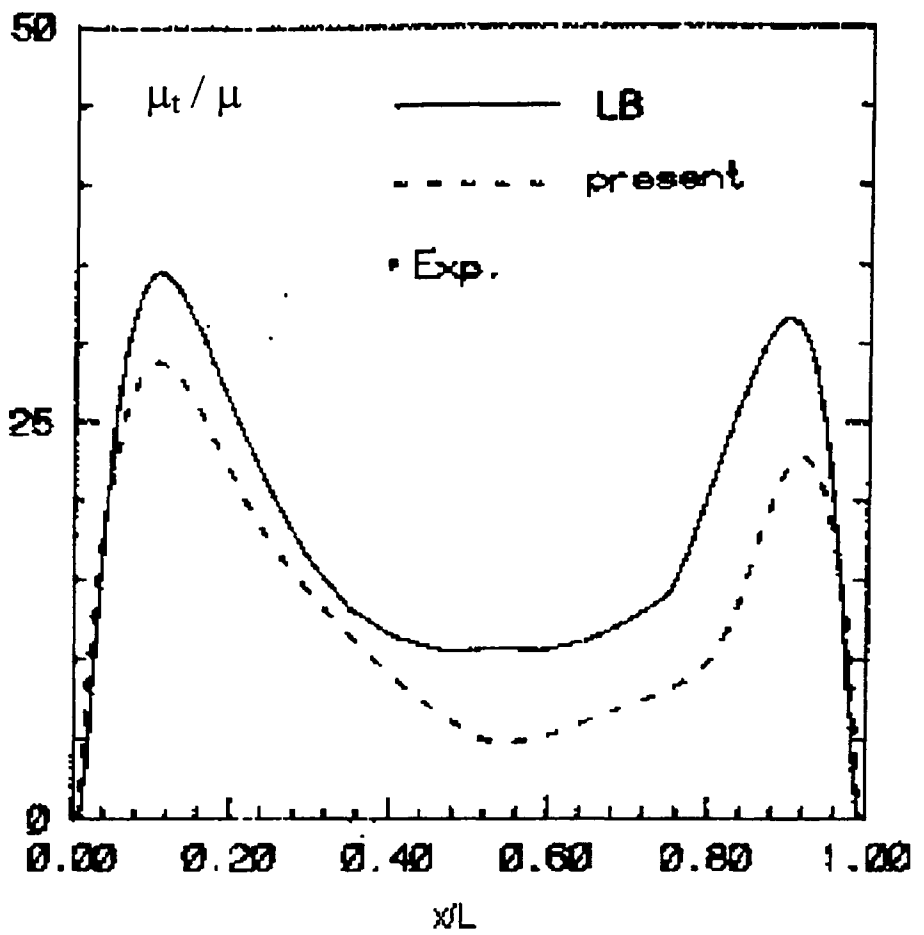


Figure 24 - Published [3] turbulent viscosity scaled with the laminar viscosity at y/H = 0.5.

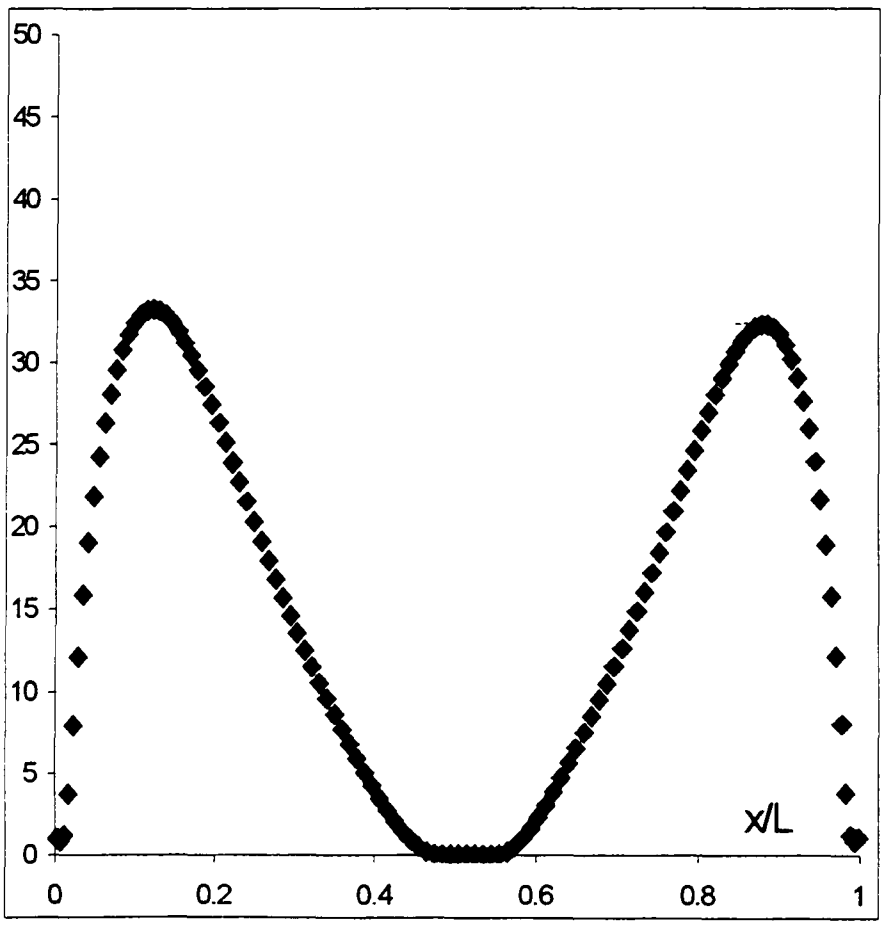


Figure 25 – SMARTFIRE predicted turbulent viscosity scaled with the laminar viscosity at y/H = 0.5.

3 Fire Validation Cases

3.1 Steckler Room Fire

Steckler et al [7] carried out a series of fire experiments within a compartment to investigate fire induced flows. The experimental data obtained from these fire tests have been used as part of the validation process for fire models both zone and field. The data represents non-spreading fires in small compartments. A series of 45 experiments were conducted by Steckler et al. to investigate fire induced flows in a compartment measuring 2.8m × 2.8m in plane and 2.18m in height (see Figure 26). The walls and ceiling were 0.1m thick and they were covered with a ceramic fibre insulation board to establish near steady state conditions within 30 minutes. The series of experiments consisted of a gas burner placed systematically in 8 different floor locations with a variety of single compartment openings ranging from small windows to wide doors. The door openings are 0.24m to 0.99m. The 0.3m diameter burner was supplied with commercial grade methane at fixed rates producing constant fire strengths of 31.6, 62.9, 105.3 and 158 kW.

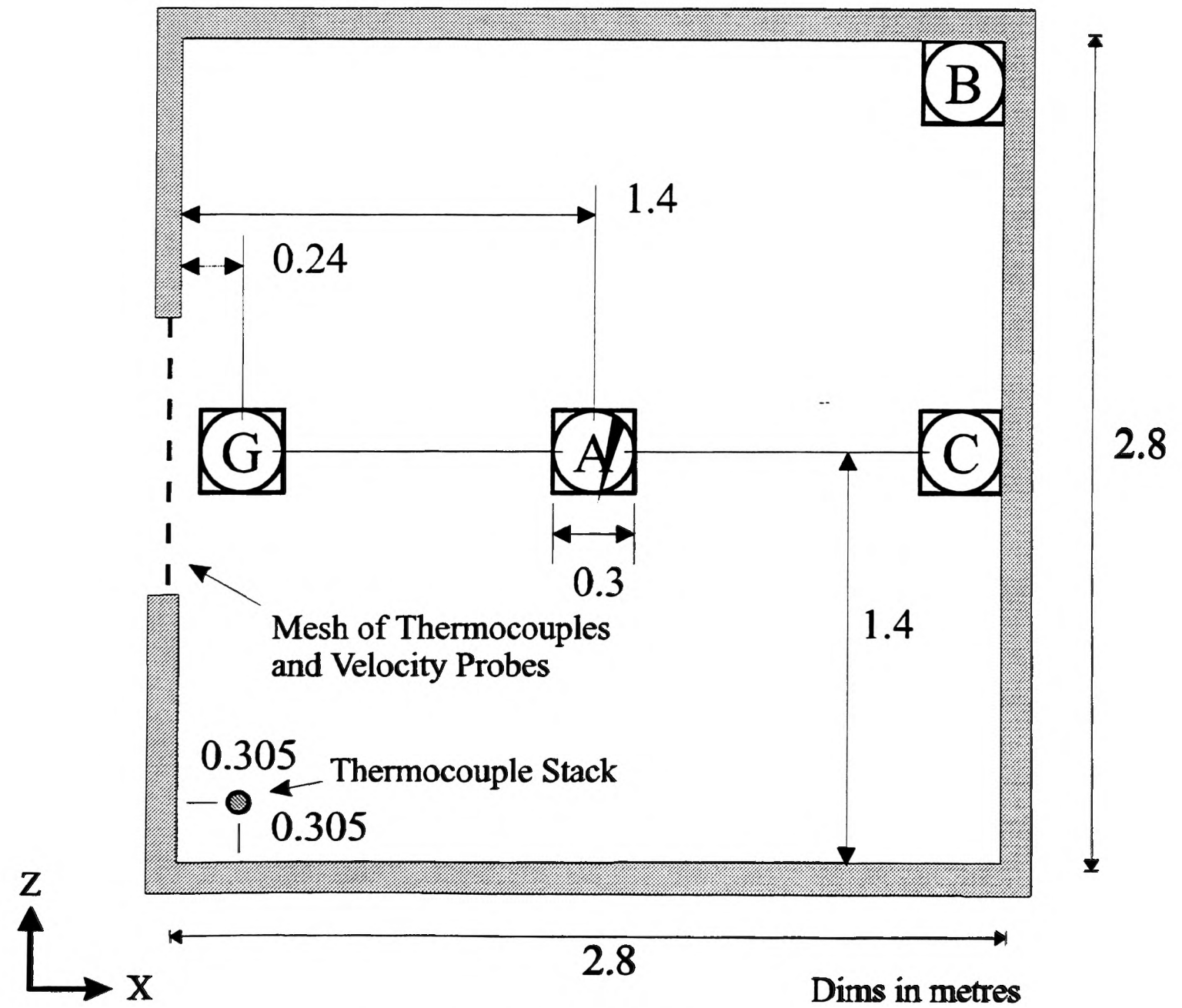


Figure 26 - Configuration of Steckler room

Bi-directional velocity probes and bare-wire thermocouples were placed within the room opening on a two-dimensional grid of 28 to 144 depending on the size of the opening to measure velocities and temperatures within the centre of the door jamb. The measured velocities may be subject to up to 10 percent error. In addition, a stack of aspirated thermocouples was placed in the front corner of the room to measure the gas temperature profile.

In the test selected here, the door measured 0.74m wide and 1.83m high and the fire, which was centrally located on the floor (position A in Figure 26), was represented by a gas burner measuring 0.3m in diameter. The burner power was 62.9 kW.

In the CFD case it is assumed that the fire is a volumetric heat source measuring 0.3m x 0.3m x 0.3m with a heat release rate of 62.9 kW.

3.1.1 Results

The SMARTFIRE results are compared with the experimental data and predictions made using FLOW3D (see Figure 27, Figure 28 and Figure 29). Both SMARTFIRE and FLOW3D used 200 one second time steps. The mesh budget consisted of 9918 (29×18×19) cells. In these calculations it was assumed that the walls were composed of common bricks. The six-flux radiation model was used with a temperature dependent absorption coefficient.

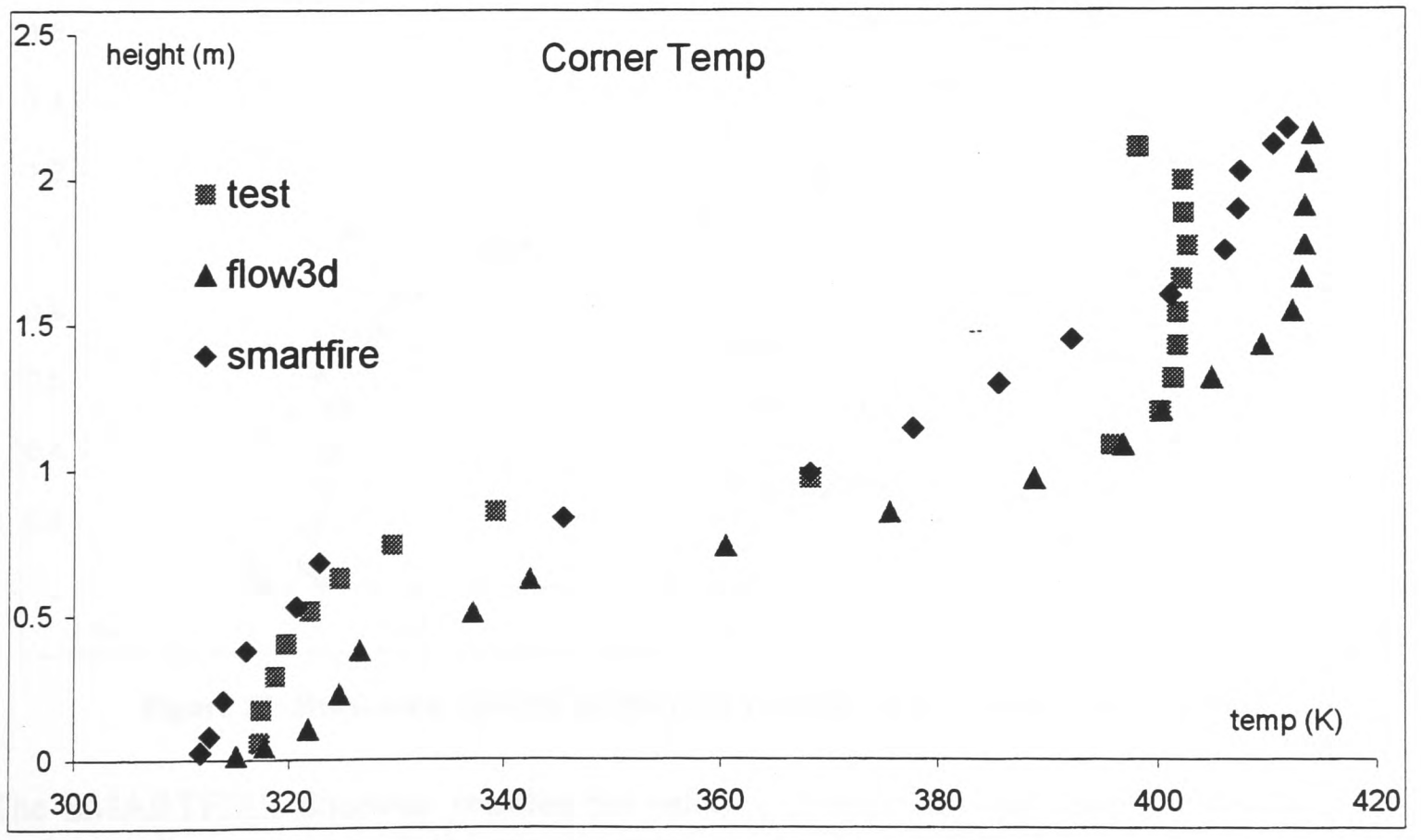


Figure 27 –Vertical Corner Stack temperatures at 0.305 from the front wall and side.

Both FLOW3D and SMARTFIRE give reasonable agreement with the experimental result for the vertical thermocouple stack in the corner of the room (Figure 27). Neither SMARTFIRE or FLOW3D capture the hot stratification layer very well but this can at least in part be attributed to the relative coarseness of the mesh.

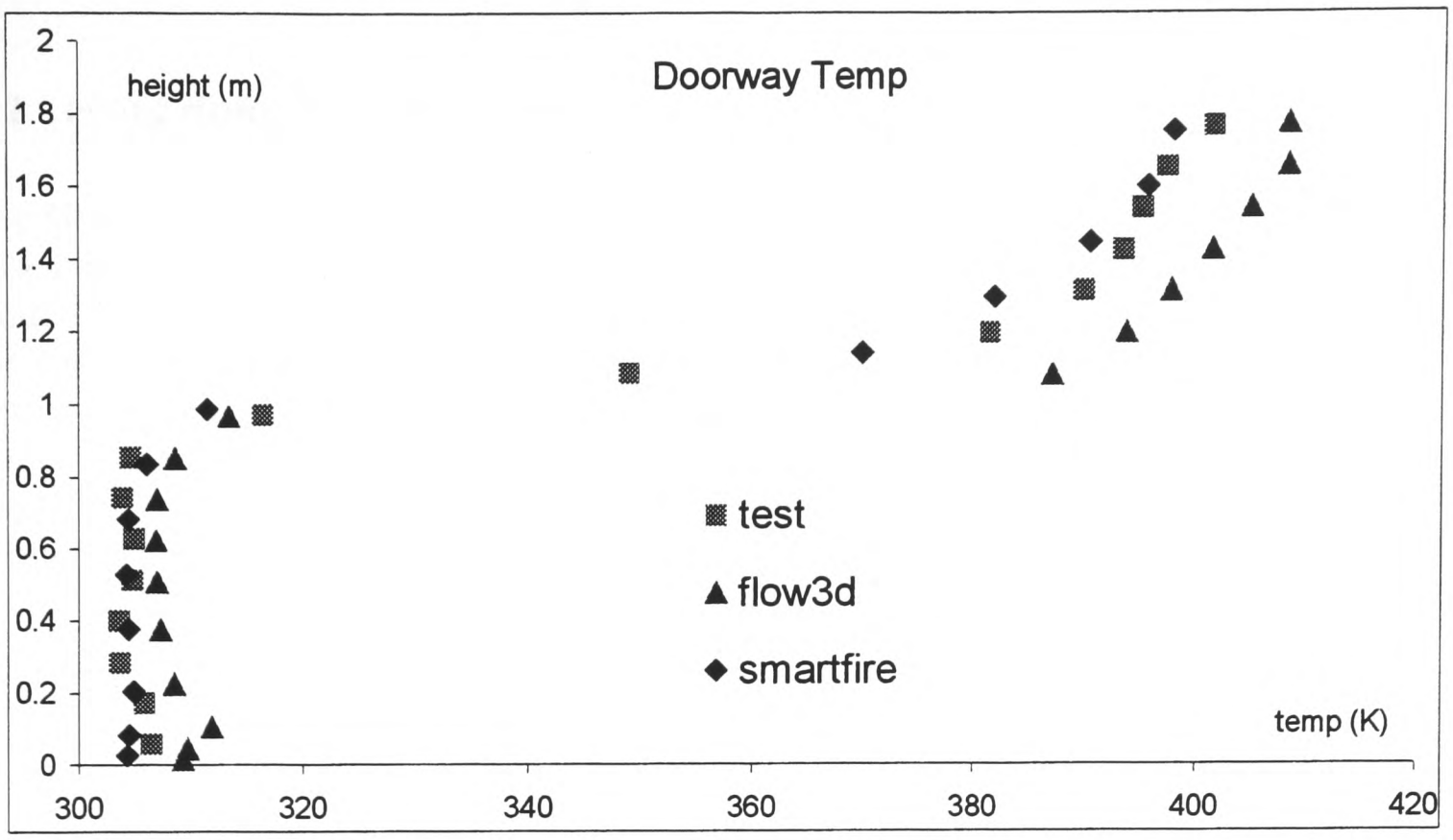


Figure 28 -Vertical Doorway temperature profile in the middle of the door.

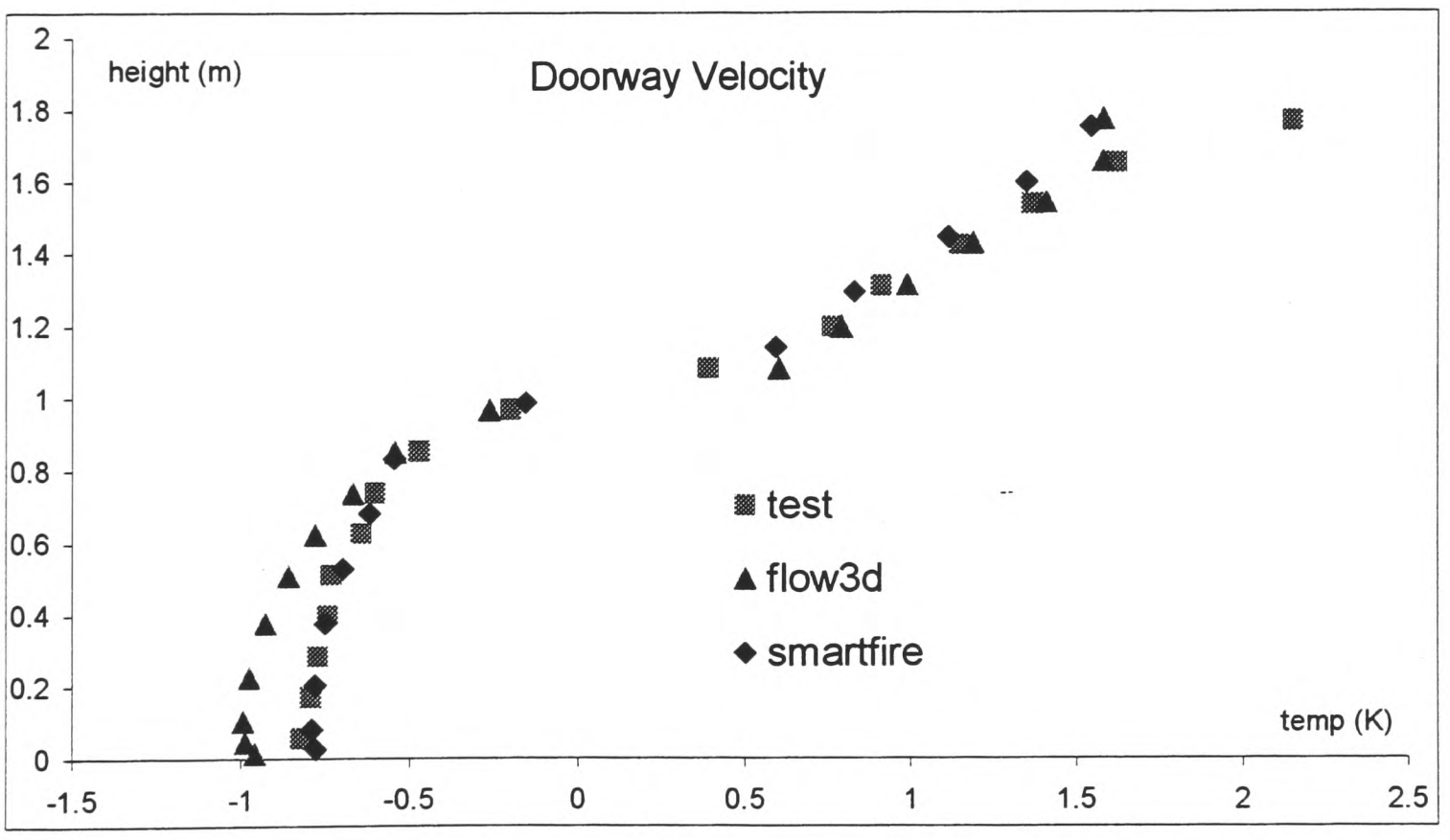


Figure 29 -Horizontal velocity profile for a vertical stack in the middle of the door.

The SMARTFIRE doorway profiles for velocity (Figure 28) and temperature (Figure 13) are in good agreement with the experimental results. The SMARTFIRE results appear to be marginally superior to the FLOW3D results in this instance.

3.2 Hong Kong Airport Case

The Hong Kong airport test case involves the Arup Fire "Cabin Concept". This case has arisen due to a journal paper Prof. Chow [8] that attempts to simulate this case using a zone model. The compartment is completely open apart from a ceiling unit. The fire is located on the floor at the centre of the building. The prescribed fire volume is 1m x 1m x 1m. The fire power is defined by the standard method, i.e., $H = 0.188t^2(kW)$ (i.e. t squared fire). The compartment is 5m(wide) × 5m(long) × 3m(high). The case has been simulated using both SMARTFIRE and FLOW3D and the six-flux radiation model. Comparisons of model predictions at 110 seconds are plotted in Figure 30, Figure 31 and Figure 32. It should be noted that this is a hypothetical case for which there is no experimental data.

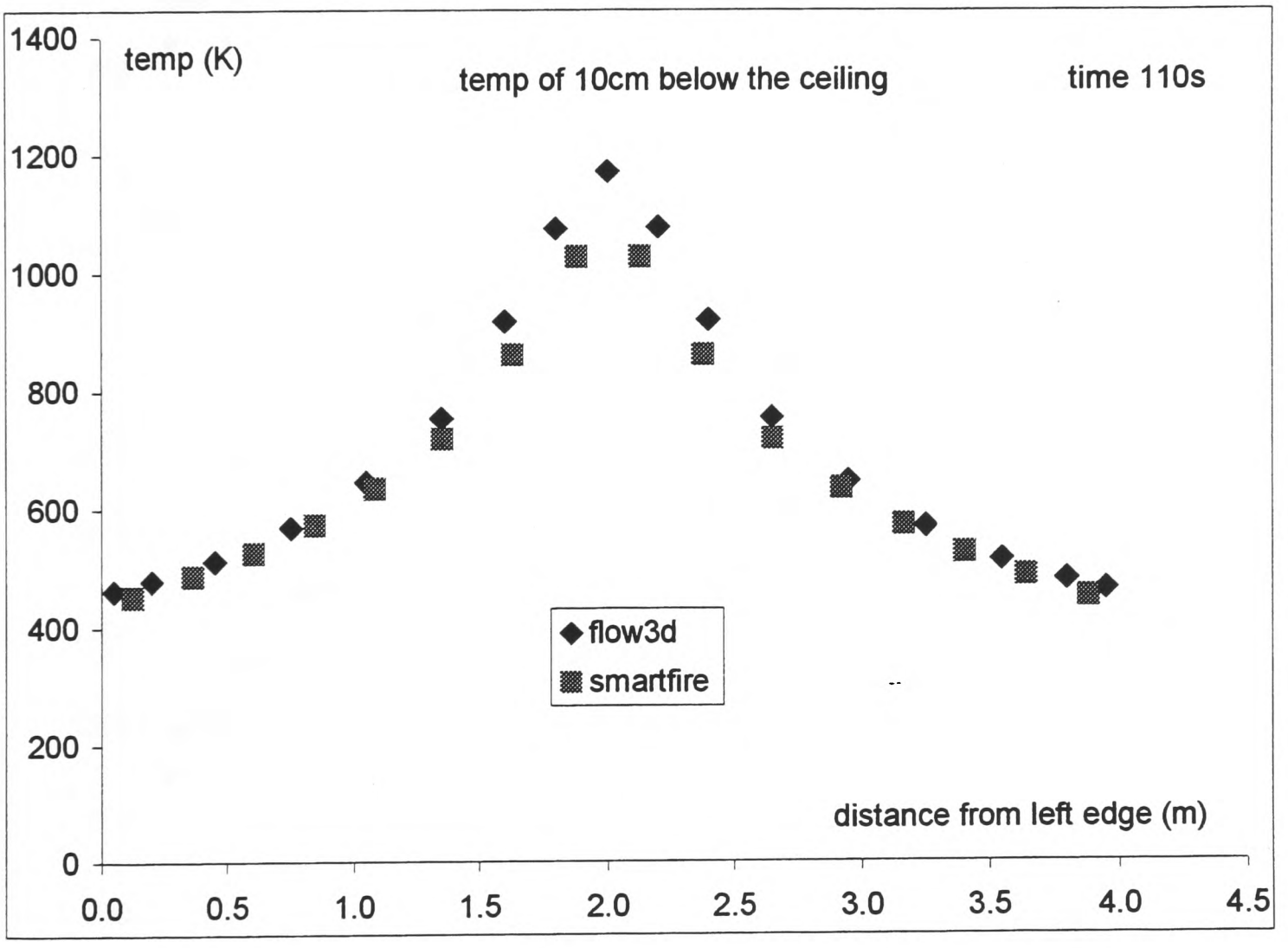


Figure 30 - Temperature profile across the cabin 0.1m below the ceiling.

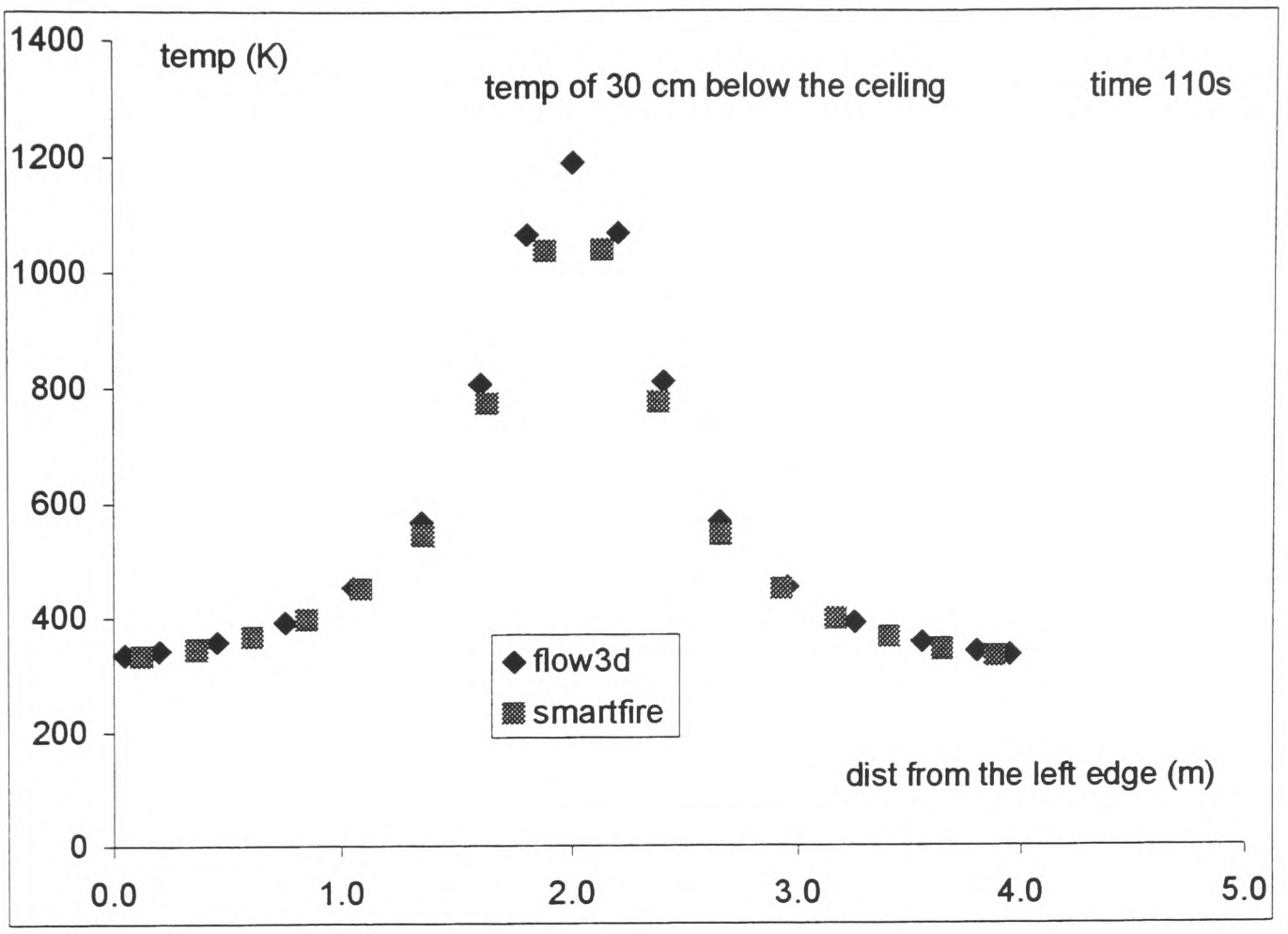


Figure 31 -Temperature profile across the cabin 0.3m below the ceiling.

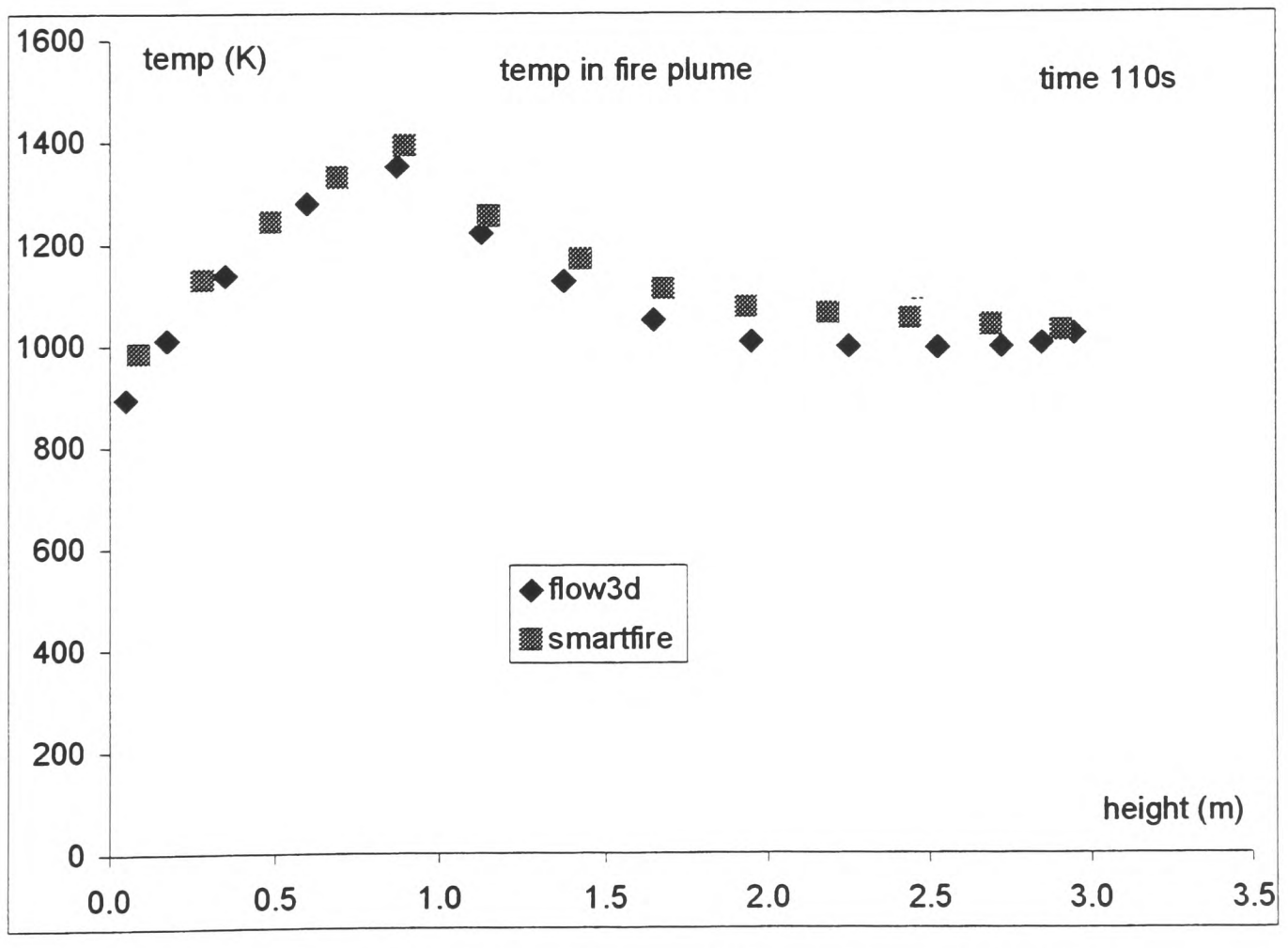


Figure 32 -Vertical temperature profile in the centre of the fire.

From the above figures it can be seen that both SMARTFIRE and FLOW3D produce very similar results.

3.3 Simulations for LPC-RUN-007.

This test case arises from a fire test conducted by the Loss Prevention Council (LPC) [9]. The test is a burning wood crib within an enclosure with a single opening. The test compartment is illustrated below in Figure 33 and Figure 34 and had a floor area of 6m x 4m and a 3.3m high ceiling. The compartment contained a door (vent) measuring 1.0m x 1.8m located on the rear 6m x 3.3m wall. The walls and ceiling of the compartment were made of fire resistant board (Asbestos) which were 0.1m thick.

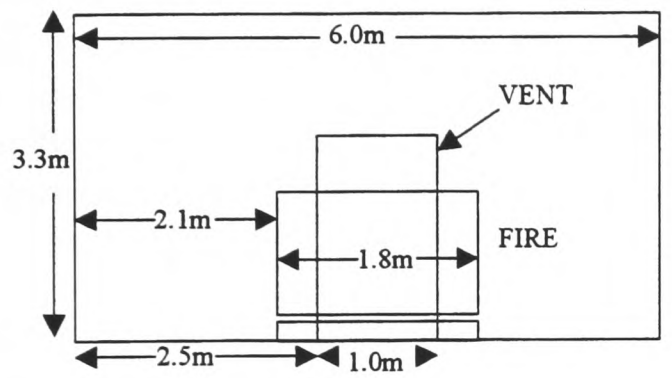


Figure 33 - Front view of LPC 007 configuration

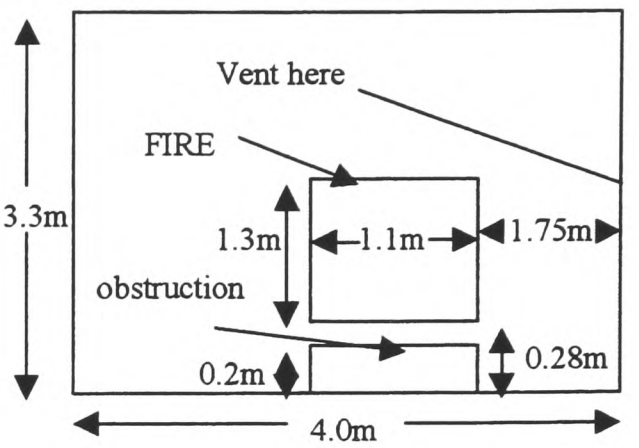


Figure 34 - Side view of LPC 007 configuration

Within the model, the fire is described by a volumetric heat release rate prescribed in a region measuring 1.8m x 1.3m x 1.1m. The fire was located 0.28m above the floor. A steel obstruction was placed directly below the fire. The heat release rate was prescribed as:

$HRR = 50.0+0.0082t^2$ (kW), for t <300 seconds;

HRR =3000 (kW), for t > 300 seconds.

Two computational meshes were used for the analysis, one consisting of approximately 10,000 cells and another consisting of 21054 cells. The results from these two simulations were in good agreement with one another. The results presented here are for the mesh consisting of 21054 cells.

Two FLOW3D simulations were carried out, in one simulation (FLOW3D_HEAT), the treatment of the fire was the same as SMARTFIRE, while in the other (FLOW3D_CHEM), a gaseous combustion model developed by FSEG was used. In the FLOW3D_CHEM simulation, the volumetric heat release rate was replaced by a volumetric mass loss rate of the wood crib whose time dependent curve was provided by the LPC report. The heat released was determined by oxygen consumption. The heat released from consuming oxygen per unit mass is approximately 13.1 MJ/kg.

The FSEG developed six-flux radiation model was used in both cases. The absorption coefficient took the following form:

$a = 0.01$, if $T < 323$K;

$a = 0.01+0.305/377(T-323)$, if $323 <= T < 700$;

$a = 0.315+0.315/700(T-700)$, if $T > 700$.

where T is the gas temperature. The wall emissivity was assumed to be 0.8.

3.3.1 Results

The corner stack gas temperatures (at various heights above the floor), predicted by SMARTFIRE and FLOW3D along with the experimental results are plotted in Figure 35. While the numerical predictions exceed the measured temperatures they appear to be in reasonable agreement. After about 600 seconds the SMARTFIRE and FLOW3D_HEAT simulations reached steady-state. As the heat release rate within the model is constant after 300 seconds, it is expected that a steady-state will be attained in the numerical predictions. However, the FLOW3D_CHEM simulation produced steadily increasing temperatures since the total heat release rate is determined by the combustion rate that depends on the concentration of fuel and oxygen and the turbulence mixing rate. The FLOW3D_CHEM simulation did not improve the magnitude of the temperature but it predicted a more realistic trend in temperature variations.

The plume temperatures predicted by SMARTFIRE and FLOW3D along with the experimental results are plotted in Figure 36. The numerical predictions appear somewhat higher than the measured values.

The same physics and mesh size was used in the FLOW3D_HEAT simulation as was used in the SMARTFIRE simulations. Up to approximately 600 seconds both codes produced near identical results. However, the FLOW3D_CHEM simulation produced a much improved prediction of not only the trends of the plume temperature variations but the magnitude of the temperatures as well. It is also worth noting that the trends in the predicted plume temperatures are very similar to those observed. It should be noted that the heat release rate was estimated by the mass loss rate of the wood crib, this measurement was unreliable. Furthermore, flames were seen to emerge from the test compartment indicating that a significant amount of energy was being released outside the fire compartment. The volumetric heat release rate models did not represent this. One way of taking this into account is to estimate the amount of combustion taking place outside the compartment and reduce the internal prescribed heat release rate by an appropriate amount. In addition, the material properties of the walls, floor and ceiling are only approximated, as is the value for the absorption coefficient.

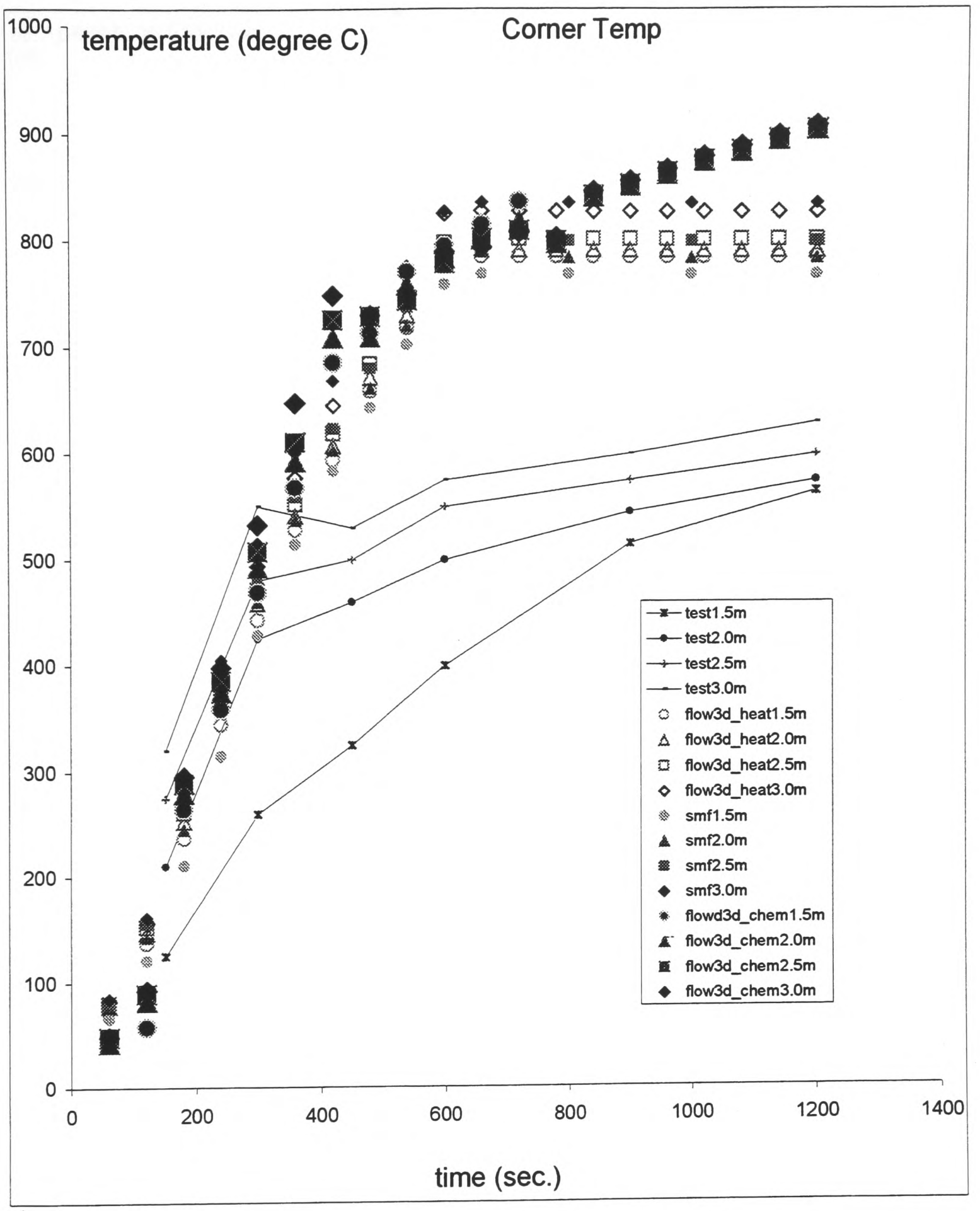


Figure 35 - The corner gas temperature predicted by the simulation.

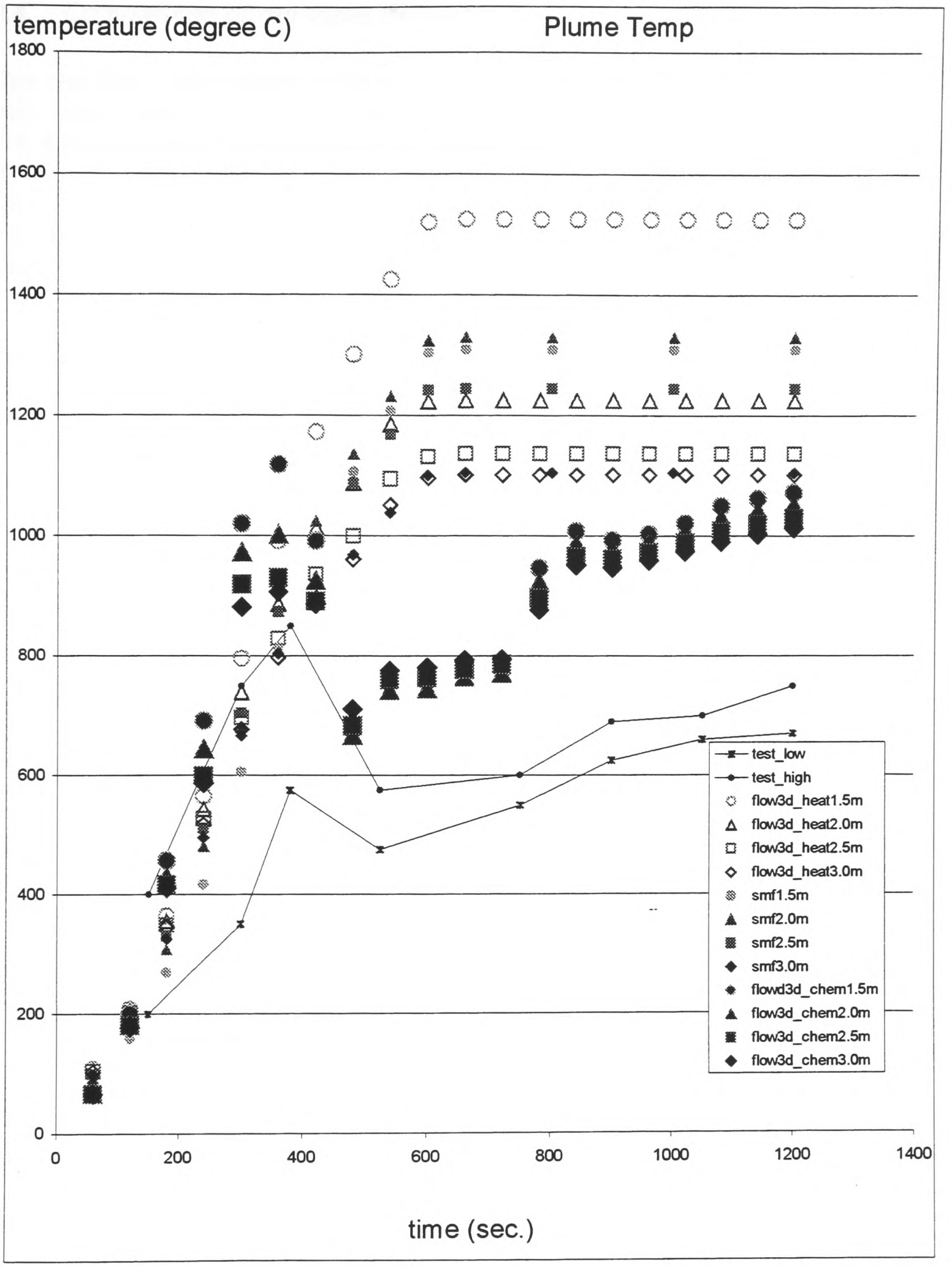


Figure 36 - The predicted plume temperatures.

It is concluded that the introduction of the gaseous combustion model results in improved predictions of corner and plume temperatures, in particular the trend prediction of the fire development. The implementation of the gaseous combustion model in SMARTFIRE is under way now. With combustion excluded, both SMARTFIRE and FLOW3D produce similar results and both of these overpredict the maximum temperatures observed in the experiment.

3.4 Comparison of run-times between PHOENICS and SMARTFIRE.

This test case is performed in order to compare the run-time performance of SMARTFIRE with other CFD codes. 10 seconds of a fire simulation using the SMARTFIRE and PHOENICS codes are compared. The run-time comparison was performed using a 90 MHz Pentium PC with 32 Mb of RAM. The test case scenario consisted of the STECKLER room with 0.74m wide door as used in Section 3.1. Both SMARTFIRE and PHOENICS used the same mesh consisting of 7920 (20×18×22) cells. For both codes a 10 second transient simulation was executed with one second time steps; each time step consisted of 100 outer iterations and each outer iteration consisted of 10 inner iterations.

SMARTFIRE required 187 minutes to complete the above case while PHOENICS only required 76 minutes. Therefore the time ratio of SMARTFIRE to PHOENICS is 2.46 : 1.0. Since SMARTFIRE is an unstructured code and PHOENICS is structured, it is expected that PHOENICS would be more efficient. However, the SMARTFIRE performance is better than the observed 3.0:1.0 of the time ratio found in other unstructured codes.

3.5 Simulation of Steckler room fire using large cell budget.

The set up for this case is similar to that in section 3.1 except the cell budget has been increased and a natural symmetry plane has been exploited. The SMARTFIRE results are compared with the experimental data and the results produced when a smaller cell budget (section 3.1.1) is used. SMARTFIRE simulated 200 time steps with 1 second time steps. The mesh budget was 49980 (49×34×30). Since a symmetry plane across the centre of the fire and the doorway is employed in this simulation, the actual cell budget is equivalent to 99960 (if a whole room simulation were conducted). The walls are assumed to be common brick with 0.8 emissivity. The SMARTFIRE six-flux radiation model is used with a temperature dependent absorption coefficient.

3.5.1 Results

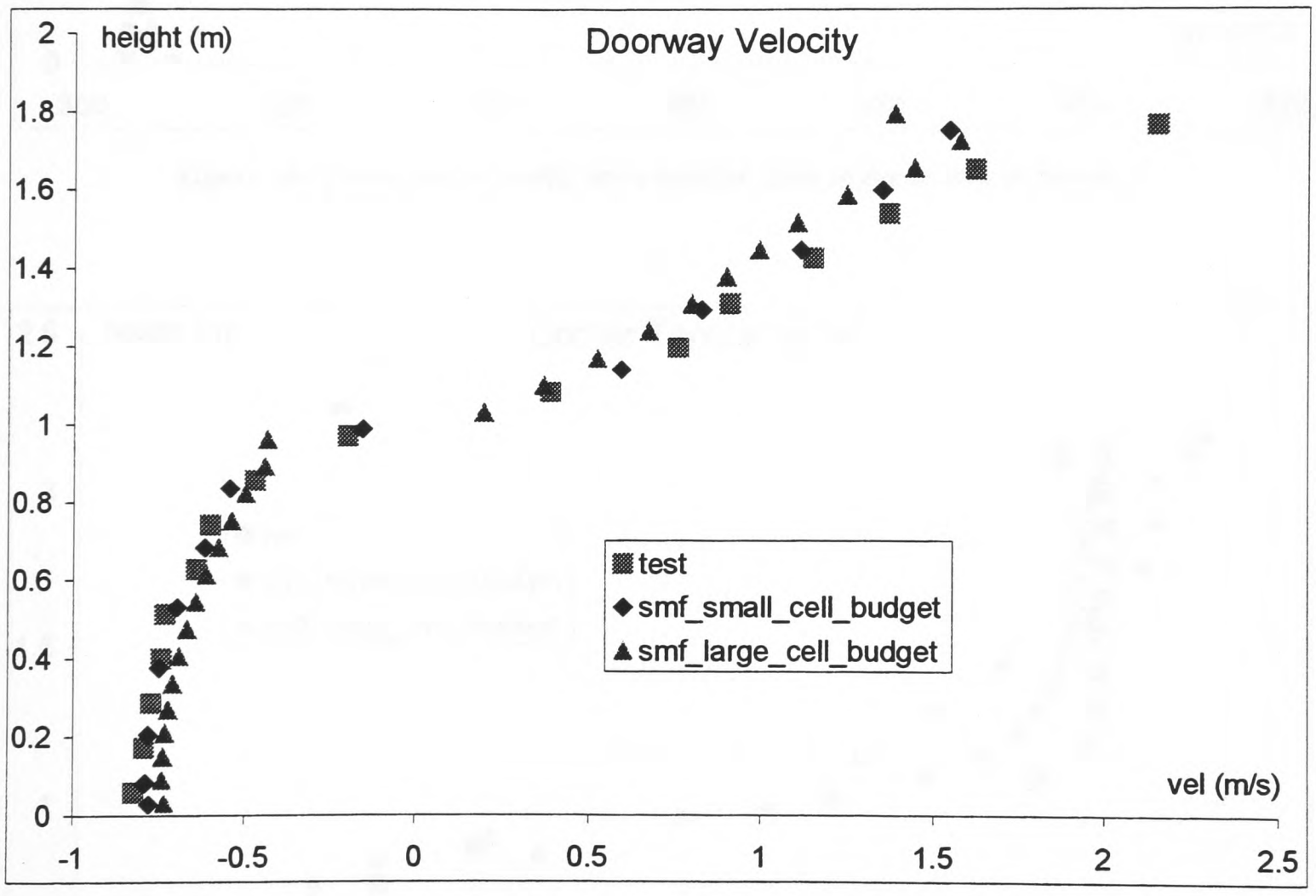


Figure 37 -Horizontal velocities for a vertical stack in the middle of the door.

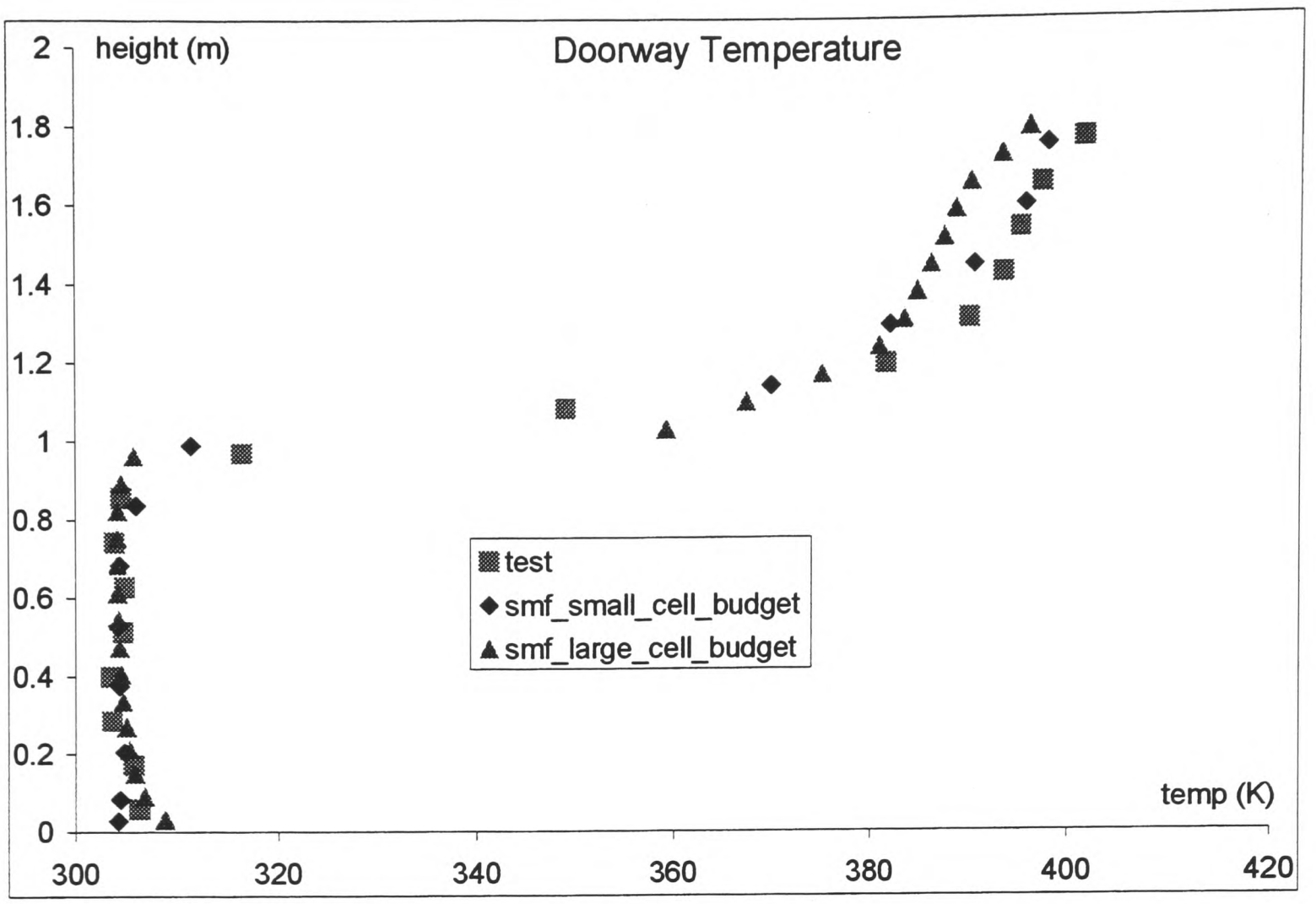


Figure 38 -Temperature profile for a vertical stack in the middle of the door.

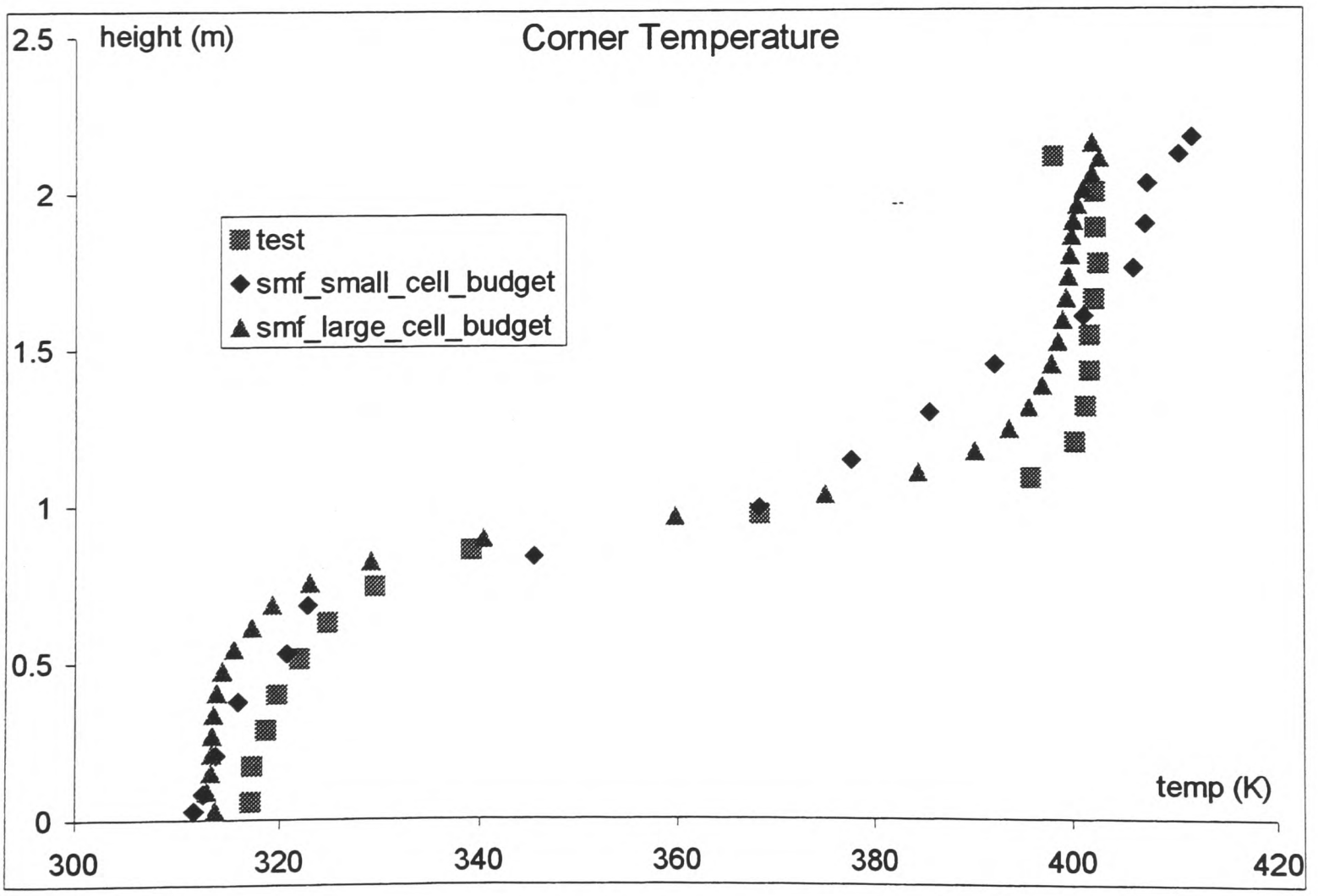


Figure 39 -Temperature profile for a vertical stack 0.305m from the front and side walls.

Compared with the simulation using a smaller cell budget (section 3.1), the velocity (Figure 37) and temperature (Figure 38) profiles of the doorway from the large cell budget simulation do not offer a significant improvement. The shape of the temperature profile of the corner stack (Figure 39) is considerably improved with the hot stratified layer being well captured by SMARTFIRE.

4 References

1) Kim J., Kline S. and Johnston J., "Investigation of a Reattachment Turbulent Shear Layer: Flow over a Backward-Facing Step", Transactions of the ASME, Journal of Fluids Engineering, 102, pp 302-308, 1980.

2) Eaton J. and Johnston J., "A Review of Research on Subsonic Turbulent Flow Reattachment", AIAA, Paper AIAA-80-1438, 1980.

3) Davidson L., "Calculation of the turbulent buoyancy-driven flow in a rectangular cavity using an efficient solver and two different low Reynolds number κ-ε turbulence models", Numerical Heat Transfer, Vol. 18, pp 129-147, 1990.

4) Launder B. and Spalding D., "The numerical computation of turbulent flow", Comp. Methods. In Applied Mech. Eng., 3, p. 269, 1974.

5) Lam C. and Bremhorst K., "A modified form of the κ-ε model for Predicting Wall Turbulence", J. of Fluid Eng., Vol. 103, pp 456-460, 1981

6) Cheesewright R., King K. and Ziai S., "Experimental data for the validation of Computer Codes for the prediction of Two-Dimensional Buoyant Cavity Flows", Significant Questions in Buoyancy Affected Enclosure or Cavity Flows, Vol. HTD-60, pp 75-81, ASME, New York, 1986

7) Steckler K., Quintiere J. and Rinkinen W., "Flow induced by fire in a compartment", NBSIR 82-2520, National Bureau of Standards, 1982.

8) Chow W., "On the "Cabin" fire safety design concept in the new Hong Kong Airport terminal buildings", J. of Fire Science, Vol. 15, pp 405-423, 1997.

9) Glocking J., Annable K. and Campbell S., "Fire Spread in multi-storey buidings – *Fire break out from heavyweight unglazed curtain wall system – Run 007*", LPC Laboratories Report TE 88932-43, 25 Feb 1997.

11.2 Copy of Journal Paper "Case study : An incremental approach to re-engineering a legacy FORTRAN Computational Fluid Dynamics code in C++", Ewer J., Knight B. and Cowell D., Reproduced from "Advances in Engineering Software", Vol. 22, pp 153-168, 1995.

Case Study : An incremental approach to re-engineering a legacy FORTRAN Computational Fluid Dynamics code in C++.

J. Ewer, B. Knight, D. Cowell

The University of Greenwich, Wellington Street, Woolwich, London, SE18 6PF, U.K.

Abstract

This article describes a practical approach to the manual re-engineering of numerical software systems. The strategy has been applied to re-develop a medium sized FORTRAN-77 Computational Fluid Dynamics (CFD) code into C++. The motivation for software reverse-engineering is described, as are the special problems which influence the re-use of a legacy numerical code. The aim of this case study was to extract the implicit logical structure from the legacy code to form the basis of a C++ version using an imposed object-oriented design. An important secondary consideration was for the preservation of tried and tested numerical algorithms without excessive degradation of run-time performance. To this end an incremental re-engineering strategy was adopted that consisted of nine main stages, with extensive regression testing between each stage. The stages used in this development are described in this paper, with examples to illustrate the techniques employed and the problems encountered. This paper concludes with an appraisal of the development strategy used and a discussion of the central problems that have been addressed in this case study.

Keywords: Re-engineering, Reverse-engineering, Computational Fluid Dynamics, CFD, FORTRAN-77, C++, Numerical software.

1. Introduction

Many numerical analysis codes currently in use were originally developed as research tools by self-taught programmers without the use of any software engineering methodology. This has led to a situation where "legacy" code is still in use - often in safety critical applications - but where the code is largely "black-box" as far as support staff are concerned. This would not present a major problem if such systems had no need for change, but this is very seldom the case. Industry continually demands code enhancements and adaptations to underlying models, thus posing a serious maintenance problem.

Perfective and adaptive maintenance have long been recognised as considerable tasks in themselves. Boehm[1] comments that these tasks often exceed the scope and effort applied to the original development. This problem is exacerbated by the informal development of many legacy systems and the poor or non-existent documentation of research codes. When existing software reaches a state where a required adaptation can no longer be easily implemented, the developer is faced with a number of available options. These range from the extremes of the creation of a new system from scratch, to extensive modification and development within the existing system. Both of these extremes are fraught with difficulties in terms of time and effort, quality assurance, reliability and consistency. These difficulties are particularly acute with numerical codes which have been developed over a number of years by many different engineers, often in a variety of critical situations. Trust in the legacy code is often the determining factor in deciding whether to move to new methods.

A relatively recent approach to the problem of adapting and maintaining legacy systems is to use re-engineering techniques. These attempt to use legacy code as a template for an improved system that can then be more effectively maintained. Again, there are a number of extreme approaches to re-engineering. One extreme is for total design abstraction from the legacy code followed be re-implementation of a new system using a suitably modified design. In the other extreme, heavy use is made of automatic re-structuring, translation and documentation tools directly on the legacy system. In practice, the latter technique is often used as a precursor to the former. Such re-engineering approaches have been used with varying degrees of success on industrial and academic numerical software systems. The following references indicate some of the techniques that are available for reverse- and re-engineering:

- A recent approach has been to develop automated source code re-structuring tools to support program transformations. One such tool is Sage++[2], an object oriented toolkit for building re-structuring tools. Sage++ is still being developed but the techniques available suggest that it will be possible to automate code re-structuring, and to a lesser extent re-engineering.

- Angus and Curtis[3] describe their experiences of re-engineering a large FORTRAN-77 numerical code in C++. Some of the advantages and disadvantages of several re-engineering strategies are discussed in their paper. The authors conclude that an incremental re-engineering approach should be used because of the potential difficulties with functional consistency in new developments.

- Byrne's paper[4] presents a re-engineering case study for an Ada re-implementation of a FORTRAN simulation code. Three approaches to re-implementation are described together with their potential problems. The "formal" stages of general reverse engineering using design abstraction and re-implementation are also discussed. The author concludes that reverse engineering is a viable solution to the perfective and adaptive maintenance of legacy code.

This article presents a re-engineering project as a case study to demonstrate that a restrictive legacy numerical application can be successfully "mined" to develop a flexible and extensible system that goes well beyond the scope and capabilities of the original. Section 2 of this paper presents a background to the class of Computational Fluid Dynamics (CFD) software required as the ultimate deliverable from the project and describes some of the required functionality. The motivation for re-engineering, and the objectives for the exercise are discussed in section 3. The decision to adopt a multi-stage, incremental approach is supported by the need to preserve exact functional operation. Section 4 describes the stages in the re-engineering process, and illustrates these with examples. Finally section 5 contains an appraisal of the techniques used in this case study and demonstrates their success by means of some simulation and timing comparisons between the legacy and re-engineered codes.

2. Background to Computational Fluid Dynamics software.

The intended deliverable from this project was for a well engineered[11] Computational Fluid Dynamics (CFD) software system. CFD codes generally use numerical

techniques to simulate and predict the ultimate or instantaneous physical properties of a fluid filled region given known initial boundary conditions. The functionality of the class of CFD codes of interest is briefly described here to clarify the capabilities of such systems and to indicate some of the implications to the software implementation of such codes:
- CFD codes attempt to predict the physical behaviour of a fluid filled system at numerous discrete points throughout the region of interest. The physical properties calculated at these points represent the continuum behaviour that would be observed in a real-world system.

- The physical behaviour can include the calculation of some or all of the following properties at all of the discrete points of the computational region:

Velocity components: the speed of the fluid flow in all three co-ordinate directions,
Pressure: the force per unit area exerted within the fluid,
Enthalpy: the heat energy of the fluid in the system,
Temperature: the measurable property associated with enthalpy,
Buoyancy: the upthrust body force due to density differences,
Kinetic energy: the energy of particulate motion, and
Dissipation rate: a calculation metric used in the solution of turbulent flow using the K-Epsilon model. See reference [**6**] for details of this model.

- The particular CFD codes of interest are described as *finite volume* codes. This means that the numerical solution is based on the conservation of physical properties over small sub-regions (cells) of the domain being modelled. The signed summation of transported properties through all faces of a cell should give a value of zero, in the absence of creation or destruction of that property.

- The codes of interest can use either a 2- or 3-Dimensional *un-structured mesh* which allows a *finite volume* "cell" to have an arbitrary number of nearest neighbours, with the single limitation that there is a one-to-one cell mapping across a single cell face. An un-structured mesh capability is desirable because it allows domains with highly irregular geometries to be simulated. This means that software must store the cell adjacencies for navigation through the mesh because no regular structure is implicit in the mesh-point co-ordinates.

- Generally the solution procedure is based on an iterative scheme whereby an approximate solution, for all of the solved variables, is used to obtain a more accurate solution by repetitive sweeps of calculation. The iterative nature of the scheme means that last-iteration values must also be stored to allow the newest properties to be evaluated.

- It was a requirement that the system can solve for both steady-state (time invariant) and transient (changing with time) simulations. In transient simulations time is divided into discrete time steps. A converged solution to all physical properties is obtained at a time step before moving on to the next time step. This means that storage is also required for old time step values because these are the initial conditions for the next time step.

The characteristics of CFD software set out above are sufficient for general information. A more detailed account of relevant background material, to the type of CFD code re-engineered in this project, may be found in the following references:

- Leschziner[5], which gives a comprehensive review of CFD techniques including the solution of turbulent re-circulating flows,

- Patanker: "Numerical Heat Transfer and Fluid Flow"[6], a recommended reference text for CFD techniques involving Finite Volume methods, and

- Patanker and Spalding[7], which describes a calculation procedure for 3-Dimensional flows.

3. The motivation for re-engineering.

Prior research into producing a knowledge based CFD code (called FLOWES[8]) indicated that the development of a totally new CFD system from scratch would be a major undertaking with no guarantee of success. This was reinforced by considerations of the allowed project duration, limited CFD experience by the application developers and apprehension about the reliability of a totally new system. The alternative to the production of a well engineered CFD code, capable of extension and adaptation, was to re-use a legacy system that has been developed "in-house" at the University of Greenwich. This legacy CFD code is known as *CWNN*[9] (*C*ode *W*ith *N*o *N*ame). Commercial CFD codes were rejected because of the potential for breach of copyright and the sensitive nature of such systems.

This project was required to deliver a multi-physics CFD application framework which could be used for all aspects of the diverse research that is conducted at the University of Greenwich. This research includes such CFD techniques as combustion modelling, free surface flows, multi-phase flow, dynamic control, magneto-hydro-dynamics and ultimately elastic / visco-plastic material deformation and collision modelling. The primary development requirement was to implement a CFD framework for researching knowledge based dynamic solution control and monitoring[12]. A control and monitoring interface of this type would be very hard to implement, within the existing code architecture, without potentially disastrous extensive modifications and adaptive maintenance.

The *CWNN* system was known to work well for certain classes of simulation, and this functionality had to be maintained in any development. The core numerics, whilst generally untidy and unclear, had been validated against experiment and other CFD codes. The re-engineering would have been worthless if the "new" and "legacy" codes exhibited different behaviour.

The system analysis and software specification for this project indicated the following development requirements over and above the CFD capabilities detailed earlier:

- *Expressive and self explanatory code.* The delivery system had to be largely self documenting so that any CFD researcher could see the mathematical or physical model represented by the source code at a suitable level of abstraction. This implied that implementation details had to be kept at a suitably low level within the procedural hierarchy.

- *Data structures that prevent incorrect usage.* The internal data structures and data access

mechanisms had to minimise potential data misuse by providing a clear and consistent interface to the simulation data. Where possible the syntax had to be consistent with existing implementation techniques because the new system is targeted at existing CFD researchers.

- *Flexible and extensible architecture.* Since the system was to be used for diverse research it needed to be easily extensible. This was best supported by the provision of library routines that greatly simplify continued development of the system and help to reduce the implementation errors caused when a new routine was created from an existing one.

- *Portability.* Ideally the system had to be portable to a number of platforms, so standard language techniques had to be used. This was a valid argument for not using mixed programming language techniques even though these are available on many platforms.

- *Optimal speed performance.* CFD systems manipulate very large data structures and would be unusable if serious speed penalties were incurred. This was traditionally seen as the greatest stumbling block to successful re-engineering in this field, although recent requirements for ease of maintenance and robustness are of growing importance. In this case study speed performance was not seen or treated as the primary requirement.

- *Robust design as perceived by system user.* The ultimate system user should see the system as being robust even when spurious data is input. This meant that code developers should have access to suitable error reporting and debugging routines to use in any new code developments. These debugging facilities should include some form of trace capability so that execution could be followed without using a debugging tool.

It should be noted that some of the requirements detailed above conflict with one another. There was, necessarily, some compromise in terms of performance. This will be discussed in more detail in the appraisal of the re-engineering later in this paper.

An early decision was taken to use C++ as the target implementation language for the re-engineered system. This decision was made as a direct result of the requirements described above. In particular, C++ supported the requirements for code clarity and abstraction. The need to provide a data architecture that prevents misuse, has integral debugging capabilities and gives consistent access was also a point in favour of C++. Ada was considered briefly as an implementation language but the high cost of robust compilers and the still limited availability on many platforms precluded its choice for this research. There was also no real syntactic advantage in using Ada instead of C++ since both differ quite significantly from FORTRAN-77. FORTRAN-90 was rejected because of its even more limited availability.

CWNN posed a number of obstacles to continued development and perfective maintenance. Many of these problems were due to features inherent in the implementation language or the development techniques employed in most numerical FORTRAN codes. The identified difficulties included:

- *Poor internal and external documentation.* The internal documentation was limited to a few comments and a header for most routines that indicated argument usage, i.e. in, out and in-out usage. There was also a very brief description of what these arguments represent. The

only external documentation was a research thesis[9] that described the underlying mathematical principles involved in an earlier 2-Dimensional version of the code.

- *Inconsistencies in naming conventions and declarations.* Variables often changed name arbitrarily between routines. This meant that the data model visible to the developer was not always consistent. There were also many instances of passing only parts of multi-dimensional arrays to subroutines but using local identifier names of the same name as the complete external array.

- *Poor code re-use.* The code had been developed as a research tool where many routines had been copied and modified from other places. This had led to excessive code duplication.

- *Extensive re-declaration of variables.* The poor data structuring facilities of FORTRAN-77 forced developers to use large COMMON blocks or long parameter lists. *CWNN* used the latter as the lesser of two evils. Possible lack of correspondence between actual arguments and parameter declarations was a large potential source of error.

- *Large monolithic routines.* The developers tended to write inline code because of the overheads of re-declaring variables in new routines and because of the small potential performance benefits of inline code. This had the drawback of creating large monolithic routines which obscured the functionality of the code.

- *Unclear data architecture.* The lack of abstract data types in FORTRAN-77 meant that data relationships often had to be implemented with integer arrays that indirectly index other data arrays. This was particularly true of the unstructured mesh representation in *CWNN* which had quite complex relationships between the data arrays that hold the definition of the cells, cell-faces and the mesh points. This led to cumbersome and non-intuitive data access techniques. Since FORTRAN-77 has no aggregation types, all data items had to be passed around explicitly. If a routine was found to need access to a certain variable, this had to be passed though all of the intervening routines from the top level of the code. FORTRAN numerical programmers generally avoid using functional data access because of the performance overheads.

- *Lack of algorithmic clarity.* The use of standard FORTRAN-77 with 6 character identifiers meant that algorithmic functionality was often very obscure. This was further compounded by the lack of data- and subroutine-dictionary and the poor documentation.

- *Mixed levels of code abstraction.* The level of algorithmic abstraction, in *CWNN*, was not always consistent. There were several types of abstraction inconsistency. Often simple assignments surround calls to complex routines thus hiding the algorithmic meaning in trivial implementation details. Furthermore, utility routines were often called after calls to complex routines but such calls were "procedurally bound" to the major routine. The utility routines were inappropriately located in the subroutine hierarchy. These particular problems were identified by examining the calling structure, the source code immediately adjacent to major routine calls and the placement of calls to the identified utility routines. The calling structure chart of the software was generated automatically by the FORTRAN re-structuring tool *SPAG*[10]. Multiple calls to a single routine identified a potential utility routine whilst a single call to a routine indicated a potentially highly abstract routine. In the case of numerical code a highly abstracted routine is one which uses a number of utility routines to

perform some function or uses many intermediate stages for calculating some property. In the example code fragment (Figure 1) the routines HCSOLV and LINRLX demonstrate the different levels of abstraction mixed together. HCSOLV actually takes flow properties, physical properties and previous conditions to calculate the new values of enthalpy for every cell of the domain. The HCSOLV routine was only called once in the whole code. Conversely LINRLX merely added a fraction of the difference between an old and new value of some property and was called from many places within the code. It was concluded that HCSOLV was a highly abstracted routine whereas the less abstracted LINRLX was a utility routine that should be moved down into HCSOLV. The simple assignment statements were inappropriately located and should be moved into adjacent routines.

Legacy code fragment	*is problematic because*
`IF ( IHEAT ) THEN`	

`  RELAXA = SRELAX(5)`
`  RMETHD = VRMETH(5)`
`  MITERS = MAXITR(5)`
`  FALSET = VFALST(5)`
	Literal values used and simple assignments prior to calling a complex numerical calculation routine.

`  CALL HCSOLV(3, RELAXA, ... )`	Highly abstracted routine call.
`  CALL SYSRES( ... )` `  SOLERR(5) = RESIDU`	Less abstracted utility routine call.

`  RELAXA = VRELAX(5)`
`  CALL LINRLX( ... )`
`  VARERR(5) = RESIDU`
	Utility routine call.

`  RELAXA = VRELAX(8)` `  RMETHD = VRMETH(8)`	More low level simple assignments.
`  CALL CSOLVT( ... )`	Call to highly abstract routine.

`  IF ( ERRINF .EQ. 0 ) STOP`
`  VARERR(8) = RESIDU`
`ENDIF`	

Figure 1 : Mixed levels of abstraction.

- *Inflexible memory usage.* FORTRAN-77 array sizes must be statically declared at compile time. This limits the flexibility of the code for running problems of different sizes. This often forces a re-compilation of the system for the specific problem to be run. This was particularly true of the un-structured mesh code where a few cells with a high number of faces greatly increased the storage requirements of all cells.

FORTRAN code	*is problematic because:*
`SOLTYP(1) = 5`	The SOLTYP array holds the solver types but the index 1 and solver type 5 could relate to anything. The intended meanings must be found elsewhere in the code.
`K = K + 0.37777 * (...)`	Explicit values used in numerical algorithms give no indication of the purpose of the values used.

Figure 2 : Use of literal constants.

- *Excessive use of literal constants.* Much of the meaning of *CWNN* was hidden by the use of literal constants. Future code developers would have little understanding of what these

seemingly arbitrary valued numbers represent, particularly in light of the meagre code documentation. Figure 2 shows the two types of literal values used in *CWNN*. These are array indexing by integer values and physical constants entered explicitly without definition.

A re-engineering strategy was sought that would remove the problems inherent in the FORTRAN implementation of *CWNN* whilst providing an extensible framework for continued research into CFD techniques. The approach described here was planned and refined during the re-engineering of the legacy code. Example test cases were used to check code consistency throughout each stage of the development.

As an investigation an attempt was made to translate the FORTRAN code into C or C++ using two proprietary tools. The results of the translations were very disappointing. The translated code was noted to be less clear than the original with little or no improvement in structure. In some instances there was considerable degeneration in code clarity with awkward and non-portable handling or certain library routines. The translators were clearly designed to allow compilation in C rather than to support continued development. This was the major reason for the manual translation strategy adopted in this case study.

4. The multi-stage re-engineering process

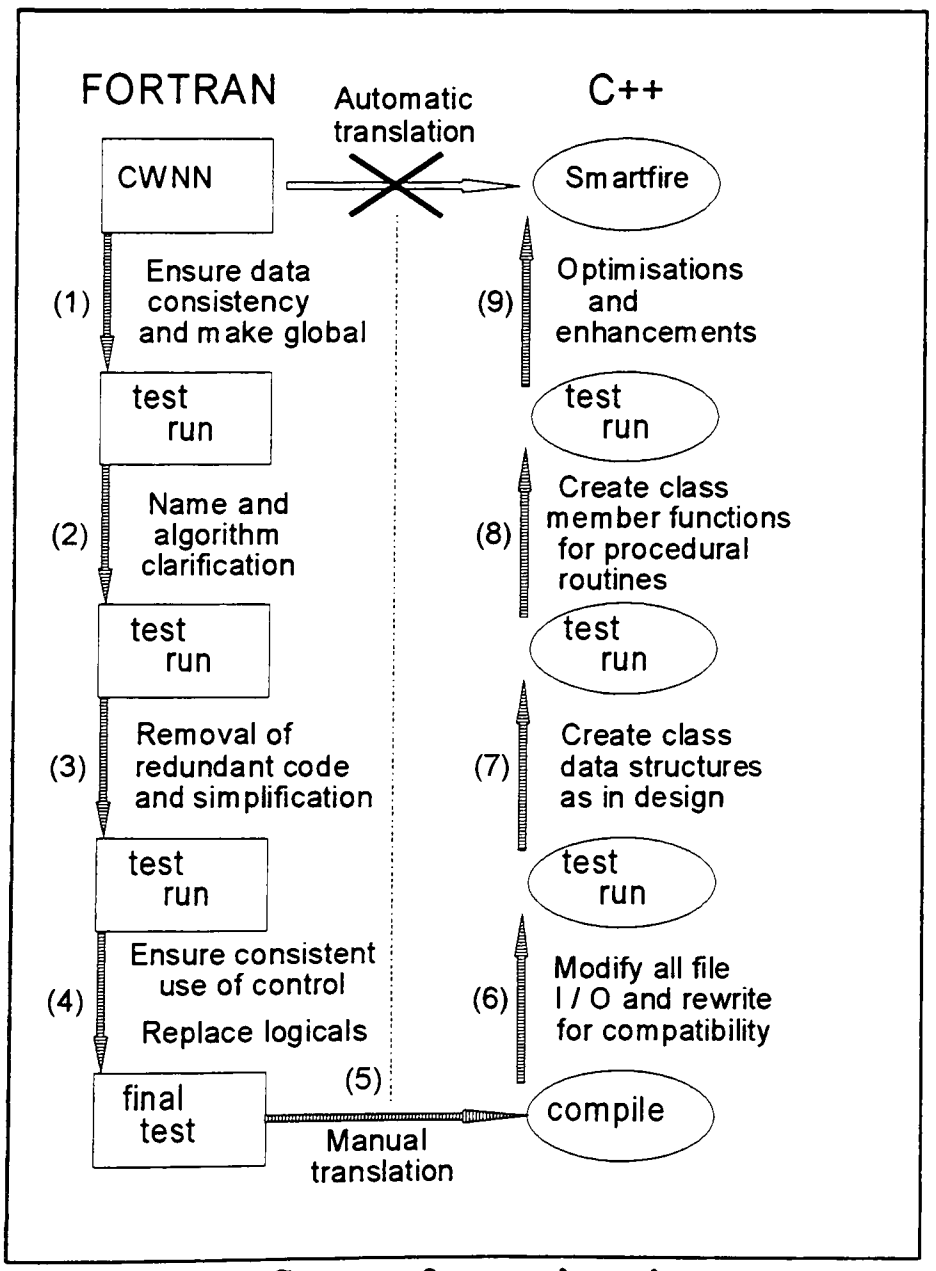


Figure 3 : Stages of re-engineering.

The adopted re-engineering strategy used a nine stage incremental process to restructure the legacy code in FORTRAN-77, translate to C++, enforce modern software engineering design principles and prepare for perfective and adaptive maintenance. The flow diagram (Figure 3) indicates the main stages in the re-engineering of *CWNN*. The central vertical line indicates the boundary between FORTRAN-77 and C++. Even after massaging the FORTRAN-77 source, the possibility of automatic translation into C++ was discounted as being potentially unreliable. It should be noted that the stage (4) FORTRAN code was a complete algorithmic design specification, in source code form, of the final delivery system.

Comprehensive regression testing (using a numerical file comparison utility called Numdiff[13]) after each incremental stage of the rewriting process ensured compatibility between the delivery system and the legacy code.

Stage (1): Ensure data consistency and make all data global.

All of the disparate source files were combined into a single file. This was necessary because the editor used throughout this research has a limit on the allowed number of open files. This was not a great disadvantage because the editor used has very fast search facilities and no effective limitation to maximum file size.

SPAG[10] was used to "pretty-print" the source code. The *SPAG* tool set also has a global code check utility that was used to generate much needed information about subroutine calling structure and variable usage. *SPAG* was configured to set the case of identifiers to indicate variable scope and usage. COMMON variables and PARAMETERs were completely capitalised whilst local variables used only lower case. Subroutine arguments had an initial capital letter followed by lower case characters. This helped to locate the appropriate declarations and showed the dependencies of any variable.

In order to restructure the software into an object oriented form, the data was grouped into classes. This could not be done if data items changed names in argument lists or were passed around as incomplete array segments. It was therefore necessary to match calling and called routine arguments and rename local variables to match external data items. *SPAG* was used extensively to document and navigate within the CFD code.

Legacy FORTRAN code

```
CALL BUOYAN( RMETHD, ELEMAT, ELEVOL,
             TEMPER, .. )


SUBROUTINE BUOYAN( RMETHD, ELEMAT,
                   VOLUME, T, .. )
INTEGER ELEMAT(TOTELE)
REAL    VOLUME(TOTELE), T(TOTELE)
B = T(I) * ...
```

becomes

```
INCLUDE 'DATABASE.INC'
CALL BUOYAN( RMETHD, .. )

SUBROUTINE BUOYAN( RMETHD, .. )
INCLUDE 'DATABASE.INC'
B = TEMPER(I) * ...
```

with DATABASE.INC defined as

```
INTEGER ELEMAT(TOTELE)
REAL ELEVOL(TOTELE), TEMPER(TOTELE)
COMMON /CELL_D/ ELEMAT, ELEVOL, TEMPER
```

Figure 4 : Passing data by include file and COMMON.

Non-standard FORTRAN include files were used to pass data between routines. These include files contained the COMMON data declarations that ensured that only one declaration existed for each variable. Any identified utility routines kept the parameter list arguments for the passing of data. In some instances COMMON was used inappropriately for passing data to utility routines. This problem was quite easy to identify because of the necessary introduction of many simple assignment statements (putting data into the COMMON variables) just before the utility routine call. Figure 4 demonstrates the passing of data in COMMON blocks.

Data items were grouped into named COMMON blocks with related items, as they were identified. This identification was facilitated mostly by the dimensions of the arrays and the limited amount of subroutine header information available. Tentative groupings were made based on the declared array sizes and these were revised as actual array variable usage was identified within the source code. For example arrays with dimensions of (1..NOCELL) indicated cell properties of some sort whilst those dimensioned as (1..NOFACE) were face properties. Many of the single variables were identified as being suitable for COMMON by simple inspection of their usage. Switch control variables tended to be more appropriately passed as arguments to routines. It was preferable to err on the side of caution because of the slight potential for naming conflicts between new COMMON variables and dummy arguments.

Legacy FORTRAN — *becomes*

```
CALL HBOUND( H(ICELL), .. )          CALL HBOUND( ICELL, .. )


SUBROUTINE HBOUND( HVAL, .. )        SUBROUTINE HBOUND( ICELL, .. )
REAL HVAL                            INCLUDE 'DATABASE.INC'
HVAL = HVAL * ...                    INTEGER ICELL
                                     H(ICELL) = H(ICELL) * ...
```

Figure 5 : Revised argument passing for COMMON data.

Dummy argument names (within subroutines) were replaced with direct access to the newly defined COMMON array variables. Where subroutines were receiving arguments which were single array elements it was necessary to ensure that the appropriate array index was available within the subroutine. The code fragment (Figure 5) indicates how array index values were passed instead of the array elements.

Stage (2): Name and algorithm clarification.

FORTRAN-77 standard 6 character identifiers and routine names were replaced with longer, lower case, names that convey functional meaning and usage. *SPAG* was used to automatically rename the identifiers as it prevents and reports renaming conflicts. Some of the initial name changes are detailed in Figure 6.

Legacy FORTRAN	->	*New naming convention*
MCSOLV	->	solve_momentum
CALGEN	->	calc_generation_rate
RDINFF	->	read_inform_file
H	->	enthalpy
TEMPER	->	temperature
U	->	u_velocity
KINETC	->	kinetic_energy
DISSIP	->	dissipation_rate

Figure 6 : Name changes for code clarification.

Inline code sections were moved into new subroutines to highlight their algorithmic meaning at an appropriate level of abstraction. Passing data by include file and COMMON facilitated this process since extensive re-declarations were no longer necessary. There are two ways to identify inline code. The first was recognition of those instances of code that keep appearing relatively unchanged throughout. In large systems it may not be possible to identify many such fragments but in this case study it was known that some of the code was implemented by copying and modifying other code fragments. An example of repetitive inline code in *CWNN* was for the calculation of the cell upwind density. This code consists of 32 lines of source code duplicated in 10 different routines. The second sort of inline code was the use of large code fragments in control constructs such as *IF (..) THEN...ENDIF* blocks or *DO...CONTINUE* loops. FORTRAN developers tend to inline code accidentally as blocks grow bigger with research, or deliberately in low level loops to avoid the overhead of a repetitive subroutine call.

Legacy FORTRAN code *becomes*

```
SOLTYP(1) = 5
```

```
INCLUDE 'PARAMS.INC'
solver_type( PRESSURE ) = SOR
```

with PARAMS.INC defined as

```
INTEGER PRESSURE, SOR
PARAMETER( PRESSURE = 1, SOR = 5 )
```

Figure 7 : Introduction of PARAMETER constants.

Literal numbers used to index arrays or used in calculations were globally defined as more meaningful PARAMETER statements in an include file. This file was then included in all routines as indicated in the example code fragment (Figure 7).

Stage (3): Removal of redundant code and simplification.

Code paths and variables that were not required for the current project were removed from the system. It was noted that solidification modelling was not necessary, so the corresponding code was completely removed. The solidification code was simple to remove because it was all switched via logical control variables. The extra variable solver also

presented no difficulty to removal because it was (like most of the other solvers) simply a copy of a former routine with the data variables changed.

Legacy code fragment

```
MITERS = MAXITR(4)
CALL SORSCH( ... )
SERROR(4) = RESIDU

RELAXA = VRELAX(4)
CALL LINRLX( ... )
VARERR(4) = RESIDU
```

Equivalent code abstracted

```
INTEGER VAR_W_VELOCITY
PARAMETER( VAR_W_VELOCITY = 4 )

CALL SORSCH( VAR_W_VELOCITY, ... )

CALL LINRLX( VAR_W_VELOCITY, ... )
```

N.B. The simple assignment statements have been moved down into the called routines.

Figure 8 : Re-locating simple assignment statements.

There were many instances where code fragments could be simplified by moving simple executable statements (generally assignments) into nearby called routines. This helped to keep the code at the same level of algorithmic complexity and avoided unnecessary clutter as shown in the example code (Figure 8).

CWNN had many "hooks" for future use. For example, dummy routine calls and logical variables were provided to allow future development of mesh-refinement. These were noted for location and function and then removed to simplify the re-engineering process.

Stage (4): Ensure consistent use of control and replace all logical variables.

Labelled lines were made to use CONTINUE rather than have executable statements. This helped with the translation to C++ and made it easier to find other loop constructs.

Instances of single line "IF (<expr>) <statement>" were replaced with the equivalent form using "IF (<expr>) THEN <statement> ENDIF" so that subsequent translation to C++ would be facilitated. Instances of "IF (<expr>) GOTO <label>" were left unchanged because these were often part of "do...while" constructs.

Loop constructs which used the standard "DO <label> <block> <label> CONTINUE" were changed to non-standard "DO <block> ENDDO" loops which avoided excessive use of continue labels. The use of "DO...ENDDO" loops also allowed easier recognition of the other uses of "<label> CONTINUE" as in FORTRAN simulated "do...while" loops. Any clearly identifiable "do...while" loops were implemented with the non-standard FORTRAN WHILE constructs instead of the usual "IF (<expr>) GOTO <start_label>" as used in the legacy code. *SPAG* was useful in this respect because it has some automatic re-structuring capabilities supporting non-standard, but widely used, control constructs.

Since C++ does not support a built-in LOGICAL type it was decided that an equivalent, robust replacement should be implemented in the FORTRAN code. The direct

translation to a C++ enumerated type was considered but, because there was no conformal mapping for assignment using the NOT value of a logical, the idea was discarded. LOGICAL variables and comparisons were replaced with integers and integer comparisons respectively. The complexity of replacing the LOGICAL values was significantly reduced by working within the FORTRAN version of the code. This also prevented errors in logic that could occur when too many translation steps had to be performed simultaneously. The replacement of a LOGICAL sometimes required introducing "IF (<expr>)" constructs to assign appropriate "boolean" values to integer variables. The integer parameters "False" and "True" (representing 0 and 1 respectively) were used throughout the code to match the ultimate C++ representation.

Stage (5): *Translate FORTRAN to procedural C++.*

When the above stages had been completed and the FORTRAN code was still running consistently it was necessary to translate the FORTRAN to procedural C++. This was because no appreciable advantage could be gained by further FORTRAN code changes. The serious limitations of the available FORTRAN-to-C translators led to the decision to translate the CFD code to procedural C++ manually. The natural course of action would be to use parsing or compiler writing tools such as *lex*[14] and *yacc*[15] but because of the high learning overheads and non-interactive nature of these utilities, an alternate approach was sought. The tool actually chosen was a powerful programmer's editor with regular expression search and replace facilities, macro record and playback, and multi-file editing capabilities. It should be pointed out that a simple text editor would not be sufficient because of the large syntactic variation that may be encountered in the source code.

Using the editor facilities, the translation to procedural C++ took 10 days and involved writing a set of individual macros to replace specific code constructs. Again *SPAG* was used prior to this task so that a consistent style and control syntax would persist throughout the code. This was necessary to enable the searches within the macros to work correctly. Figure 9 indicates some of the macro text replacements that were used during the manual translation from FORTRAN-77 to procedural C++. The use of regular expression searches and macro replacements does require that care was taken to perform the replacements in order of most complex to least complex to prevent incorrect matching with parts of other expressions. The main problems are with *DO*, *END* and *IF* which can be part of other keywords like *END IF* or *END DO*. There are also potential problems with accidentally matching search expressions with literal strings or parts of variable names. Using case sensitive searches, after *SPAG* had been used to consistently set the case of keywords and identifiers, minimised the potential for problems. Clearly these problems would not be present using compiler writing tools (e.g. *yacc* or *lex*) because all identifiers are recognised as whole tokens. The program editor was useful in one respect because the regular expression text replacements are interactively controlled.

Legacy FORTRAN	->	*Macro replacement*
ELSEIF (..) THEN	->	} else if (..){
IF (..) THEN	->	if (..){
ELSE	->	} else {
ENDIF	->	}
CALL	->	/* CALL REMOVED */
DO I = a, b, c	->	for (I=a; I>=min(a,b) && I<=max(a,b); I+=c){
DO I = a, b	->	for (I=a; I<=b; I++){
ENDDO	->	}
SUBROUTINE .. (...)	->	void .. (...){
END	->	}
RETURN	->	return;
PRINT*, ...	->	cout << ... << endl;
nnn CONTINUE	->	Label_nnn:
GOTO nnn	->	goto Label_nnn;

Figure 9 : Macro replacements.

String variables (i.e. FORTRAN-77 CHARACTER*(n)) were dealt with on an individual basis since the numerical code only had a few routines which manipulated strings. Literal strings were easily replaced by the [''] delimited versions of C++.

One of the major problems encountered during the translation was the difference in array indexing syntax. FORTRAN style array indexing is vastly different to C++ style array indexing. It was decided to effect these changes manually (using searches and macros) on a variable by variable basis. Macros were used to change the ( ) indices to [ ] indices, but these could not be used globally because of the complexities of multi-dimensional arrays, partial array argument passing and arrays that index other arrays. It was necessary to increase the declared array dimension sizes by one, and waste the 0^{th} element, because of the declaration syntax used in C++. Fortunately most of the FORTRAN arrays had (1:n) indices, but (-m:n) or (m:n) indices can be represented by simply adding or subtracting a suitable constant at the declaration and each reference. All of the arrays were initially translated to statically declared C++ arrays. The potential problems of passing segments of multi-dimensional arrays were largely avoided because of the earlier consistency modifications made to the legacy code in stage (1).

Although this *ad hoc* approach worked well in this instance, the authors came to the conclusion that for re-engineering much larger programs, it would be advisable to make heavy use of automated compiler writing tools at this stage or, if available, translation tools.

CWNN has the traditional batch-mode "INPUT -> PROCESS -> OUTPUT" execution path so it was not necessary for the C++ output to work in an identical fashion to the FORTRAN output for initial compile and run testing. Simple use was made of *cout <<* and *printf()* whilst leaving the original commented-out FORTRAN for later reference.

Single item input presented no problem using *fscanf()* and *cin >>* as appropriate. List directed and formatted FORTRAN input were more problematic. The approach adopted was to replace a list directed FORTRAN input with a collection of single item or looped-over *fscanf()* calls. It was necessary to remove end-of-line comments from input files until a permanent "in-code" solution could be implemented.

After several trial compilations and minor fixes a clean compilation was obtained. A small data set and a C++ debugger were used to check that input files were accurately read.

Stage (6): Modify all file I/O and rewrite for compatibility.

The first task in C++ was to ensure that all of the file input and output was being performed correctly. This was very easy to check by immediately dumping any item read out to a log file and comparing this with the original data file. It was often necessary to use the *ifstream getline()* function to clear to the end of input lines because of the different handling of I/O by FORTRAN and C++.

One of the input problem-specification files made use of a script like language that presented some difficulties because of potentially multiple command arguments. These problems were overcome by implementing a line parsing routine and corresponding token extraction utilities to interpret the lines of input. This was the only area where new code had to be developed due to the differences between the FORTRAN and C++ languages.

Generally most numerical reading was as simple in C++ as it was in FORTRAN however care must be taken with formatted input where numbers can be written with no separators between them. If, as with the FORTRAN implementation, the format was known then this difficulty can be overcome by reading the required field width into a character buffer and extracting the values from the buffer. One other difficulty was with PC C++ compilers which tend to write float values in double precision exponential format. A writing routine was developed for one file which required single precision format for use in another package.

Stage (7): Implement class objects to replace array structures.

The original array structures of the FORTRAN code were very unsatisfactory because there was no explicit grouping and no obvious relationship between many of the variables, apart from the nature of the indexing. The groupings used to make COMMON variables in the legacy code provided a means of collecting data items into structures (C++ classes). This allowed the creation of physically meaningful entities with known attributes. The diagram (Figure 10) shows some of the legacy code FORTRAN-77 arrays.

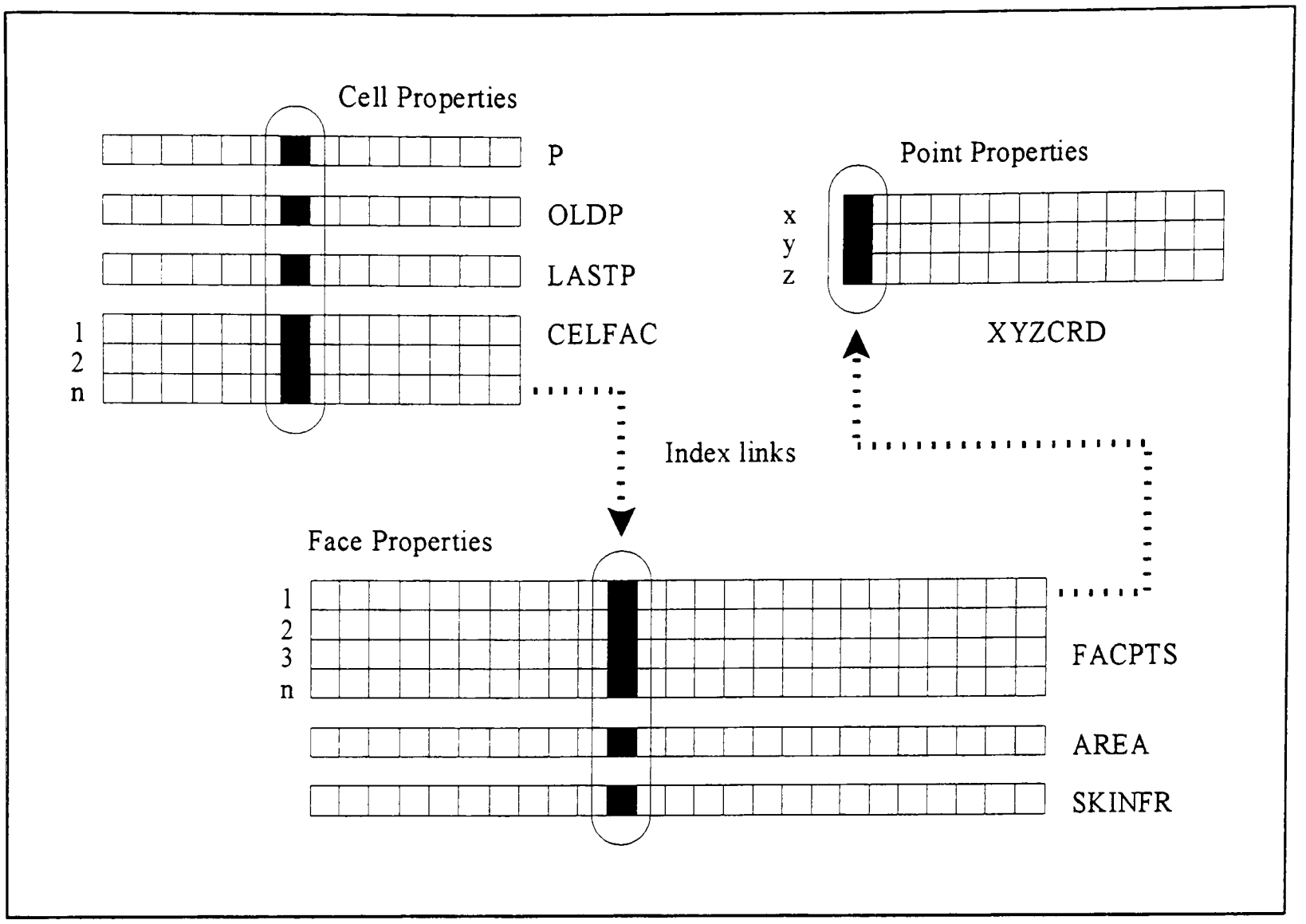


Figure 10 : Legacy code arrays.

This corresponds to the diagram (Figure 11) of the equivalent C++ classes. Some of the arrays in *CWNN* (See Figure 10) contain actual data values (e.g. P, OLDP, AREA and XYZCRD) whilst others contain index values (e.g. CELFAC and FACPTS) that are used to reference data items in other arrays. It should be noted at this point that the links between the identified class objects could have been implemented using pointers and arrays of pointers instead of integer indices and arrays of indices. The major problem with using pointers extensively is that this would necessarily introduce pointer de-referencing to access values. This would almost certainly be unfamiliar to many numerical CFD developers. It was decided that the object links be implemented in a form not too dissimilar from the legacy code. Whilst this was often less elegant than other techniques it did have the benefits of consistency and ease of implementation.

There was a potential problem for the storage of cell properties because of the need to maintain up to four versions of some variables. For example a transient flow simulation needs old time-step, last sweep, previous iteration and newest values of pressure. Also variable usage is determined by simulation type. Using explicit cell class attributes for cell properties (e.g. cell.pressure, cell.old_pressure, cell.last_pressure, and cell.previous_pressure) would always use storage regardless of the type of simulation. It was decided that simple "data" arrays of properties should be created and then indexed by parameter type identifiers (See Figure 11). The "slots", in the data array, can then be assigned as required by the simulation. This was particularly important, for example, in a heat-transfer simulation where the overhead of storing the other flow variables is highly undesirable. This approach also allowed for functional data access with expressive selection arguments, and ease of data monitoring.

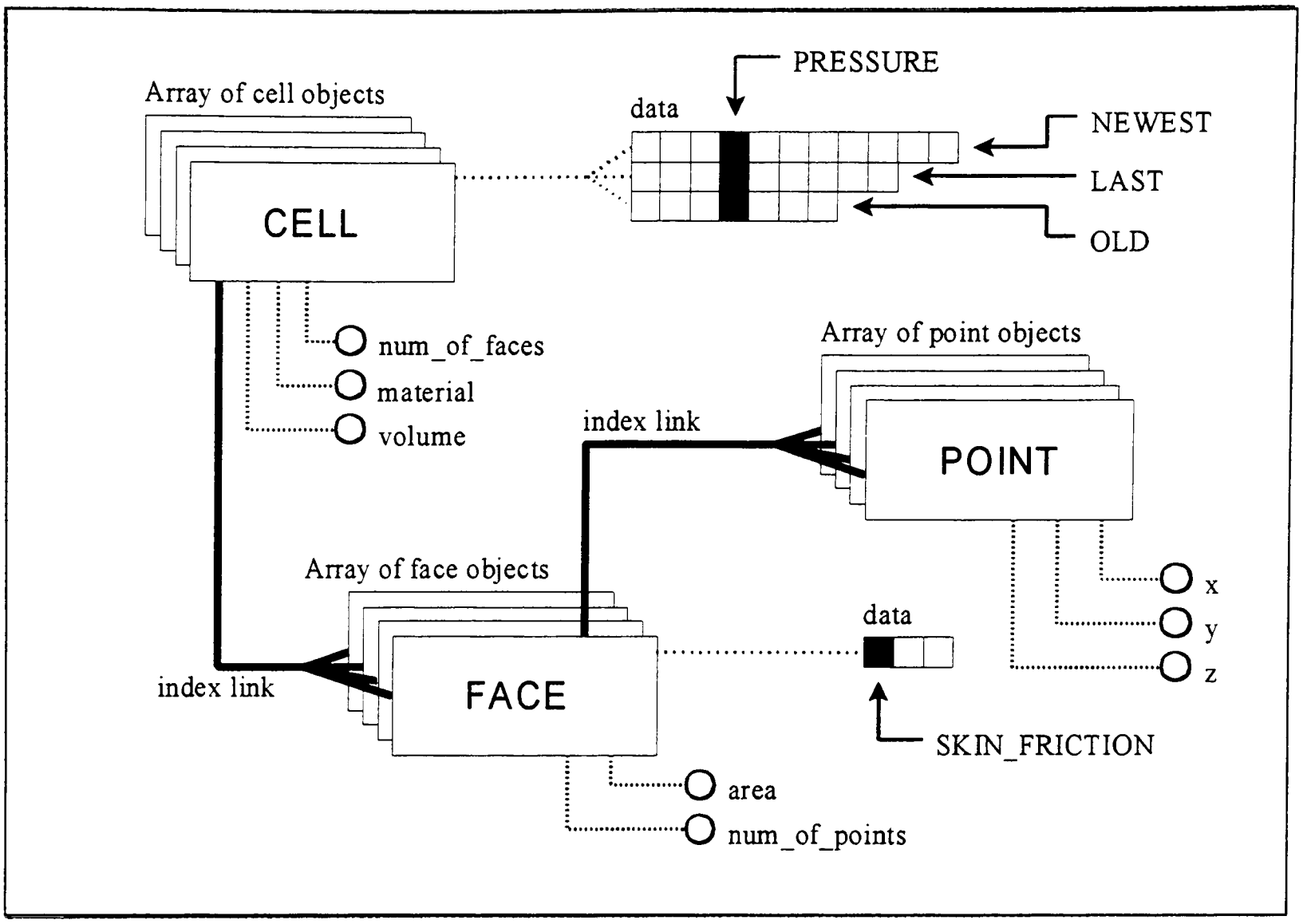


Figure 11 : Re-engineered Object Oriented data structures.

Stage (8): Create Class member functions for procedural routines.

Legacy routine for all cells

```
void calculate_all_volumes( void ){
   for ( i = 1; i <= max_cells; i++ ){
//    calculate volume for current cell
      ..................
      ..................
      cell[i].volume = ...
   }
}
```

New cell method

```
void Cell_Class::calc_volume( void ){
// calculate volume for current cell
   ..................
   ..................
   *this.volume = ...
}
```

and the original routine becomes

```
void calculate_all_volumes( void ){
   for ( i = 1; i <= max_cells; i++ ){
      cell[i].calc_volume()
   }
}
```

Figure 12 : Identification of class methods.

Many of the procedural calculation routines contained code similar to the fragment in Figure 12 which calculates cell volumes. The inline code (looping over all cells) often forms a natural class member function for the individual loop objects. The loop code can be abstracted into class utility methods as indicated in the figure. The advantage of this approach was that utility routines are created that provide much greater flexibility than was

afforded by the original software architecture. Previously the software could only calculate for all objects at once whereas it may be desirable (as in the case of mesh adaption and mesh refinement) to limit the calculation to selected objects.

The identification of class methods from FORTRAN legacy code was quite straightforward because numerical codes tend to be optimised for speed rather than for memory usage. This means that developers introduce variables to hold values like face areas, volumes and normals rather than repetitively re-calculating them. This allows methods to be identified in the initialisation stages of the software where these values are first set. The variable storage was also kept for optimal performance. The initial development may not require the flexibility afforded by this approach but subsequent research is likely to benefit.

Another area where methods were identified was in routines that were "copy-modified" versions of others. Some code fragments had the same algorithmic structure but used different variables. These were relatively simple to parameterise and abstract to function calls.

Stage (9): Optimisation and enhancements.

The initial class oriented C++ version used statically declared arrays of objects. Dynamically declared arrays were considered to be desirable and were easy to implement. This gives much greater code flexibility without the need to re-compile for larger simulations. It was also possible to implement arrays of pointers to objects so that individual objects could be created and destroyed as required. This could be very important for cell refinement where more cells are created during the running of the program.

The debugging of CFD codes has traditionally been a problem. The initial re-engineered system used direct data access to cell objects but such access cannot be easily controlled or monitored. The approach finally adopted was to use access functions that return references rather than data values. Such access functions can then be used on either side of assignment statements to set or get values. Figure 13 shows the implementation of the access function in C++. Since data access was implemented as a function, debug code could be planted to monitor the usage of a chosen cell or variable. This is demonstrated by the debug code to watch for negative temperatures. An optimised version of this access function uses an inline definition so that it has no greater performance overhead than a direct array access.

```
Enhanced debug facilities for code development via class methods

float & Cell_Class::access( int mode, int var ){

#ifdef __DEBUG_CODE__
   if ( ( mode == NEWEST ) && ( var == TEMPERATURE ) ) {
      if ( data[mode][var] < 0.0 ){
// Error negative temperature detected in cell data access
      }
   }
#endif

   return data[ mode ][ var ];
}
```

Figure 13 : New data access function.

The solvers available in *CWNN* were all based on whole matrix solving techniques. It was suggested, by a CFD researcher, that a true cell-by-cell solver should be developed. This has been implemented from some of the software components produced during this project.

The implementation of a vector class for normal and displacement vectors has greatly simplified the source code. The original FORTRAN code had to loop over all three dimensions for calculations whereas now, simple vector algebra can be performed. Operator overloading has been used to give the vector algebra a more natural syntax, as found in most reference texts. Figure 14 shows the equivalent loops for calculating the cell centres from the legacy code and the re-engineered code, which uses vector methods. Such instances were relatively easy to find because the loop dimensions go from 1 to 3 (or 1 to *DIMENSIONS*). It was possible to abstract one and sometimes two levels of looping because of the new operators and functions provided for vector algebra in the vector class. These functions included *dot*- and *cross*- product utilities.

Original legacy code

```
   DO 1 I = 1, NOCELL
      DO 2 J = 1, 3
         CENTRE( I, J ) = 0.0
2     CONTINUE
      DO 3 J = 1, 3
         DO 4 K = 1, NPTCTY( CELTYP( I ) )
            CENTRE( I, J ) = CENTRE( I, J )
     +      + XYZCRD( CELPTS( I, K ), J )
4        CONTINUE
         CENTRE( I, J ) = CENTRE( I, J )
     +      / REAL( NPTCTY( CELTYP(I) ) )
3     CONTINUE
1  CONTINUE
```

New C++ code using a vector class

```
int i, j;
for ( i=1; i<=num_of_cells; i++ ){
   cell[i].mid.set( 0.0, 0.0, 0.0 );
   for ( j=1; j<=cell[i].num_of_pts; j++ ){
      cell[i].mid = cell[i].mid +
         point[ cell[i].pt_num[j] ];
   }
   cell[i].mid = cell[i].mid /
      (float) cell[i].num_of_pts;
}
```

N.B. This code does not loop for the three directions because the overloaded vector operators "+" and "/" hide these details.

Figure 14 : Example of using a class for vector algebra.

Any further optimisation and enhancement features, that were identified early in the re-engineering stages, were researched and implemented in this stage. An example of optimisation was the relocation of loop invariant calculations outside of low level loop constructs.

5. An appraisal of the re-engineering technique used

The following sections detail code and project statistics, give comparative simulation results between the legacy and new systems, and provide an appraisal of the re-engineering techniques employed in this case-study.

Statistics associated with re-engineering CWNN:

The following code statistics provide a crude comparison between the legacy and new systems. This information should be regarded as being of academic interest only and not necessarily typical of any other or similar re-engineering projects.

- The legacy system consisted of 107 source files that contained 158 routines. There were 22,450 Lines-Of-Code (LOC) excluding comments.

- The re-engineered system has 4 source and 13 header files and has 395 routines including class member functions. There are 11,250 LOC in source files and 1,400 LOC in header files.

The project statistics (Table 1) indicate approximate durations of the individual stages used during the re-engineering. The final stage (stage 9) has not been included because perfective and adaptive maintenance is ongoing. The project durations are measured in Person-Weeks (PW). A Person-Week is defined as 5 work days for one system developer.

Stage	Description	Duration (PW)
-	Background research into un-structured mesh CFD.	6
-	Project planning and learning to use tools.	3
(1)	Ensure data consistency and make data global.	3
(2)	Name and algorithm clarification.	2
(3)	Removal of code and simplification.	3
(4)	Ensure consistent use of control and replace logicals.	2
(5)	Translate from FORTRAN to C++.	2
(6)	File I/O modifications.	3
(7)	Implement data classes to replace arrays.	4
(8)	Create class member functions.	4

Table 1 : Project statistics.

Comparative timings and run-time behaviour between the legacy and new systems:

The first simulation considered was for a turbulent straight duct flow of 2000 cells. The solved variables, relaxation parameters, solver types and convergence criteria are identical in all of the runs conducted.

Hardware, compiler and system	Iterations
486DX-50 PC, Salford FORTRAN v2.74, Legacy	116
486DX-50 PC, Zortech C++ v3.1, New	116
Sun classic, f77 v3.0, Legacy	116
Sun classic, CC v4.0, New	116

Table 2 : The number of iterations taken for convergence to a tolerance of $1.0x10^{-5}$ for both the legacy and new systems.

In order to obtain unbiased and representative results, the timings were conducted on several hardware platforms with more than ample RAM available. The systems used were an MS-DOS based 486DX IBM compatible PC (running at 50MHz) and a Sun classic running the Solaris 2.3 unix operating system.

System	Compiler	Time
Legacy	Salford FORTRAN v2.74	16m 07s
New	Zortech C++ v3.1	18m 16s

Table 3 : The 486DX PC comparative timings for 100 iterations.

System	Compiler	Compile options	Time
Legacy	f77 v3.0	-O3 -cg89	4m 10s
New	CC v4.0	-O3 -cg89	6m 17s

Table 4 : The Sun classic comparative timings for 100 iterations.

It should be noted that the PC compilations were configured to be as similar as possible within the limitations of available compiler options.

Comparison of results for the legacy and new systems:

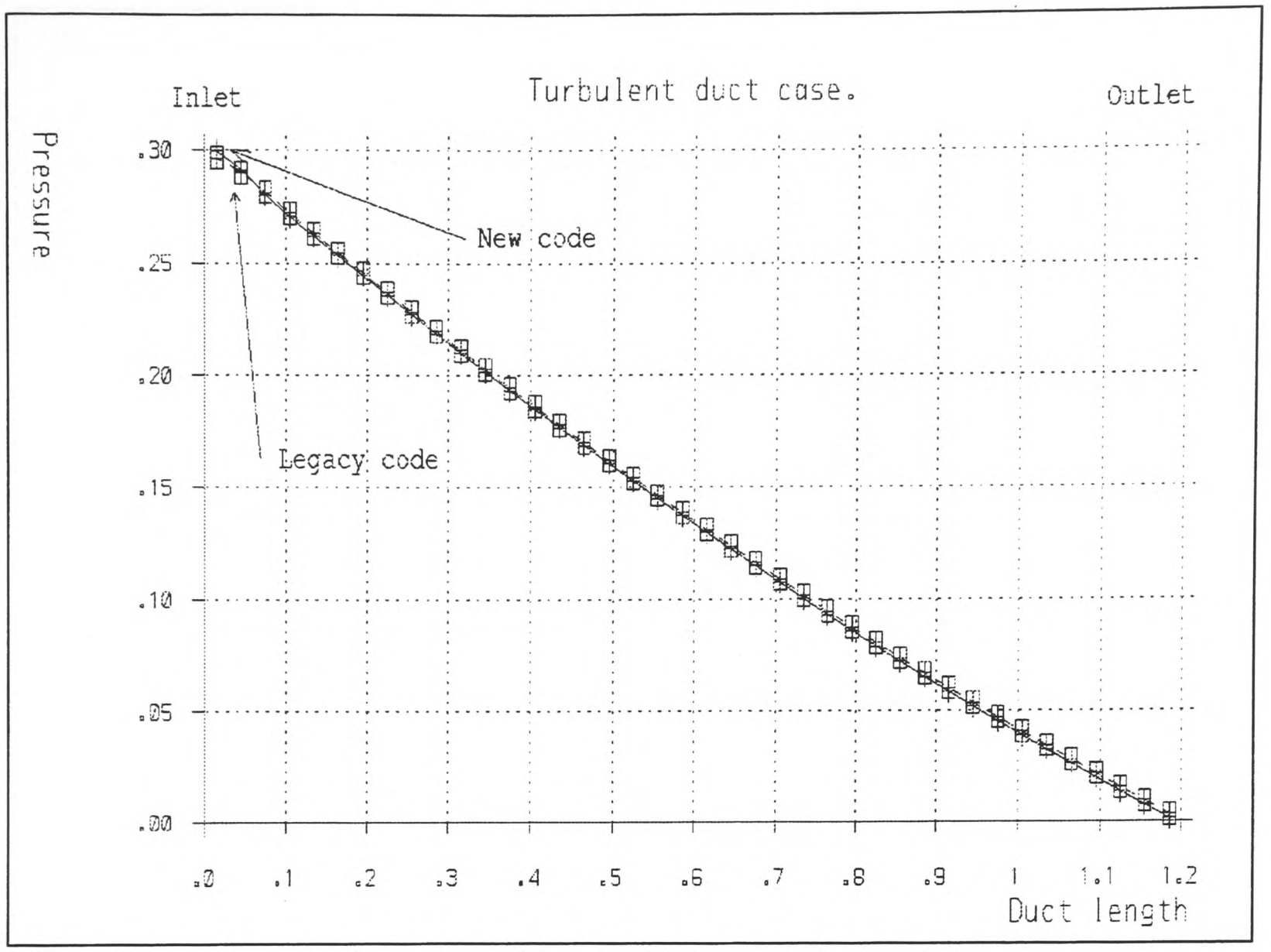


Figure 15 : Comparative values of Pressure for the Turbulent Duct.

Figure 15 shows a graph data plot for the solved pressure values from the Turbulent duct simulation described above. The legacy and new system results are overlaid on the same plot. The results are almost identical with a minor discrepancy at the inlet. This difference was due to a recent modification to the inlet boundary condition handling that was recommended by one of the legacy code developers. This "fix" was not applied to the legacy code version used in the comparisons. The other solved properties are all similarly consistent.

A more problematic simulation was also used for data consistency comparisons. This was necessary to exercise the Enthalpy, Temperature, Buoyancy and flow interactions within the system. These properties and couplings were not required in the turbulent duct flow described earlier. The following simulation was for a fluid filled box cavity that has one vertical hot wall and one vertical cold wall. Fluid temperature changes cause buoyancy (expansion) forces that drive a fully re-circulating flow. This was indicated by the vector flow plot (Figure 16) taken from a visualisation of the results. The horizontal slice indicated in the diagram was used to compare the results of the velocities calculated in the legacy and re-engineered systems in Figure 17.

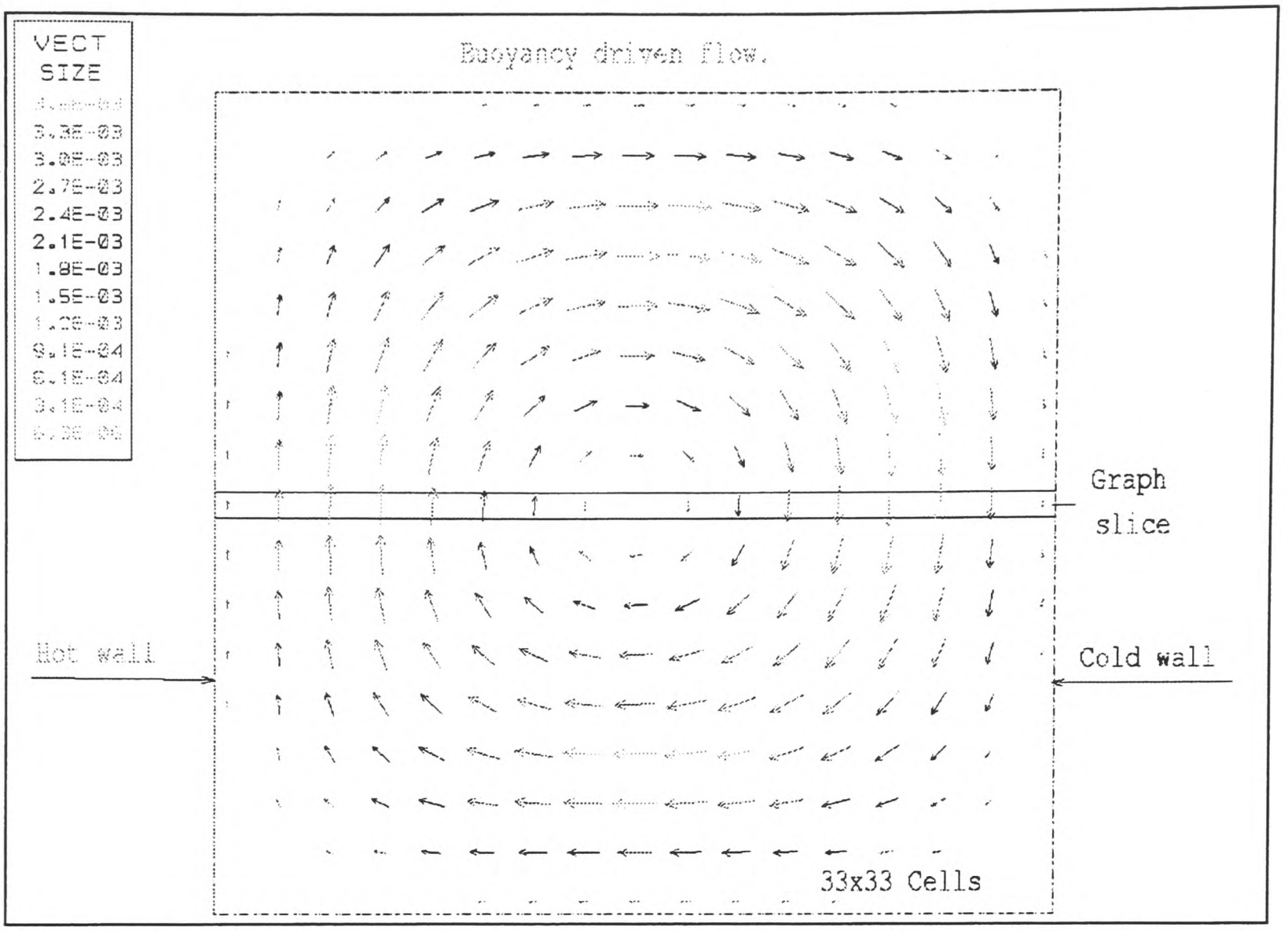


Figure 16 : Visualisation of flow vectors for natural convection.

The corresponding graph plot (Figure 17) shows the vertical velocity components across the width of the domain. This was the indicated slice of the natural convection buoyancy driven flow from Figure 16. These results have some minor differences because the simulation has not been run to full convergence. It should be noted that combined heat transfer and flow simulations are particularly difficult to converge because of the complexities inherent in the coupling between the heat and flow properties. Often such complex simulations are partially or significantly transient (with oscillatory flow behaviour) although this case has only been run in steady state mode.

It should be noted that the solution comparisons made in this section are not intended to validate the re-engineered code against either experiment or theory but rather to validate that the re-engineering process has not significantly altered the code behaviour.

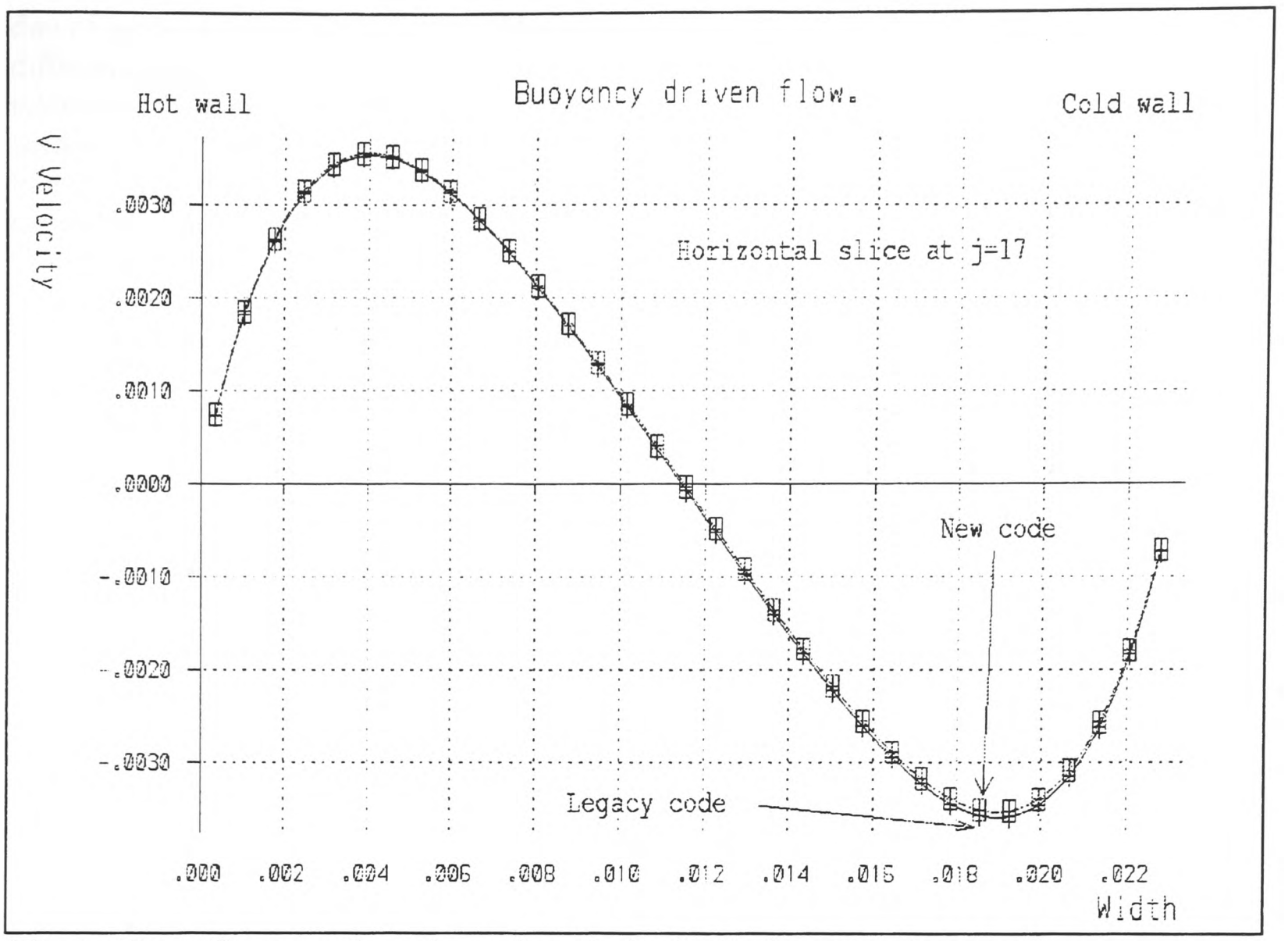


Figure 17 : Comparative values of vertical velocity for natural convection.

Appraisal of the re-engineering strategy:

The observed 50 percent performance overhead of the C++ implementation, on the Sun system, was probably the more realistic of the timing comparisons because both the Sun FORTRAN and Sun C++ compilers generate compatible binary code and have identical compilation options. Similar overheads have been reported in other numerical code translations and are mostly due to the introduction of more layers of function calls at a low level within the C++ code. This was particularly true of the case-study code implementation where two and sometimes three levels of function calls have been introduced. It is generally accepted that the introduction of a high degree of object-orientation and code clarity usually affect the performance adversely. Higher overheads have been recorded in C++ implementations where inheritance and virtual classes have been used.

The consistency between the simulation solutions of the legacy and re-engineered systems indicates that the codes exhibit highly compatible behaviour. This was coupled with an easily comprehensible implementation that was largely self documenting. The enhanced monitoring and debugging facilities of the re-engineered system have been used in subsequent research to good effect. These factors indicate that the re-engineering objectives have largely been met during this case study.

One of the major benefits of the re-engineering approach adopted in this research was that the newest version of the code could be compiled and tested after each incremental stage. In fact the testing was generally done after every major global change or after a single

day of work. The testing was simply batch automated to run overnight and a numerical file differencing utility, Numdiff[13] was written to compare the output data files to some small tolerance. This gives an assurance of continued consistency and meant that a worse case scenario was the loss of a single day of work. This contrasts with a new development where it may be many months, or even years, before the new system could be compared for consistency with legacy code.

There is a tendency in a re-engineering process to try "to run before you can walk" because enhancements and modifications can appear obvious to the developer while engaged on other tasks. The approach adopted in this research aimed to strictly order the modifications and later enhancements so that potential side effects are reduced.

Clearly the translation to procedural C++ could have been conducted on the unmodified legacy code but such a step would have been huge by comparison with the incremental stages described earlier. It would also have left the developer floundering in un-structured C++ with little feeling as to the functionality of the code or the scope for re-engineering. The re-structuring, re-naming and clarification of the legacy code prior to translation were vital to understanding the system and as a preparation for re-engineering.

As with many software systems, there are alternate implementation strategies. This was particularly true of the implementation of classes for the case study system. A truly object- oriented approach might have suggested having a virtual cell base class with sub-classes to represent all of the possible different types of cell. Although this technique would have been possible to utilise it was discarded because of the performance overheads of dynamic (run-time) linking. Also the system was targeted at CFD numerical software developers who are almost exclusively procedural FORTRAN-77 programmers. Extensive use of the object oriented and message passing paradigm would certainly confuse many of the intended users of the system.

Conclusions

The perceived problem of maintaining legacy code has increased over recent years. This is particularly true in the field of numerical simulation where some companies are partially or wholly reliant on large "in-house" legacy codes. Code re-use and re-engineering are desirable but can go drastically wrong with a huge waste of development resources. This paper demonstrates a re-engineering approach that has been applied to a medium sized FORTRAN numerical application with considerable success. The incremental strategy adopted has the benefit of ensuring that the delivery system has consistent behaviour with the legacy code. Many of the potential sources of error, inherent in the original system, have been removed or minimised during re-engineering.

The numerical consistency demonstrated by the results and the acceptable timings justify the use of re-engineering in this project. The strategy described is considered to be a suitable solution to the problem of maintaining medium sized legacy numerical systems. The ease with which certain system enhancements have been implemented supports the re-use of legacy code and vindicates the decision to use C++ as the target implementation language.

This case study leads to the conclusion that re-using legacy code need not be

intimidating or a great problem to continued software development. Indeed it arguable that software re-use can save costly development resources that producing (and debugging) a completely new system would require. Admittedly the approach adopted for this research still needs a significant development commitment and was by no means effortless, as some of the new re-engineering tools claim, however the clarity and flexibility of the delivery system justify the effort involved.

The only major drawback associated with this project was in user acceptance of the delivery system. Most of the existing CFD researchers are committed FORTRAN-77 programmers who are often intimidated by, and unwilling to learn, other programming languages. It is hoped that the demonstrable consistency of the results, code clarity and ease of use of the re-engineered system will encourage researchers to use the new system, in spite of its implementation in unfamiliar C++.

References

[1] Boehm B., Software Engineering, *IEEE Transactions on computers*, C. 25, No. 12, 1976, pp 1226-1241.

[2] Bodin F. et al., Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools, *OON-SKI'94 conference proceedings*, Oregon, USA., pp 122-136.

[3] Angus I. & Curtis W., From Fortran to Object Orientation: Experiences with a Production Flutter Analysis Code., *OON-SKI'94 conference proceedings*, Oregon, USA., pp 174-180.

[4] Byrne E., Software Reverse Engineering: A Case Study, *Software-Practice and Experience*, December 1991, Vol. 21(12), pp 1349-1364.

[5] Leschinzer M., Modeling turbulent recirculating flows by finite-volume methods -- current status and future directions, *International Journal of Heat and Fluid Flow*, September 1989, Vol. 10, No. 3, pp 186-202.

[6] Patankar S., *Numerical Heat Transfer and Fluid Flow*, Pub. Hemisphere, 1980.

[7] Patankar S. & Spalding, D., A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows, *International Journal of Numerical Heat and Mass Transfer - 15*, 1972, pp 1887-1906.

[8] Petridis M. & Knight B., FLOWES: An intelligent Computational Fluid Dynamics system, *Engineering Applications of Artificial Intelligence*, Pergamon, 1992, Vol. 5, No. 1, pp 51-58.

[9] Chow P., *Control Volume Unstructured Mesh Procedure for Convection-Diffusion Solidification Process*, PhD Thesis, January 1993, School of Mathematics, Statistics and Scientific Computing, The University of Greenwich, Wellington Street, Woolwich, SE18 6PF, England.

[10] "*SPAG*" is part of the plusFORT FORTRAN tool set, copyright 1986-1993, Polyhedron Software Limited, England.

[11] Petridis M., Knight B. & Edwards D., A Design for Reliable CFD Software, *Reliability and Robustness of Engineering Software II*, Eds. Brebbia, C. & Ferrante, A., Elsevier, 1991, pp 3-18.

[12] Ewer J., Knight B. & Petridis M., An Intelligent User Interface for Computational Fluid Dynamics Software, *Applications of Artificial Intelligence Techniques in Engineering VIII*, Eds Revski, G. et al., 1993, Vol. 1, pp 77-91.

[13] "*Numdiff*" is a public domain numerical file differencing utility written for this project by J. Ewer, School of Mathematics, Statistics and Scientific Computing, The University of Greenwich, Wellington Street, Woolwich, SE18 6PF, England.

[14] Kernigan B. & Wilson B., "lex - a lexical analysis tool", *The C programming language*, Pub. Prentice-Hall, 1988.

[15] Kernigan B. & Wilson B., "yacc - *y*et *a*nother *c*ompiler *c*ompiler", *The C programming language*, Pub. Prentice-Hall, 1988.

[c] Please note that the authors acknowledge all registered Trade Mark names and Copyright names mentioned in this text.

11.3 Copy of Conference Paper "The development and Application of Group Solvers in the SMARTFIRE Fire Field Model", Ewer J., Galea E., Patel M. and Knight B., Reproduced from Proceedings of Interflam '99, Edinburgh, UK, June/July 1999, Vol. 2, pp 939-950.

APPENDIX 3 : Conference Paper "The Development and Application of Group Solvers in the SMARTFIRE Fire Field Model", Ewer J., Galea E., Patel M. and Knight B., Proceedings of Interflam '99, Edinburgh, UK, 1999, Vol. 2, pp 939-950.

THE DEVELOPMENT AND APPLICATION OF GROUP SOLVERS IN THE SMARTFIRE FIRE FIELD MODEL.

J.A.C.Ewer, E.R.Galea, M.K.Patel and B.Knight.

Fire Safety Engineering Group
Centre for Numerical Modelling and Process Analysis,
University of Greenwich,
London SE18 6PF, UK
http://fseg.gre.ac.uk/

ABSTRACT

This paper describes a new solution technique - known as the ***"group solver"*** - currently under development within the SMARTFIRE Computational Fluid Dynamics environment. The group solver is used to obtain numerical solutions to the algebraic equations associated with fire field modelling. The purpose of the technique is to reduce the computational overheads associated with traditional numerical solvers typically used in fire field modelling applications. In the example, discussed in this paper, the group solver is shown to provide a 37% saving in computational time over a traditional solver.

INTRODUCTION

In traditional Computational Fluid Dynamics (CFD) based fire models [1], control of the numerical solver applies equally over all of the cells throughout the solution domain. In large geometry cases this can create a significant, and at times limiting, computational overhead. This is particularly true in cases where the fire occupies a relatively small proportion of an otherwise large solution domain for part, or all, of the simulation period. An example of this may be the early stages of fire growth within an airport terminal or a road/rail tunnel. The group solver concept attempts to address this problem algorithmically, by providing optimal processing in regions of the domain where and when it is required.

In the group solver concept, the solution domain is split into an arbitrary number of groups-of-cells. A group is defined as a unique collection of cells that can have solver control parameters independent from any other groups in the solution domain. Group solvers can be activated independently for each solved variable. Internally, the group solver makes use of standard numerical "point-by-point" solution methods such as JOR or SOR.

One way in which this may be achieved is by controlling the number of iterations that the solver performs in the various groups. For instance, the maximum number of iterations in an "Inactive group" will be considerably smaller than the number for an "Active group". As the solution develops, cells can migrate to and from groups, thus receiving more or less computational attention. The overall convergence criteria are still configured as for conventional problems so there should be no significant difference in the quality of the converged solution.

Previous research on group solver technology focussed on static group membership in a two-dimensional application [2]. In the remainder of this paper, the SMARTFIRE software will be briefly described followed by a detailed description of the group solver technique. The technique will then be demonstrated for both static and dynamic group membership through the use of a simple three-dimensional fire application.

THE SMARTFIRE FIRE FIELD MODEL

SMARTFIRE is an open architecture CFD environment with an integrated knowledge based system that attempts to make fire field modelling accessible to non-experts in CFD. There are three major components to the software: CFD code, User Interfaces, and Expert System. By embedding expert knowledge into the CFD software, it is hoped that fire field modelling is made more accessible to the fire engineer with limited CFD expertise. The expertise currently embedded within the code is used to support the critical task of mesh specification of fire field simulation scenarios. Expertise is also being developed for the optimal automated dynamic solution control of fire field simulations during the solution process [3].

The software is fully developed by the University of Greenwich using a combination of in-house and proprietary software building blocks. It is designed to run on PC's under the WIN95/98 or NT operating systems. The minimum computer platform required is a Pentium PC with 32 Mbytes of memory. The primary components of the software have been described previously [2-5] and so will not be repeated here. However, as this paper concentrates on the CFD engine, this will be briefly outlined here. Figure 1 shows the conceptual architecture of the software and its modules.

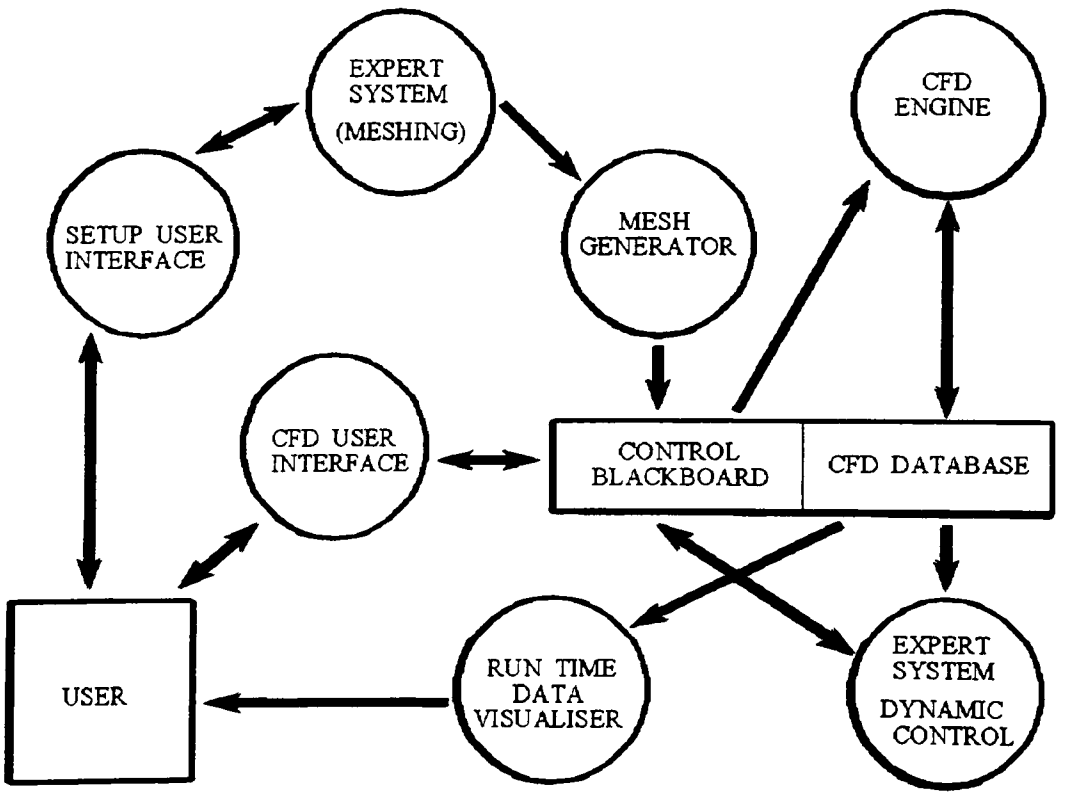


FIGURE 1: The modular architecture of SMARTFIRE.

The CFD engine of SMARTFIRE, called CWNN++, is written in C++ and has been developed in-house from an existing FORTRAN code [6]. CWNN++ uses validated numerical methods from the legacy FORTRAN code enhanced by object oriented developments in C++ and additional physics features that are required for fire field modelling [11]. CWNN++ uses three-dimensional unstructured meshes, enabling complex irregular geometries to be meshed without the need for body fitted co-ordinate grids. Unlike conventional CFD technology such as PHOENICS [7], FLOW3D [8], JASMINE [9] and SOFHIE [10], this allows extremely complex geometries to be meshed efficiently.

The CFD engine is under continual development and evaluation. As part of the testing procedure, model predictions are compared with other commercial CFD codes and against data generated through physical experimentation. Test cases [11] include standard CFD benchmark cases such as the backward facing step and moving lid problems and the Steckler room fire experiments [12]. Results produced by SMARTFIRE for the Steckler experiments [5, 11] compare favourably with those produced using commercial CFD codes such as CFDS-FLOW3D and PHOENICS [13].

The current release of the software is V2.01 [11]. In this version, the software represents fires as a transient volumetric heat and mass source. Standard models for gaseous combustion (i.e. diffusion and eddy-dissipation models), while not enabled in the general release software, are implemented and undergoing testing. The code uses the SIMPLE [14] algorithm and can solve turbulent (two equation K-Epsilon closure with buoyancy modification) or laminar flow problems under transient or steady state conditions. Radiation is represented through the use of an enhanced six-flux radiation model [11]. Further information concerning SMARTFIRE including a demonstration may be found on the World Wide Web [15].

BACKGROUND TO GROUP SOLVERS

Since SMARTFIRE is a truly unstructured mesh CFD code there are a limited number of reliable and general purpose numerical techniques available to solve the systems of algebraic equations for each of the primary field variables. Structured mesh CFD codes can exploit the structured nature of the data (e.g. using lines or planes) in various solvers to give more efficient solution than for the point-by-point iterative solvers commonly used in unstructured codes. One of the goals of this work has been to investigate and, if possible, exploit reliable techniques that prove to be of benefit to fire modelling within unstructured mesh CFD codes. One such technique, developed by the authors, is the group solvers.

In previous published research [2], it was demonstrated that a multiple region two-dimensional fire scenario could use static group solver techniques to save 31% of the processing time. The demonstration consisted of two rooms, one containing a fire, the other (separated by a wall) was initially free of fire or its influence. Under a given set of conditions, the partition was removed allowing the fire to spread into both compartments. The saving in computational effort was achieved by effectively removing the need to perform the computations for half of the solution domain during part of the simulation, prior to the partition removal. In this way approximately half of the solution domain was fully de-coupled from the simulation for part of the simulation. This work used the concept of a static "geometric" group to dictate the solution strategy in each region. Here we

demonstrate the concept of static groups in three-dimensional simulations and extend the concept to include "dynamic" groups.

DESCRIPTION OF GROUP SOLVERS

Group solvers are a conceptual extension of the simple linear, iterative, algebraic equation solvers usually referred to as Jaccobi Over Relaxation (JOR) or Successive Over Relaxation (SOR). At the most basic level these solvers involve the repetitive update of the solution of a property variable within each cell based on the contributions from nearest neighbouring cells, a portion of the previous solution value and the source quantity for each cell. In a CFD context the contributions from neighbouring cells represent the convection and/or diffusion of a physical property throughout the solution domain whilst the source indicates the creation or destruction of the physical property in the considered cell. The distinction between JOR and SOR solvers is that SOR always uses the most up-to-date versions of the solution when calculating the next update. This can make the SOR solver less stable than the JOR solver but it has the significant advantage of spreading the solution much more rapidly than JOR.

In the typical whole-domain JOR or SOR solver, the solution in each and every cell of the domain is updated repetitively until the difference between successive updates is sufficiently small. Clearly, if the solution domain contains many cells that are far removed from any active flow region or worse are totally de-coupled from the region of interest for a portion of the simulation, then not all of these JOR or SOR calculations are performing any useful advancement of the solution. This is especially true of many of the large complex geometries used in fire field modelling (e.g. whole building simulations).

The group solver concept allows the domain to be partitioned into "geometric" or "logical" (i.e. solution dependant) groups of cells that then use the iterative point-by-point update described above. The difference for the group solvers is that each group can have a unique set of control parameters to configure the maximum number of iterations to perform, the tolerance to use for convergence testing and/or the linear solver relaxation to be used. In this paper, the investigation only concerns the potential benefits of limiting the number of iterations that are used within each group of cells – while maintaining the desired level of convergence.

Since, in an unstructured code, a group does not need to be limited to some pre-configured geometric region it is possible to further extend the group solver techniques by allowing groups to determine their own cell-membership as the solution develops. This has been implemented within SMARTFIRE to allow an arbitrary number of groups which can contain either geometric or solution dependant membership (provided that each cell only exists in one group) and that furthermore the dynamic groups can exchange cells as the simulation solution develops. In practice, the dynamic membership is configured so that each dynamic group has an acceptance range of values which will trigger a non-member cell to be transferred into that group if its property value is within the configured range and if the cell is not already contained in another static "geometric" group.

The implementation of the group SOR solver requires particular care, at the algorithmic level, to ensure that the groups are not de-coupled into JOR connectivity between groups.

This scenario is possible if the looping between group-inner-iterations and between groups is mismanaged. One way this can occur is to give simple external looping for all groups and internally to loop for all configured group-inner-iterations. There are several possible methods of handling group-inner-looping for cases where groups have different numbers of configured inner-iterations. It was decided to interleave the processing between groups without using the simplest 1:1 interleave ratio, which while easier to implement could result in poor efficiency. The more complex interleaving technique, actually used, causes each group to be visited in turn and performs one (or more) of the inner iterations before moving to the next group. The looping amongst groups continues until each group reaches its configured maximum number of inner-iterations or until convergence is detected.

In order to attain maximal optimisation for cases with truly de-coupled (and hence uninteresting) group regions, it was also necessary to limit processing of such groups so that simple calculated variables are not updated. Mostly there is little difficulty in performing this optimisation because the support variables are generally closely linked, in their usage, to associated solved variables.

It should be noted that many of the variables in a fire modelling simulation have a definite "directionality" that can be exploited by matching the marching order of the cells within SOR solvers with this direction. SMARTFIRE has been implemented to use bi-directional marching order for all SOR type solvers, which gave a saving of up to 20% over the usual uni-directional marching order - when used on the simulation case described in this paper. All of the timings compare bi-directional group and whole-domain SOR solvers.

AN APPLICATION OF GROUPS SOLVERS IN A THREE-DIMENSIONAL FIRE SCENARIO

The case used to investigate group solvers is a preliminary investigation into fire spread between the floors of a multi storey building where window sizes are varied to modify the ejected plume behaviour. In the case presented here only the lower (ground) floor room is modelled together with the outer wall of the second and third floors above. In subsequent research it is intended that the upper floor rooms will also be fully modelled with windows that can be broken by the incident heat flux from the ejected spill plume.

In order to investigate the benefits of the group solvers a number of test cases were prepared. The geometry and mesh used in all of the tests was identical and great care was taken to ensure that the mesh was sufficiently refined across the height and width of the window, near the walls of the room and outside and just above the window. These considerations are critical to obtaining a reliable and accurate simulation of the ejected plume.

The geometry (see figure 2) was set up with room dimensions of 4.0m (x) X 3.4m (y) X 6.0m (z). The centrally located fire is represented as a volumetric heat load which is applied over a volume of 1.0m X 1.2m X 2.0m. The fire uses an "alpha t squared" power curve, which reaches 2.0 MW (using a fast growth rate) in three minutes of simulated time. The window aperture has a size of 2.0m (y) X 2.0m (z) and is centrally located on the high X-face of the room. The exterior wall, above the window, extends for a height of 10.5m vertically (to allow for the addition of two open rooms above fire room and a further room

height to move the free surface boundary sufficiently far away from any upper floor windows that may be used). The extended region beyond the window has the same Z-width as the room and extends for a distance of 6.0m in the X-direction in order to give ample room for the plume ejection. All of the surfaces of the extended region have a free surface boundary condition except for the floor, which is assumed to be solid. The outside region is assumed to be calm prior to the fire. The walls are assumed to be brick with a thickness of 0.1m. The mesh used for the simulation consisted of 40,572 cells with NX=36, NY=49 and NZ=23. The number of cells in the geometric regions was as follows: Dead region (non participating rooms above fire compartment i.e. de-coupled region) has 14,260 cells, Fire-room has 8,280 cells and the entire extended region has 18,032 cells (see figure 3).

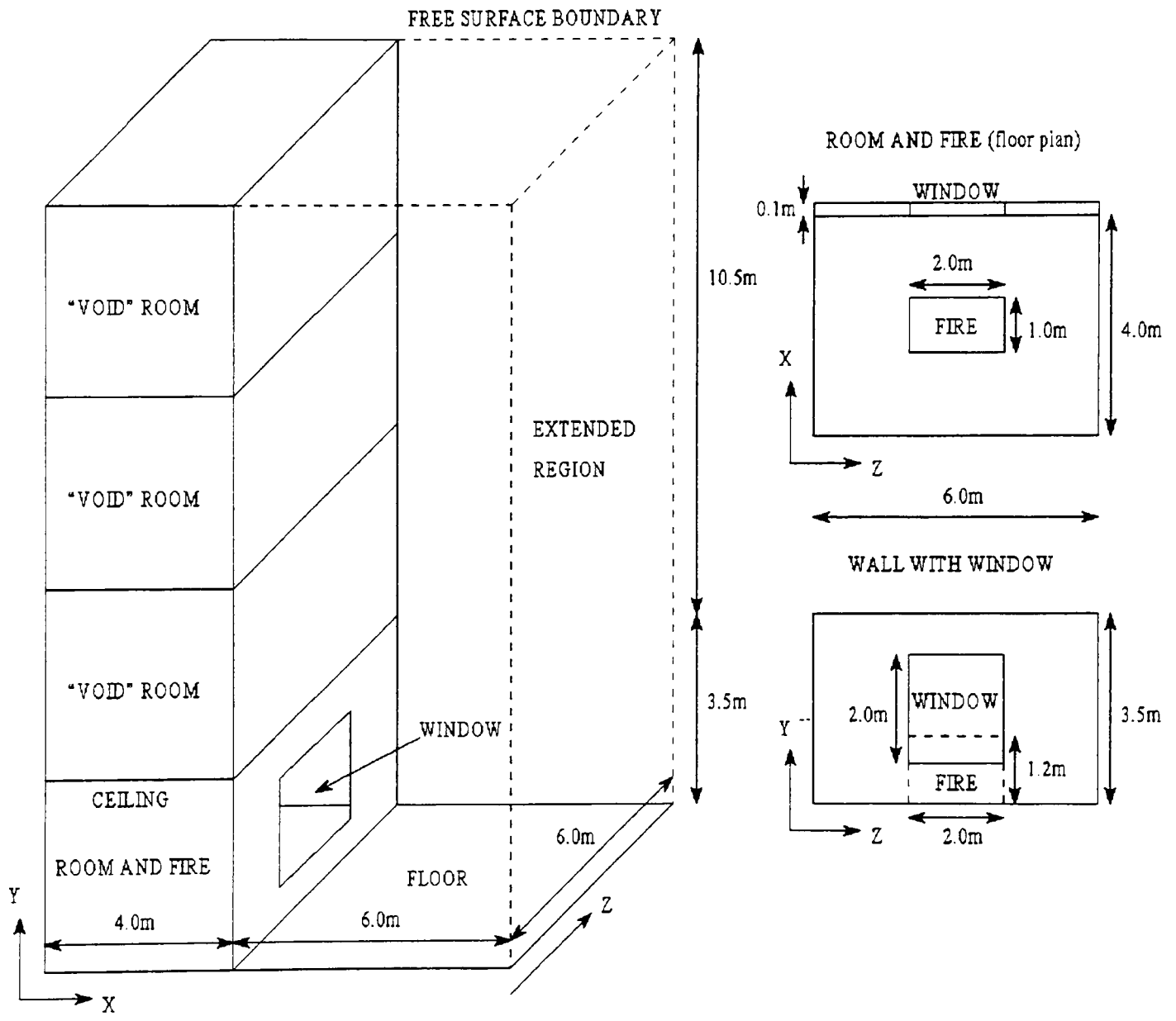


FIGURE 2 : The multi-storey geometry used for the group solver tests.

The simulation involves buoyancy driven flow with K-Epsilon turbulence model (buoyancy modified) and incorporates the enhanced six-flux radiation model [11]. The entire simulation was configured to perform 90 time steps of 2 second duration. The solver configurations used in the various simulations are summerised in table 1.

TABLE 1 : Summary of solver configurations used in simulations.

Variable(s)	Solver update method	Linear relax	False time relax	Whole domain iterations	"Active" group iterations	"Calm" group iterations	"Void" group iterations
Pressure	SOR	0.6	NA	50	50	12	0
Momentum	SOR	1.0	0.5	6	6	2	0
Turbulence	SOR	1.0	0.1	20	20	5	0
Enthalpy	SOR	1.0	2.0	30	30	8	0
Radiation	SOR	1.0	0.0	20	20	5	0

Furthermore all solvers were able to terminate their inner iterations if a common convergence level was reached. Each time step was forced to have all normalised variable residuals converged, to 1.0e-03, before the next time step could be started.

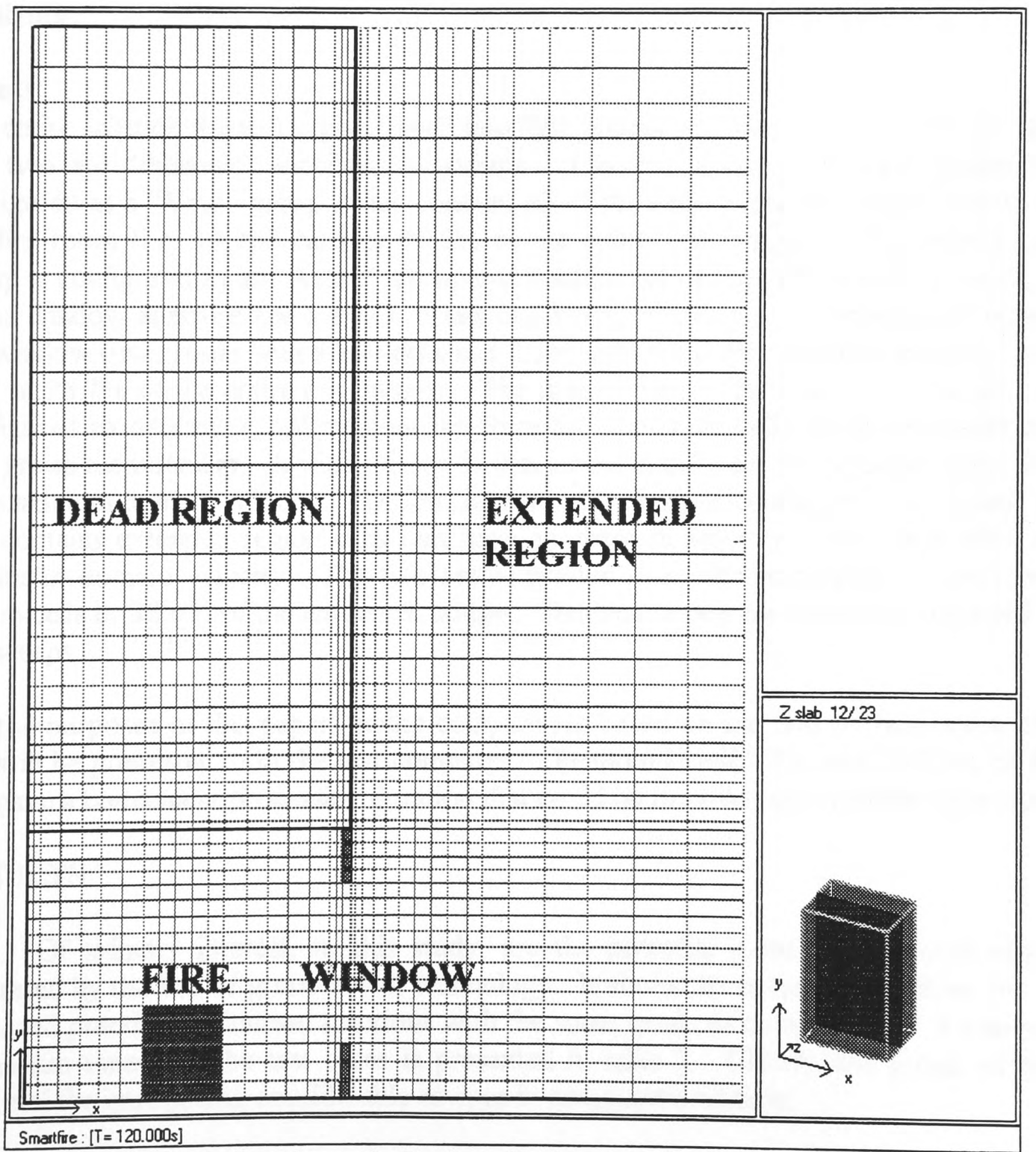


FIGURE 3 : Vertical slice through the domain showing the mesh and the various regions.

For comparison purposes, the following three test cases were simulated:

Case 1:
The simulation is configured with all solved variables using the whole domain SOR solvers as specified in Table 1. For comparison purposes this constitutes the base case. The group solvers are not utilised in this test and so the code is run in a conventional manner.

Case 2:
The entire solution domain is configured into two static "geometric" groups, one group configured as a "Void" group and another configured as an "Active" group (see table 1). The "Void" group contains all of the cells in the decoupled region above the fire room (i.e. 14,260 cells or 35.2% of the entire cell buget). The "Active" group contains all of the cells that are not in the "Void" group region (i.e. 26,312 cells or 64.8% of the entire cell budget). While the group solvers are activated, group membership remains the same throughout the simulation.

Case 3:
The entire solution domain is partitioned into four groups, two are static "geometric" groups and two are "dynamic" membership groups. The first group is a static group that is configured as a "Void" group which contains all of the cells in the de-coupled region above the fire room (i.e. 14,260 cells or 35.2% of the entire cell budget). The second "static" group is configured as an "Active" group and contains all of the cells in the fire room, those in the window aperture and a small rectangular block of cells that is immediately outside of the window (uses room with 8,280 cells and 2,366 cells from the extended region i.e. 10,646 cells or 26.2% of the entire cell budget). The third group is "dynamic" and "Active" and is configured to determine cell membership from the non-static cells of the extended region. The group membership selection criteria is that absolute cell velocity is greater than 10% of the maximum domain velocity. The fourth "dynamic" group is configured as a "Calm" group and contains extended region cells that have an absolute velocity of less than 10% of the maximum domain velocity. The two active groups share the remaining 15,666 extended region cells or 38.6% of the entire cell budget. Dynamic group membership is updated every 10 sweeps.

For the purposes of this paper, timing comparisons based on the first 50 time steps of each test will be presented. On the test computer (a Pentium II 400MHz with 256MB of RAM) this gave a convenient processing duration that could be run overnight without interruption.

RESULTS

Of primary interest, to this study, are the potential gains in numerical efficiency generated by the use of group solver technology. It should be noted that all three test cases produced practically identical solutions with the same levels of convergence. A comparison of the run times for the test cases is presented in table 2. Clearly, the group solver has potential for introducing considerable savings in computational time.

TABLE 2 : Comparison of group solver performance over the three test cases.

Test scenario	CPU time used for 50 time steps	Total number of sweeps used	Percentage time saving over case 1
Case 1 : Whole domain solvers	15h 51m 40s (57,100 seconds)	3095	0.0%
Case 2 : Static groups	11h 43m 36s (42,216 seconds)	3089	26.1%
Case 3: Static and dynamic groups	9h 56m 45s (35,805 seconds)	2919	37.3%

The fire dynamics in these test cases proceeded as expected. As the window opening to the fire compartment was considered narrow, a strong plume was ejected from the compartment. As the plume rotated and ascended vertically, it did not attach to the building façade. These results are consistent with earlier modelling work [16] and with reported experimental observations [17].

By 100 seconds, of simulated time, the rising plume outside of the compartment was fully developed and clearly unattached from the building façade. Continuing the simulation beyond this point merely increased the temperature of the fire compartment, the rising plume and the building façade.

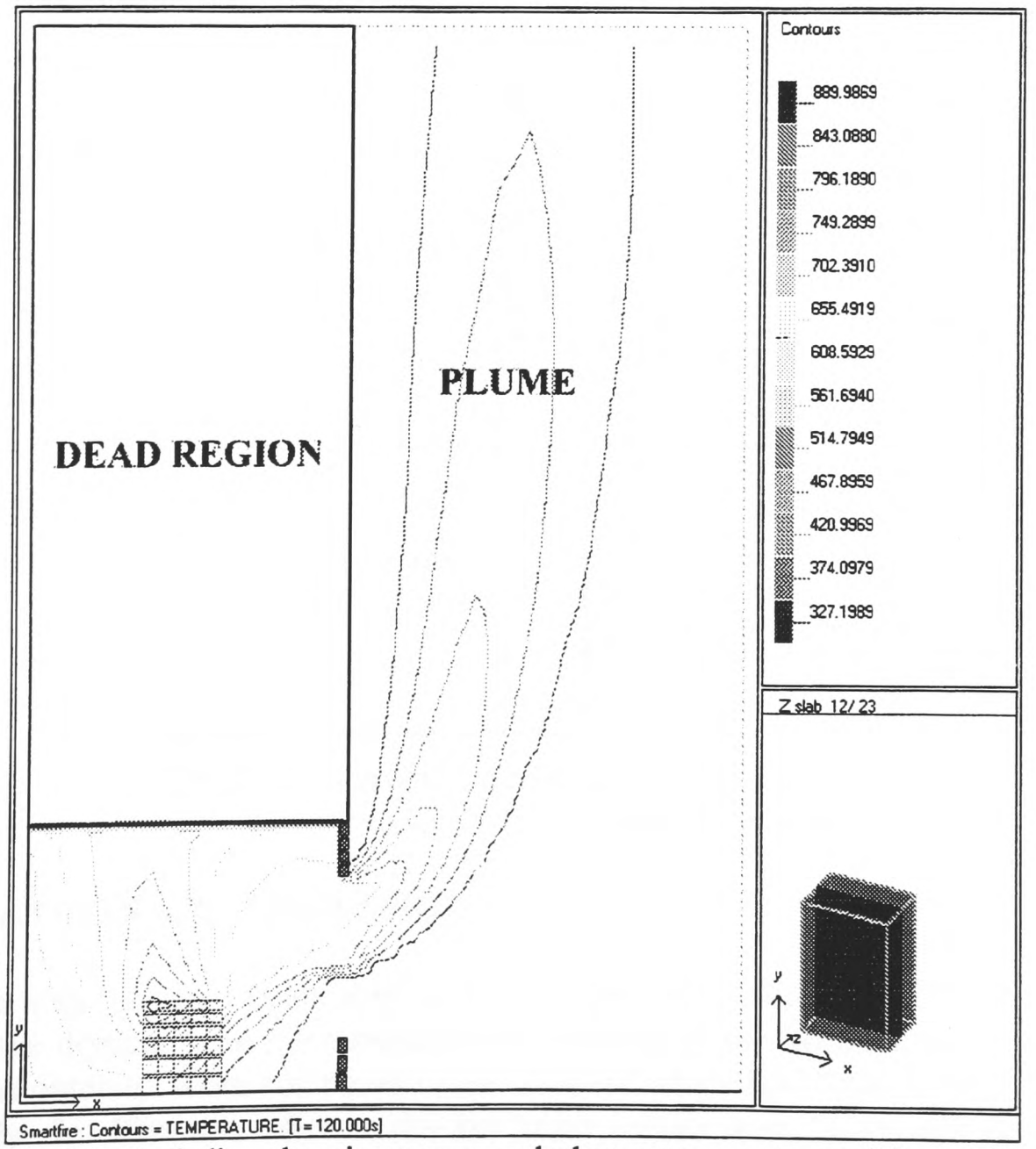


FIGURE 4 : Vertical slice showing room and plume temperatures (K) at 120 seconds.

The results for temperature displayed in figure 4 were taken at a simulation time of 120 seconds from the whole field SOR simulation in Case 1. Only the results from Case 1 are presented here as the comparable results from Case 2 and Case 3 displayed no apparent differences. Within the solution fields produced by Cases 1-3, maximum temperatures differed by at most, 1 Kelvin in the range of 318 to 914 Kelvin.

In order to verify that the dynamic group solver membership mechanisms were operating as expected, a vertical slice visualisation of group membership was created. This group visualisation (see figure 5) shows that the "active" dynamic group in the extended region has captured the plume extent correctly.

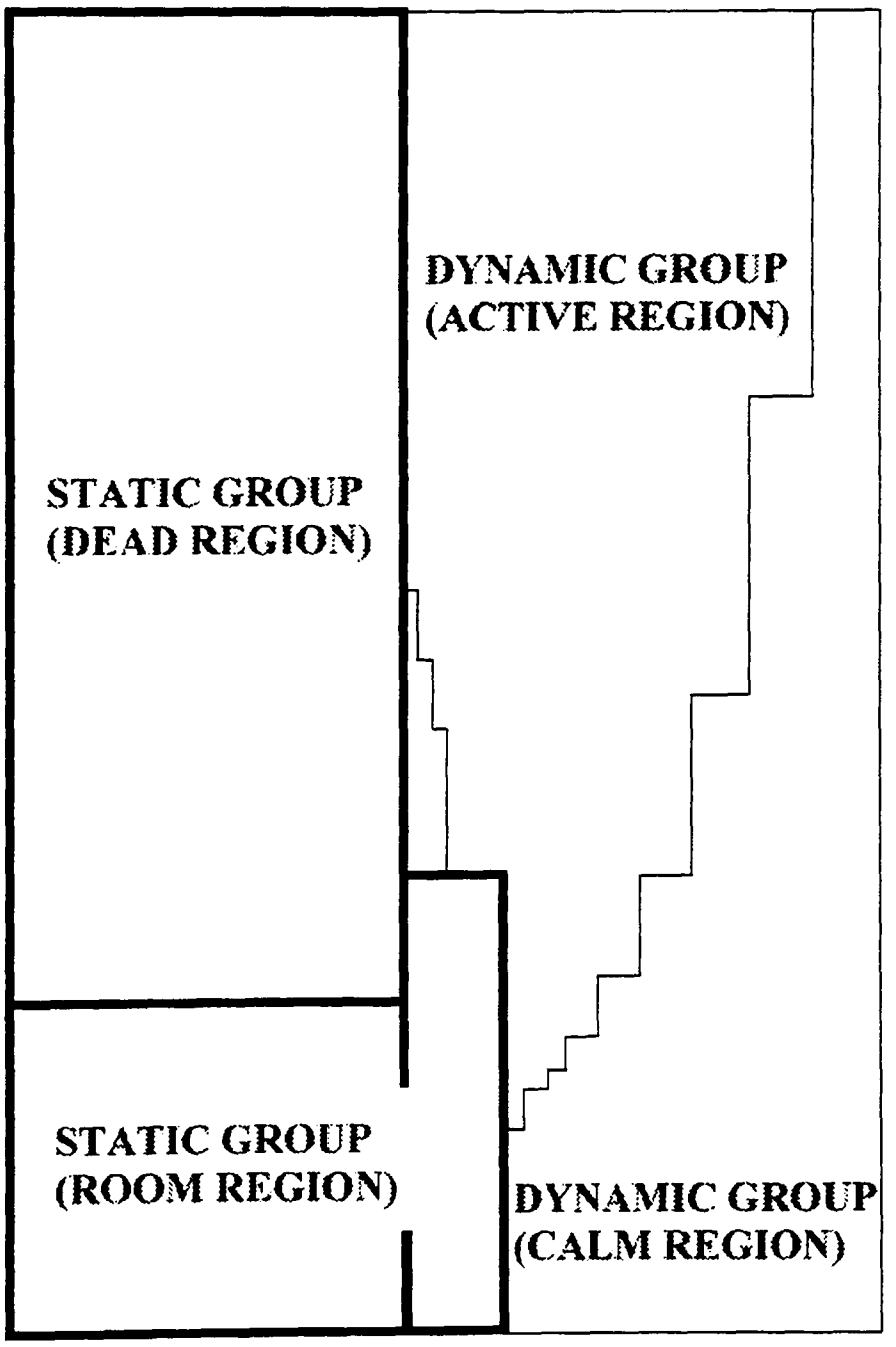


FIGURE 5 : Vertical slice showing static and dynamic group membership at 120 seconds for test case 3.

OBSERVATIONS AND DISCUSSION

The static group solvers used in Case 2 demonstrated that, by effectively removing 35.2% of the domain from the computations, a saving of processing time of 4h 8m 4s (or 26.1%) was obtained when compared to the standard whole field SOR solvers processing all cells equally. In effect this indicates that the group solvers were 74.2% efficient at removing the processing overhead of the de-coupled region from the simulation. While a 100%

efficiency may be desired, this result was anticipated because there are still many calculations performed in the "de-coupled" region for material properties and simple calculated variables. It is anticipated that this figure can be improved somewhat by increasing the use of "group" activated calculations within the rest of the CFD code.

In Case 3 both static and dynamic groups are used with the majority of the extended region being continuously evaluated for applied processing strategy. In this case an overall processing time saving of 5h 54m 55s (or 37.3%) was achieved when compared to the standard whole field SOR solvers processing all cells equally. It should be noted that much of this saving is due to the "de-coupled" void group which, as shown in Case 2, saves 26.1% of the processing. The remaining 11.2% saving is due to the optimisation of processing within the extended region which targets less solver processing in cells with relatively low velocity flow. The fact that this saving is comparatively less than for the "de-coupled" region is also anticipated. This can be explained by considering the work performed in the "de-coupled" and dynamic groups. In the "de-coupled" group, it was not necessary to build the system matrix coefficients for the member cells whereas cells in a group that performs one (or more) iterations must build the system matrix coefficients. Building the system matrix coefficients is relatively costly compared to solving the matrix.

CONCLUSIONS

The results indicate that there are large potential savings to be gained in the simulation of fire modelling scenarios by the targeting and optimisation of processing effort in fully de-coupled, suitably stratified or geometrically related flows. Furthermore, these savings need not result in compromised accuracy of the final solution. The techniques developed and presented here resulted in considerable run-time savings of up to 37% of processing time. It is anticipated that this figure can be improved significantly when a better understanding of the balancing required between groups and variables is achieved.

As group solvers are a new concept, there was little or no expertise to guide in the optimal selection of number of groups to use, the choice of group membership conditions and the relative amounts of processing used in each group. Furthermore there are a number of remaining group solver control options which were not varied during the test simulations.

It is anticipated that in large scale simulations, which may involve whole buildings, there are likely to be much greater savings possible with intelligent use of group solvers that can target the processing only on the active flow and fire regions until the solution characteristics in other regions become significant.

Current research effort is directed at gaining a better understanding of when it is appropriate to use groups and how best to balance the processing between groups in order to obtain optimal convergence and simulation times. Dynamic groups have been shown to give modest performance improvements but more work is needed to determine if there are any further benefits possible due to combined solution monitoring and dynamic knowledge based control of the processing within both the static and dynamic groups. Whilst the use of group solvers increases the complexity of the knowledge based control, it is also most likely to provide the most significant savings and most reliable solutions.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the financial support of the UK EPSRC through research grant GR/L56749 and the Loss Prevention Council.

REFERENCES

1. Galea E.R., "On the field modelling approach to the simulation of enclosure fires", Journal of Fire Protection Engineering, vol 1 (1), 1989, pp 11-22.

2. Taylor S., Petridis M., Knight B., Ewer J., Galea E.R. and Patel M., "SMARTFIRE: An Integrated CFD code and expert system for fire field modelling", Fire Safety Science, Proceedings of the 5th Int. Symp., Ed: Hasemi Y., 1997, pp 1285-1296.

3. Ewer J., Galea E.R., Knight B., Patel M., Janes D., Petridis M., "Fire Field Modelling using the SMARTFIRE Automated Dynamic Solution Control Environment", CMS Press, Paper Number 98/IM/41, ISBN 1899991387, London, 1998.

4. Taylor S., Galea E.R., Patel M., Petridis M., Knight B. and Ewer J., "SMARTFIRE: An Intelligent Fire Field Model", Proc. Interflam 96, Cambridge, UK, March 1996, pp 671-680.

5. Ewer J., Galea E.R., Taylor S., Patel M.K. and Knight B., "SMARTFIRE: An Intelligent CFD Based Fire Field Model", To appear in Journal of Fire Protection Engineering, 1999.

6. Ewer J., Knight B. and Cowell D., "Case Study: An Incremental Approach to Re-engineering a Legacy FORTRAN Computational Fluid Dynamics Code in C++", Advances in Engineering Software, vol. 22, 1995, pp 153-168.

7. Spalding D.B., "A General Purpose computer Program For Multi-Dimensional One- and Two- Phase Flow", Mathematics and Computers in Simulations, North Holland (IMACS), Vol. XXIII, 1981, 267.

8. FLOW3D Release 2.3.3 Reference Guide, CFD Dept AEA Harwell UK, Feb 1991.

9. Kumar S., Gupta A.K. and Cox G., "Effects of Thermal Radiation on the Fluid Dynamics of Compartment Fires", Fire Safety Science - Proc. of the Third Intl. Symp., 1991, pp 345-354.

10. Lewis M.J., Moss M.B. and Rubini P.A., "CFD Modelling of Combustion and Heat Transfer in Compartment Fires", Fire Safety Science, Proc. of the 5th Int. Symp., Ed: Hasemi Y., 1997, pp 463-474.

11. Galea E.R., Knight B., Patel M., Ewer J., Petridis M., and Taylor S., "SMARTFIRE V2.01 build 365, User Guide and Technical Manual", Smartfire CD and bound manual, 1999.

12. Steckler K.D., Quintiere J.G. and Rinkinen W.J., "Flow Induced By Fire in a Compartment", NBSIR 82-2520, National Bureau of Standards, Washington, 1982.

13. Kerrison L., Mawhinney N., Galea E.R., Hoffmann N. and Patel M.K., "A Comparison of Two Fire Field Models With Experimental Room Fire Data", Fire Safety Science - Proc. of the Fourth Intl. Symp., Ottawa, Canada, 13-17 July 1994, pp 161-172.

14. Pantakar S.V., "Numerical Heat Transfer and Fluid Flow", Intertext Books, McGraw Hill, New York, 1980.

15. www: http://fseg.gre.ac.uk/

16. Galea E.R., Berhane D. and Hoffmann N., "CFD Analysis of Fire Plumes Emerging from Windows with External Protrusions in High-Rise Buildings", Proc. Interflam 96, Cambridge, UK, March 1996, pp 835-839.

17. Yokoi S., "Study on the prevention of fire-spread caused by hot upward current." Report of the Building Research Institute, 1960.

11.4 SMARTFIRE USER GUIDE : Technical Reference.

SMARTFIRE TECHNICAL REFERENCE GUIDE

By J. Ewer, M. Patel and F. Jia.

TECHNICAL REFERENCE CONTENTS

NOMENCLATURE

Symbol	Meaning	Equation of first mention
$\underline{U}$	Velocity	1
T	Time	1
u_I	I^{th} velocity component	2
P	Pressure	2
S_I	I^{th} variable/model source term	2
x_I	I^{th} co-ordinate direction	2
H	Enthalpy	3a
K	Conductivity	3a
C_p	Specific heat capacity	3a
T	Temperature	3b
K	Kinetic energy	4a
P	Turbulent production rate	4a
G	Buoyancy rate	4a
$C_{1\varepsilon}$	Turbulent constant	4b
C_3	Turbulent constant	4b
R_f	Richardson Number	4b
$C_{2\varepsilon}$	Turbulent constant	4b
G	Gravity	4b
C_μ	Turbulent constant	4b
R	Radiation flux	5a
A	Absorption coefficient	5a
E	Emmisivity	5a
S	Scatter co-efficient	5a
I,J,K,L,M,N	Six-flux radiation directional fluxes	5c
F	Scalar quantity	6a
V_p, V_p^0	Volume of cell at current/previous time	9a
N	Normal component	9b
A_f	Face area	9b
d_{AP}	Distance between A and P cell nodes	9f
F_f	Strength of convection	10b
D_f	Strength of diffusion	10b
a_p	Pth cell coefficient	10c
a_{nb}	Neighboring cell coefficients	10c
e,w,n,s	East, West, North and South neighbors	11a

ρ	Density	1
μ_{eff}	Effective viscosity	2
ε	Dissipation rate	4a
μ_{lam}	Laminar viscosity	4a
υ_t	Turbulent kinematic viscosity	4a
σ_k , σ_ε	Turbulent constants	4d
β	Expansion coefficient	4d
σ	Stefan-Boltzmann constant	5b
Γ_I	I^{th} variable diffusion coefficient	7
ϕ	Dependent variable	8
ϕ_p	P^{th} cell variables	10c
ϕ_{nb}	Neighboring cell variables	10c
ϕ^T	Transpose of vector	11e
Δy_p	Cell center to wall distance	13d
τ_w	Wall shear	13e
φ	Dependent variable	13g

Theoretical Background

1.0 Introduction

This document presents a mathematical description of the SMARTFIRE code including governing equations, discretisation, solution algorithm, linear solvers and boundary conditions. The objective is to inform the user as to how the governing equations are handled in order to:

(i) Provide some insight into the numerical schemes that may help in understanding the performance of the code in terms of stability and convergence and,

(ii) Illustrate how the information provided by the user impacts on the performance of the code and the final solution.

It is felt that a better understanding of the numerical techniques used in SMARTFIRE will considerably improve the ability of the user to achieve the ultimate objective, which is obtaining a converged and stable solution for the flow situation under study.

1.1 Basic Equations used in SMARTFIRE

The equations representing the conservation of mass, momentum and energy for transient flows in Cartesian co-ordinates assume the following form:

1.1.1 Mass Conservation

For any flow situation, the flow field should satisfy the mass continuity equation given by:

$$\frac{\partial \rho}{\partial t} + div(\rho \underline{u}) = 0 \qquad (1)$$

1.1.2 Momentum Conservation

The conservation of momentum in the three co-ordinate directions is given by the equation:

$$\frac{\partial(\rho u_i)}{\partial t} + div(\rho \underline{u} u_i) = -\frac{\partial P}{\partial x_i} + div(\mu_{eff}\, grad u_i) + S_{u_i} \qquad (2)$$

where u_i is the velocity in the x, y and z directions and P is the pressure.

1.1.3 Energy Conservation

For energy conservation, we solve the enthalpy form of the equation given by:

$$\frac{\partial(\rho h)}{\partial t} + div(\rho \underline{u} h) = div\left\{(k + \frac{\rho \upsilon_t}{\sigma_T}) grad\left(\frac{h}{c_p}\right)\right\} + S_h \qquad (3a)$$

where the temperature is evaluated from the expression

$$T = \frac{h}{c_p} \qquad (3b)$$

1.1.4 Turbulence Model

The buoyancy modified two-equation (k-ε) turbulence model represents turbulent flow. The model consists of the turbulent kinetic energy equation

$$\frac{\partial k}{\partial t} + div(\rho \underline{u} k) = div\left(\left[\mu_{lam} + \frac{\rho \nu_t}{\sigma_k}\right] grad k\right) + P + G - \rho\varepsilon \qquad (4a)$$

and the dissipation rate equation

$$\frac{\partial \varepsilon}{\partial t} + div(\rho \underline{u} \varepsilon) = div\left(\left[\mu_{lam} + \frac{\rho \nu_t}{\sigma_\varepsilon}\right] grad \varepsilon\right) + \frac{\varepsilon}{k}\left[C_{1\varepsilon}\left(P + C_3 \max(G,0)\right) - C_{2\varepsilon}\rho\varepsilon\right] \qquad (4b)$$

where P represents the turbulent production rate

$$P = 2\rho\nu_t\left\{\left(\left[\frac{\partial u}{\partial x}\right]^2 + \left[\frac{\partial v}{\partial y}\right]^2 + \left[\frac{\partial w}{\partial z}\right]^2\right)\right\} \rho\nu_t\left\{\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z}\right)^2\right\} \qquad (4c)$$

and G represents the Buoyancy term, given by

$$G = -\beta g \rho \nu_t \frac{\partial T}{\partial y} \qquad or \qquad G = -g\rho\nu_t \frac{\partial \rho}{\partial T} \qquad (4d)$$

and

$$\beta = -\frac{1}{\rho}\frac{\partial \rho}{\partial T} \qquad (4e)$$

The apparent turbulent viscosity is evaluated by using the expression

$$\upsilon_t = C_\mu \frac{k^2}{\varepsilon} \tag{4f}$$

The turbulence model contains five constants that are adjustable. The standard k-ε turbulence model employs the first five values for the constants given in the table below. The final value is for the buoyancy correction applied to the standard turbulence model.

C_μ	σ_k	σ_ε	$C_{1\varepsilon}$	$C_{2\varepsilon}$	C_3
0.09	1.0	1.3	1.44	1.92	1.0

1.1.5 Radiation Models

It is essential that when simulating fires, we represent adequately the characteristics of heat transfer and energy balance in the model. Within the fire model there are two primary modes of heat transfer, namely convection and radiation. While convective heat transfer is accounted for by the transport equations, radiative heat transfer requires a separate sub-model. Within SMARTFIRE, two radiation models are provided. These are (a) the Radiosity model and (b) the Six-Flux Radiation model. The Radiosity model is simple in nature and involves the solution of a single extra variable that is the radiant potential within each cell. While this ensures that the model is efficient in terms of CPU time it is a crude representation of radiation. The modified Six-Flux radiation model on the other hand solves for six equations, one in each Co-ordinate direction (both positive and negative directions) and makes the model more accurate but less efficient in terms of CPU time, when compared with the Radiosity model. These models are presented below:

1.1.5.1 Radiosity Model

The equation for the Radiosity, R, takes the form

$$\frac{d}{dx_i}\left[\frac{4}{3(\alpha+s)}\frac{dR}{dx_i}\right]+\alpha\,(E-R)=0 \tag{5a}$$

where α is the absorption coefficient, s is the scattering coefficient and E is the black body emissive power of the fluid calculated using

$$E = \sigma T^4 \tag{5b}$$

where T is the temperature of the fluid and σ is the Stefan-Boltzmann constant.

Transfer of heat through radiation leads to a source in the enthalpy equation equal to the negative of the source in the Radiosity equation, given below:

$$S_{radiosity} = \alpha\ (E - R) \tag{5c}$$

1.1.5.2 Six-Flux Radiation Model

In the six-flux radiation model heat fluxes R_i , are calculated by solving additional conservation equations in each component direction which have the form:

$$\left.\begin{aligned}
\frac{dI}{dx} &= -(\alpha + s)I + \alpha E + \frac{s}{6}(I + J + K + L + M + N) \\
\frac{dJ}{dx} &= +(\alpha + s)J - \alpha E - \frac{s}{6}(I + J + K + L + M + N) \\
\frac{dK}{dy} &= -(\alpha + s)K + \alpha E + \frac{s}{6}(I + J + K + L + M + N) \\
\frac{dL}{dy} &= +(\alpha + s)L - \alpha E - \frac{s}{6}(I + J + K + L + M + N) \\
\frac{dM}{dz} &= -(\alpha + s)M + \alpha E + \frac{s}{6}(I + J + K + L + M + N) \\
\frac{dN}{dz} &= +(\alpha + s)N - \alpha E - \frac{s}{6}(I + J + K + L + M + N)
\end{aligned}\right\} \tag{6a}$$

where α is the absorption coefficient, s is the scattering coefficient E is the black body emissive power of the fluid and I, J, K, L, M and N the six coordinate direction radiative fluxes.

Transfer of heat through radiation leads to a source in the enthalpy equation given by:

$$S_{six-flux} = \alpha\ ((I - E) + (K - E) + (M - E) + (J - E) + (L - E) + (N - E)) \tag{6b}$$

1.1.5.3 Absorption Coefficient

The absorption coefficient is evaluated using the following piecewise linear approximation:

$T < 50\ ^\circ C$	$\alpha = \alpha_{ambient}$
$T > 50\ ^\circ C$ and $T < (T_{plume}/2)$	$\alpha = \alpha_{ambient} + (c(T_{plume}/2) - \alpha_{ambient})/((T_{plume}/2)-50)(T-50)$
$T > (T_{plume}/2)$	$\alpha = cT$

(6c)

1.1.6 Species Conservation

The conservation of any scalar quantity, f, is represented by the equation given below:

$$\frac{\partial(\rho f)}{\partial t} + div(\rho \underline{u} f) = div(\Gamma_f grad(f)) + S_f \tag{7}$$

1.1.7 Auxiliary Equations

The density is calculated using the Ideal gas law, given by

$\rho = PW / RT$

where P is pressure, W is molecular weight, R is Universal gas constant and T is temperature.

1.2 General Scalar Equation

From the brief introduction of the equations used to represent complex fluid flow, it is clear that all the equations can be cast into a generalised form given by

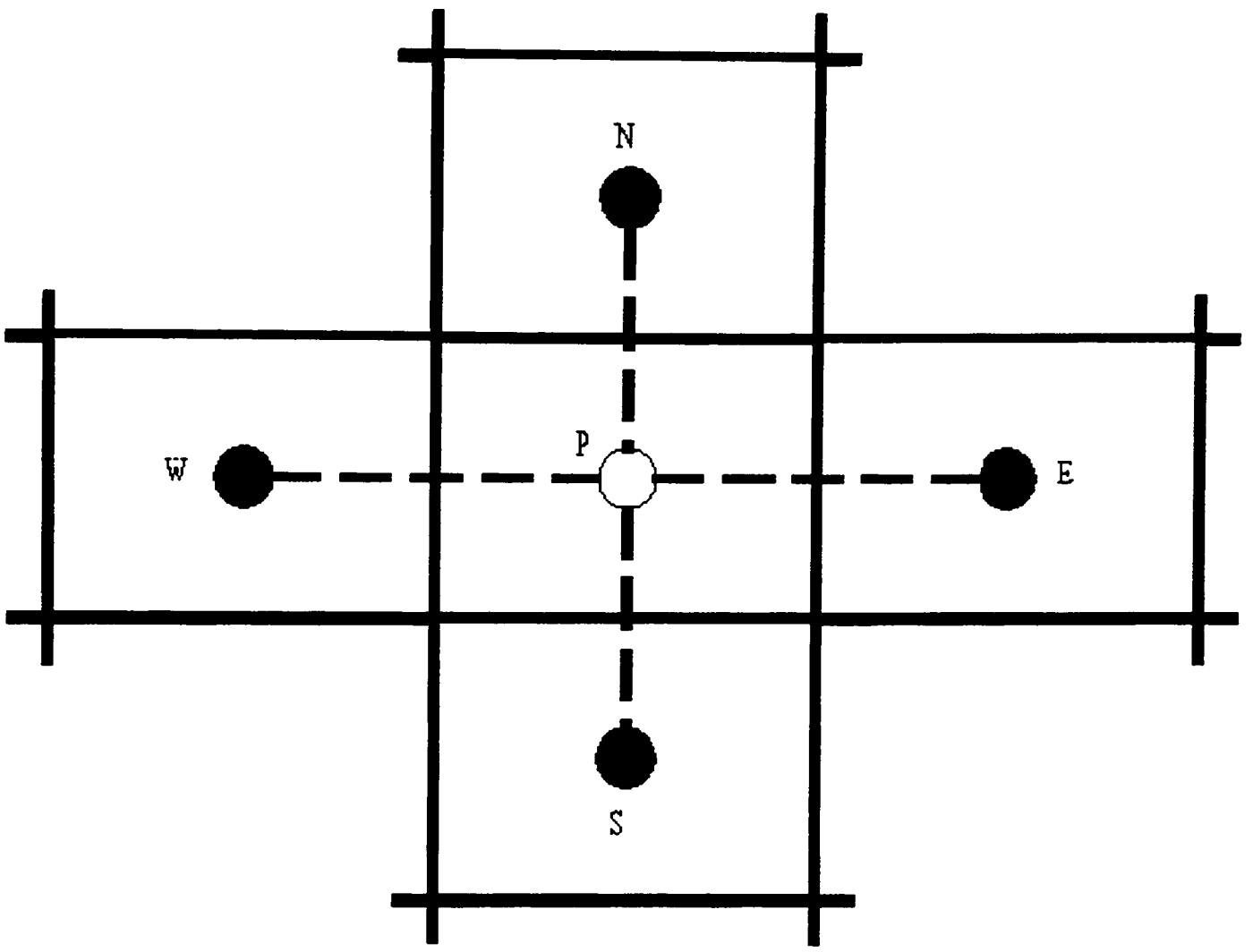
$$\underset{\text{Transient}}{\frac{\partial(\rho\phi)}{\partial t}} + \underset{\text{Convection}}{div(\rho \underline{u}\,\phi)} = \underset{\text{Diffusion}}{div(\Gamma_\phi grad(\phi))} + \underset{\text{Source}}{S_\phi} \qquad (8)$$

where ρ is density, $\underline{\boldsymbol{u}}$ is the vector velocity, Γ is the diffusion coefficient for the quantity ϕ and $\boldsymbol{S}$ is the source term for ϕ at any point. The four terms in the equation are the transient term, the convection term, the diffusion term and the source term. This equation is commonly known as the convection-diffusion form of transport equations.

1.2.1 Approximations of the Terms

All the terms in the convection-diffusion equation need to be approximated in order for the equation to be solved. In what follows there are no details as to how this process is achieved, however the final forms of the approximations are presented for completeness. The steps are

(1) Discretise the flow domain into a collection of control volumes,
(2) Integrate the individual terms over the surface or volume of each control volume,
(3) Approximate first derivatives using upwind values,
(4) Approximate face values for dependent quantities using sensible averaging approaches,
(5) Finally construct the full system of algebraic equations to solve.



The previously mentioned approximations are based on a typical control volume arrangement as depicted above. This is a typical 2D Control Volume set-up. The computational molecule shows the influence of neighbouring control volumes (North, South, West and East) on the control volume of interest (labelled P). It should be noted that SMARTFIRE actually uses a 3D unstructured mesh but similar concepts apply.

1.2.1.1 Transient Term

The transient term is approximated using the backward difference technique. This gives

$$\int_{t-\Delta t}^{t}\int_{V}\frac{\partial(\rho\phi)}{\partial t}dV\,dt \cong \frac{V_P\rho_P\phi_P - V_P^0\rho_P^0\phi_P^0}{\Delta t} \qquad (9a)$$

1.2.1.2 Convection Term

The convection term is the most important term in the equation. Care must be taken in approximating this term, as inappropriate approximations can lead to large errors or instability problems if not handled with care. This gives

$$\int_V div(\rho \underline{u}\phi)dV = \int_S \rho(\underline{u}\cdot\underline{n})\phi dS \quad \cong \quad \sum_f \rho_f(\underline{u}\cdot\underline{n})_f A_f \phi_f \tag{9b}$$

In this equation the value of ρ_f is given the value in the upwind element. Thus

$$\rho_f = \rho_P \quad if \quad (\underline{u}\cdot\underline{n})_f > 0.0 \quad and \quad \rho_f = \rho_A \quad if \quad (\underline{u}\cdot\underline{n})_f < 0.0 \tag{9c}$$

The convected quantity, ϕ, at the face needs further approximation. One possible approach is to use arithmetic averaging, e.g.

$$\phi_f = \alpha_f \phi_P + (1-\alpha_f)\phi_A \tag{9d}$$

Assuming this choice, then the final form of the discretised convection term becomes

$$\sum_f \rho_f(\underline{u}\cdot\underline{n})_f A_f\left[\alpha_f \phi_P + (1-\alpha_f)\phi_A\right] \tag{9e}$$

Other possible choices of the approximation of the convected, ϕ, are presented in the section entitled discretisation schemes.

1.2.1.3 Diffusion Term

The diffusion term is approximated using the approximation:

$$\int_S div(\Gamma_\phi grad(\phi))dV = \int_S \Gamma_\phi grad(\phi)\cdot \underline{n} dS \quad \cong \quad \sum_f (\Gamma_\phi)_f A_f \left(\frac{\phi_A - \phi_P}{d_{AP}}\right) \tag{9f}$$

where the diffusion coefficient is approximated by

$$(\Gamma_\phi)_f = \frac{(\Gamma_\phi)_A(\Gamma_\phi)_P}{\alpha_f(\Gamma_\phi)_P + (1-\alpha_f)(\Gamma_\phi)_A} \tag{9g}$$

1.2.1.4 Source Term

In general, since the source term is a function of the dependent variable, ϕ, a linearized form is used in the final discretised equation to ensure diagonal dominance of the system matrix. This form is as given below:

$$S_\phi = S_C - S_P\phi = V_P(S_C - S_P\phi_P) \tag{9h}$$

1.3 Overall Discretisation Equation

Having obtained expressions for the discretised form of each of the terms in the conservation equation in the previous section, the discretised form of the full equation is obtained by simply adding together all these contributions. Assuming arithmetic averaging for the evaluation of the face value of ϕ in the convection term, the discretised equation becomes

$$\frac{(\rho_P \phi_P V_P - \rho_P^0 \phi_P^0 V_P^0)}{\Delta t} + \sum_f \left[\rho_f (\underline{u} \cdot \underline{n})_f \{\alpha_f \phi_P + (1-\alpha_f)\phi_A\} A_f - (\Gamma_\phi)_f \left(\frac{\phi_A - \phi_P}{d_{AP}} \right) A_f \right] = (S_C - S_P \phi_P) V_P \quad (10a)$$

Defining the quantities F_f and D_f as

$$\left. \begin{aligned} F_f &= A_f \rho_f (\underline{u} \cdot \underline{n})_f \\ D_f &= A_f (\Gamma_\phi)_f / d_{AP} \end{aligned} \right\} \quad (10b)$$

where F_f is the strength of the convection of ϕ and D_f is the diffusion conductance, the equation can be further be simplified and written in the following form:

$$a_P \phi_P = \sum_{nb} a_{nb} \phi_{nb} + b \quad (10c)$$

where the summation is over all neighboring elements and the equations for the coefficients, a_p and a_{nb} in the above equation are given by

$$\left. \begin{aligned} a_{nb} &= D_f - (1-\alpha_f) F_f \\ a_P &= \sum_f (D_f + \alpha_f F_f) + a_P^o - S_P V_P \\ a_P^o &= \frac{\rho_P^0 \phi_P^0 V_P^0}{\Delta t} \\ b &= S_C V_P + a_P^o \phi_P^0 \end{aligned} \right\} \quad (10d)$$

The convected quantity of the dependent variable, ϕ, can be represented in a number of ways. The most commonly used technique is to use the upwind value. This is known as the upwind scheme, where the face value is approximated by the following rule:

$$\phi_f = \phi_P \quad if \quad F_f > 0.0 \quad and \quad \phi_f = \phi_A \quad if \quad F_f < 0.0 \quad (10e)$$

In which case the coefficients become

$$\left.\begin{aligned} a_{nb} &= D_f + \max(-F_f, 0.0) \\ a_P &= \sum_f \left[D_f + \max(F_f, 0.0)\right] + a_P^o - S_P V \\ a_P^o &= \frac{\rho_P^0 \phi_P^0 V_P^0}{\Delta t} \\ b &= S_C V_P + a_P^o \phi_P^0 \end{aligned}\right\} \qquad (10f)$$

In general a wide choice is available for the evaluation of the convected face value of ϕ. To incorporate a generalized version of the final discretised equation, we introduce a function A(|P|) which allows for any differencing scheme to formulated and incorporated, where P is the Peclet number given by F_f / D_f. This gives

$$\frac{\rho_P \phi_P V_P - \rho_P^0 \phi_P^0 V_P^0}{\Delta t} + \sum_f \left[\left\{D_f A(|P_f|) + \max(-F_f, 0.0)\right\}(\phi_P - \phi_A) + F_f \phi_P\right] = (S_C - S_P \phi_P) V_P \qquad (10g)$$

where the expression for A(|P|), for various schemes are given in the Table below.

Scheme	Formulae for $A(\|P\|)$
Central Differencing	1 - 0.5 $\|P\|$
Upwind	1
Hybrid	Max (0, 1 - 0.5 $\|P\|$)
Power Law	Max (0, (1 - 0.1 $\|P\|$)^5)
Exponential	$\|P\|$ / exp ($\|P\|$) - 1

In which case the coefficients become

$$\left.\begin{aligned} a_{nb} &= D_f A(|P|) + \max(-F_f, 0.0) \\ a_P &= \sum_f \left[D_f A(|P|) + \max(F_f, 0.0)\right] + a_P^o - S_P V \\ a_P^o &= \frac{\rho_P^0 \phi_P^0 V_P^0}{\Delta t} \\ b &= S_C V_P + a_P^o \phi_P^0 \end{aligned}\right\} \qquad (10h)$$

Note: The relative merits of the various differencing schemes are not discussed here.

1.4 Algebraic Equations

The algebraic equations, as described in the previous section, need to be solved using

appropriate methods. The choice of solution techniques used will effect both the accuracy of the solution and the effort required obtaining the solution. Thus it is important that several solution techniques are available for solving a multitude of different problems. The next section briefly describes the solution techniques used within SMARTFIRE.

1.4.1 Solution of the Algebraic Equations

The starting point for the solution of any equation is the set of algebraic equations. In this case we use the following representation:

$$Ax = b \tag{10i}$$

where A is a matrix of n x m elements and x and b are vectors of n elements.

The finite-volume discretisation approach results in a set of algebraic equations, which when represented in matrix form generates quite a large system matrix. Due to the nature of the stencil used, the system matrix although large, is quite sparse. Since the resulting algebraic equations are not linear in nature as the coefficients are themselves functions of other dependent variables, it is prudent to use iterative solvers to attain solutions efficiently.

A number of solution techniques are available in SMARTFIRE namely, the Jacobi Over Relaxation (JOR) method, the Successive Over Relaxation (SOR) method, Conjugate Gradient Method (CGM), Bi-Conjugate Gradient method (BiCG) and the Whole Field Solver. The two most frequently used solution techniques for point by point linear equations are the JOR method and the SOR method. Each of these techniques are briefly described below.

1.4.1.1 JOR / SOR SOLVER

For the JOR technique a typical brick-shaped cell is updated as follows:

$$\phi_{p,JOR} = \frac{\left(a_e\phi_{e,old} + a_w\phi_{w,old} + a_n\phi_{n,old} + a_s\phi_{s,old} + a_h\phi_{h,old} + a_l\phi_{l,old} + b\right)}{a_p} \tag{11a}$$

whereas for the SOR techniques a typical brick-shaped cell is updated as follows:

$$\phi_{p,SOR} = \frac{\left(a_e\phi_{e,new} + a_w\phi_{w,new} + a_n\phi_{n,new} + a_s\phi_{s,new} + a_h\phi_{h,new} + a_l\phi_{l,new} + b\right)}{a_p} \tag{11b}$$

The final cell value is then updated, using linear relaxation as follows:

$$\phi_{p,new} = relax * (\phi_{p,old} - \phi_{p,(JOR\ or\ SOR)}) + \phi_{p,(JOR\ or\ SOR)} \tag{11c}$$

1.4.1.2 CGM SOLVER

A general iterative scheme could be considered in the form,

$$\phi_{p,new} = f(\phi^i, \underline{\underline{A}}, \underline{b}) \qquad (11d)$$

where f is a function that would aid the convergence of ϕ^i to $\phi_{p,new}$. For the CGM scheme, the function f is prescribed by

$$f(\phi) = 0.5\underline{\phi}^T \underline{\underline{A}}\underline{\phi} - b^T \underline{\phi} + c \qquad (11e)$$

the gradient of which reduces to (assuming symmetric A),

$$f'(\phi) = \underline{\underline{A}}\underline{\phi} - \underline{b} \qquad (11f)$$

Furthermore, if A is positive-definite, then the solution of Ax=b would be the global minimum of the function f. The CGM solver aims to obtain the minimum of f as the solution.

1.4.1.3 BiCG SOLVER

The BiCG solver is a variant of the CGM solver where the square matrix A can be non-singular and non-symmetric. There is a pre-stage to cast the non-singular and unsymmetric matrix in to a symmetric positive definite matrix, as required by the CGM solver. This is achieved by using the transpose of A as given below:

$$[(\underline{\underline{A}})^T \underline{\underline{A}}]\phi = (\underline{\underline{A}})^T \underline{b} \qquad (11g)$$

1.4.1.4 WHOLE FIELD SOLVER

The whole field solver, as the name suggests aims at solving the equations at the end of the sweep, when all the nodal points have been visited and the system matrix build up. The whole field solver differs from a point wise solver (for example JOR, SOR) by using extra inner loops and back-substitution to ensure that the updated solution influences every other control volume.

1.4.2 SIMPLE Solution Procedure

The fire model comprises a set of highly non-linear and tightly coupled equations. When solving such a set of equations, the order and manner in which the solution progresses is vital. Several solution procedures can be used to solve the equations. In SMARTFIRE the SIMPLE solution procedure is used to solve the equation set. The procedure is outlined below:

Step 1: Guess the initial pressure field p*

Step 2: Solve momentum equations with guessed pressure field to obtain u*, v*, w*

Step 3: Solve for the pressure correction p'

Step 4: Calculate the new pressure field using p* and p'

Step 5: Calculate the new velocity components u, v and w using u*, v*, w* and p'

Step 6: Solve the other conserved quantities i.e. Enthalpy, Temperature, Turbulence, Concentration, Radiation, density, viscosity, etc.

Step 7: Treat the corrected pressure p as p* and return to step 2.

Step 8: Repeat Steps 2 to 7 until the solution has converged

Step 9: Repeat Steps 2 to 8 for the next time step

1.4.2.1 Dependent Variable Storage Considerations

Although the staggered grid storage arrangement, where the velocity components in each coordinate direction are stored at the cell-face, has been the most widely used technique for pressure based solution schemes, it has been recognized that the storage requirements for such schemes is very large. In SMARTFIRE, a collocated grid arrangement, where velocity components in each coordinate direction are stored at the cell-center, is used. The consequences of such an approach are:

(1) Huge reduction in storage requirements for geometrical related quantities, and
(2) The prediction of undesirable "checker-board" pressure fields.

To alleviate the prediction of the checker-board pressure fields, SMARTFIRE uses the Rhie and Chow technique to predict the flux at the cell-faces, where they are needed, by means of an algorithm that is free from the checker-board oscillations. Thus the face velocity depends on the pressure values prevailing at the cell centers of the neighboring cells (without using interpolation), and interpolated values of the other quantities used within the momentum equation. This approach is similar to the staggered approach, where the fluxes at the faces are identical for both the non-staggered and staggered approach.

1.4.3 Convergence and Relaxation Methods

Since the equations of fluid flow are coupled and non-linear, it is important that when using iterative solution techniques, as presented in the previous section, those relaxation techniques are used to control the solution. Relaxation techniques aid the solution to converge. Within

SMARTFIRE, there are three relaxation techniques available. These are the solver relaxation, linear relaxation and false time step relaxation techniques. Details of these techniques are presented in the next two sub-sections.

1.4.3.1 Linear relaxation

The linear relaxation technique allows the variation of a solved for variable, ϕ, in a linear fashion. The amount by which the variable is allowed to vary is controlled by the expression

$$\phi_{updated} = \alpha\phi_{calculated} + (1-\alpha)\phi_{old} \qquad (12a)$$

where α takes values between 0 and 1.7. The terms over relaxation and under relaxation are defined in the range of α as presented in the table below.

$\alpha < 1.0$	Under relaxation
$\alpha = 1.0$	No relaxation
$\alpha > 1.0$	Over relaxation

This technique can be applied for any variable that needs updating both within the solver and externally.

1.4.3.2 False time step relaxation

Using the concept of inertial relaxation, otherwise known as false time step relaxation, an equation of the form:

$$\phi_{updated} = \frac{\sum_{nb} a_{nb}\phi_{nb} + b + F_t\phi_{previous}}{(a_P + F_t)} \qquad (12c)$$

is produced, where F_t is termed the false time step, the units of which are the same as those of the a_P coefficient, i.e. kg/s. Thus the larger F_t is the stronger the relaxation. This technique is normally used for both steady and transient simulations.

1.4.4 Residual Calculation Methods

All solvers need termination criteria. In the case of SMARTFIRE, several options are available at several stages within the solution stage. Various forms are presented below.

1.4.4.1 Solver residual

This is the maximum error term evaluated from substituting the newest x solution vector into the system of algebraic linear equations and evaluating the maximum difference between the left and right hand sides of the equation. i.e.

$$\underline{r} = \underline{\underline{A}}\underline{x}_{p,new} - \underline{b} \qquad (12d)$$

The solver residual is used to determine if convergence has been reached and hence the solver inner iterations can cease.

1.4.4.2 Variable residual

The variable residual is a measure of the solution error between solution sweeps and is used to check for convergence for individual variables, using an expression of the form:

$$\phi_{p,new} - \phi_{old} = r \qquad (12e)$$

A variety of methods for calculating the variable residual are available as listed in the table below:

1	ABSOLUTE L1 NORM
2	ABSOLUTE L2 NORM
3	ABSOLUTE LI NORM
4	RELATIVE L1 NORM
5	RELATIVE L2 NORM
6	RELATIVE LI NORM
7	REFERENCE L1 NORM
8	REFERENCE L2 NORM
9	REFERENCE LI NORM

See SMARTFIRE MANUAL FILE SETUP GUIDE in the RESIDUAL METHODS section.

1.5 Boundary Conditions

To complete the definition of the fire problem it is necessary to specify a set of boundary and initial conditions. This sections deals with the boundary conditions that are available within SMARTFIRE. It is very important that the user specifies the boundary conditions correctly and also understands its impact within the numerical solution procedure. The section details the boundary conditions related to each equation solved in terms of the names as used within the SMARTFIRE User Interface.

1.5.1 Pressure Equation

1.5.1.1 Outlet or Free boundary

$$P = P_{external} \quad \text{or} \quad P = P_{fixed} \qquad (13a)$$

1.5.2 Momentum Equation

1.5.2.1 Inlet

Prescribe u, v, w as $\underline{V}$ (13b)

1.5.2.2 Outlets

Calculated u, v and w in internal cell (13c)

1.5.2.3 Walls

$$F_{shear} = area * \mu\rho \frac{1}{\Delta y_p} u \quad (13d)$$

1.5.2.4 Wall Functions

Using the widely used log-law for the wall, and the y^+ ($= \frac{\Delta y_p}{\nu}\sqrt{\frac{\tau_w}{\rho}}$) limits, the shear force is represented using the expression

$$F_{shear} = -\tau_w * area \quad = \quad -\mu \frac{u_p}{\Delta y_p} * area \quad (13e)$$

This expression is normally modified depending on the flow regime it is applied in. For turbulent flows

$$F_{shear} = -\rho C_\mu^{3/4} k_p^{*1/2} \frac{area}{u^+} \quad (13f)$$

1.5.2.5 Symmetry

$$\frac{\partial \varphi}{\partial n} = 0 \quad (13g)$$

1.5.3 Energy Equation

1.5.3.1 Fire as Enthalpy volumetric source

S_H = Fixed value (or) $S_H = (A+Bt+Ct^2+De^{Et})$ (13h)

Where A, B, C, D and E are user defined constants.

1.5.3.2 Walls

1.5.3.2.1 Adiabatic

$$\frac{\partial H}{\partial n} = 0 \qquad (13i)$$

1.5.3.2.2 Fixed Temperature (Dirichlet)

$$H = c_p T \qquad (13j)$$

1.5.3.2.3 Fixed Flux (Neumann)

$$\frac{\partial H}{\partial n} = fixed\,value \qquad (13k)$$

1.5.3.2.4 Flux / Temperature (Convective)

$$\frac{\partial H}{\partial n} = H_c(T_{amb} - T) \qquad (13l)$$

1.5.3.2.5 Flux / Temperature / Materials (Conductive)

As 1.5.3.2.3 BUT H_c (prescribed) is used (with wall material properties) to calculate an estimated wall T that is used instead of Tgas to then find the heat flux (13m)

1.5.3.2.6 Turbulent Wall Layer (Calculated Flux)

(1) Estimate a y + distance using turbulent wall layer function.
(2) Use y + value to calculate a H_c value based on material properties in near wall layer (13n)
(3) Use the calculated H_c value in the same way as 1.5.3.2.5

1.5.4 Turbulence Equations

1.5.4.1 Kinetic Energy

Wall: k is obtained by solving the discretised governing equation (13o)

1.5.4.2 Dissipation Rate

Wall: $$\varepsilon = \frac{0.1643}{K}\frac{k^{1.5}}{y} \tag{13p}$$

1.5.4.3 Log-Law

$$y^+ = \frac{1}{\kappa}\ln(Ey^+) \tag{13q}$$

1.5.5 Radiation Equation

1.5.5.1 Solid Surface

$$\left.\begin{aligned} I &= \varepsilon_w E_w + (1-\varepsilon_w)J \quad \textit{for lower surface} \\ J &= \varepsilon_w E_w + (1-\varepsilon_w)I \quad \textit{for higher surface} \end{aligned}\right\} \tag{13r}$$

Where E_w is calculated from T_w (which uses the following conditions)

$T_w = T_{gas}$ for adiabatic heat boundaries
$T_w = T_{solid}$ for solid material adjacency
$T_w = T_{calculated}$ for CONDUCTIVE or TURBULENT heat boundaries

1.5.5.2 Free Space

$$\left.\begin{aligned} I &= E_w \quad \textit{for lower surface} \\ J &= E_w \quad \textit{for higher surface} \end{aligned}\right\} \tag{13s}$$

1.5.5.3 Fixed Temperature

Same as 1.5.5.1 But E_w is calculated from a prescribed T_w as

$$E_w = \sigma T_w^{\,4} \tag{13t}$$

1.5.6 Concentration Equation

1.5.6.1 Fixed Value

$$\phi = value \tag{13u}$$

1.5.6.2 Fixed Flux (Neumann)

$$\frac{\partial \phi}{\partial n} = value \tag{13v}$$

1.5.6.3 Linear Flux

$$\frac{\partial \phi}{\partial n} = flux\_coeff * (\phi_{Ambient} - \phi_P) \qquad (13w)$$

1.5.6.4 Symmetry Plane

$$\frac{\partial \phi}{\partial n} = 0 \qquad (13x)$$

11.5 DATA DICTIONARY FOR CWNN++ GEOMETRY CLASSES.

DATA DICTIONARY FOR GEOMETRY CLASSES IN SMARTFIRE

GEOMETRY CLASSES

OVERVIEW OF THE GEOMETRY CLASSES

This data dictionary serves to describe the construction and usage of the classes that hold the geometric and related data within the "SMARTFIRE" system. It should be noted that the basic strategy and philosophy that has been employed is to minimise the complexity of the data access and to minimise the memory overheads whilst at the same time providing the greatest flexibility for future enhancement. This has led to the extensive use of data access functions that hide the internal data structure from the user and allow for optimised storage.

The following Entity Relationship Diagram (See Figure 1) details the logical relationships between the geometry items.

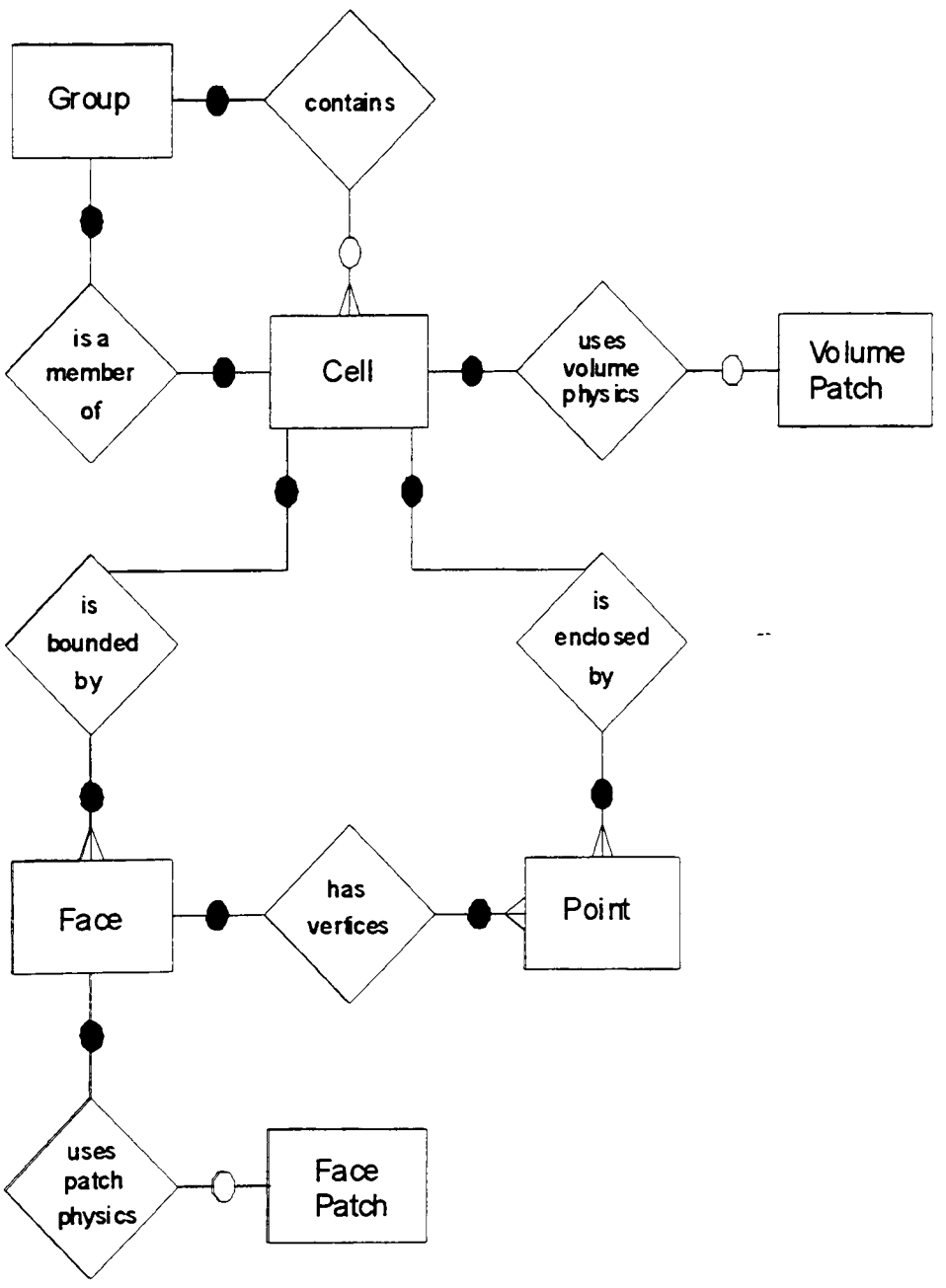


FIGURE 1 : Entity Relationship Diagram for the Smartfire geometry.

It should be noted that there are some redundancies in the logical ERD that have been introduced to meet known performance issues. For example it is clear that it is possible to determine which

group a particular cell belongs to by searching the groups for the cell in question. In practice this would be very inefficient and compute intensive. Consequently a reverse relationship link is introduced explicitly to provide the required information quickly.

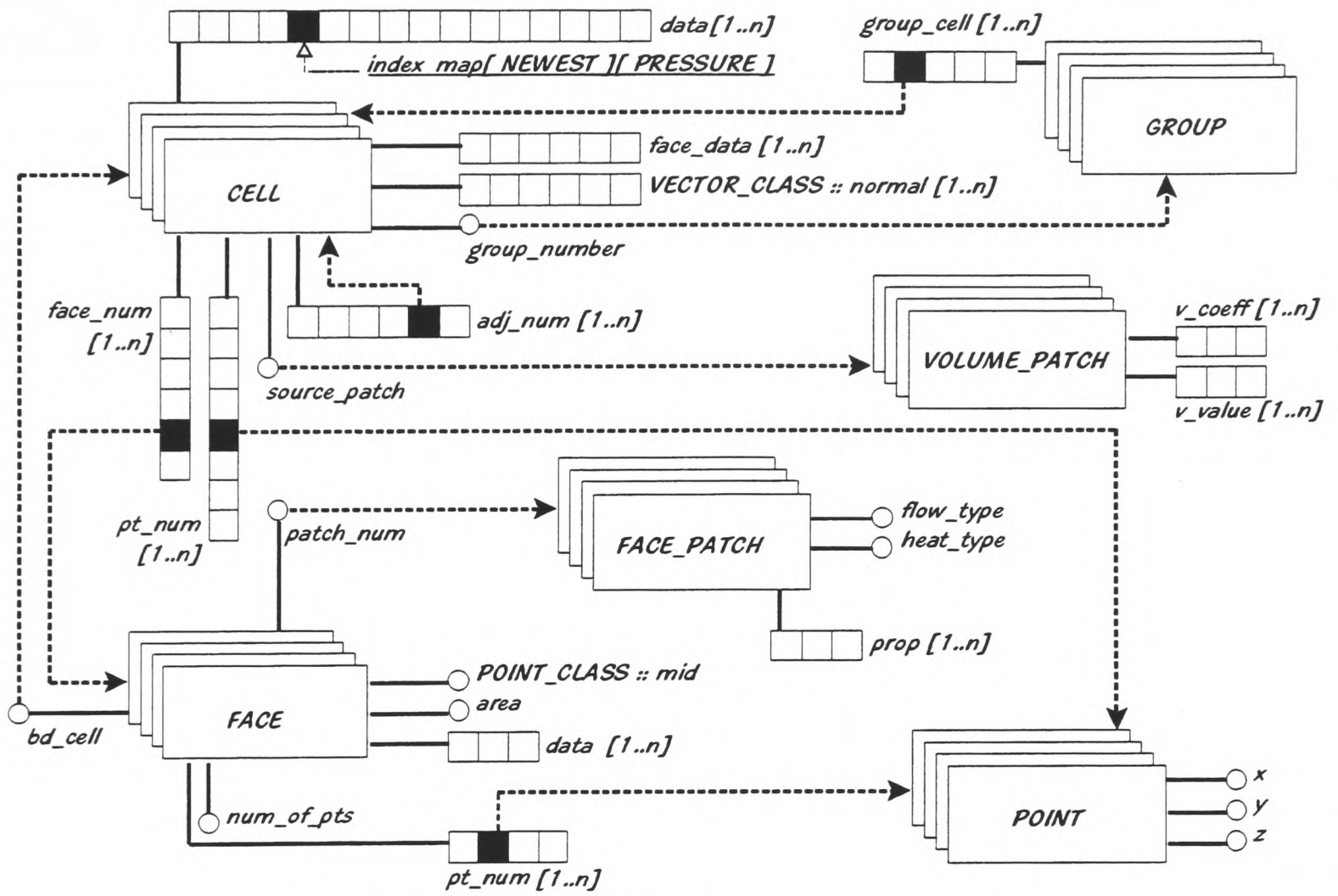


FIGURE 2 : The inter-relationships between geometry class objects in the physical implementation.

The figure (See Figure 2 above) shows some of the actual physical inter-relationships between the geometry class objects. It should be noted that the solid lines denote class data members. The dotted lines indicate implied or actual links to other objects. These links have generally been implemented as simple index values, rather than pointers, because of the variety of ways that the links are used and accessed by other objects. The figure is, by no means, complete because many of the cell class attributes would not fit on the limited size diagram.

The vector class does not appear on the geometry diagrams because it is little more than a utility class to be used for storage within more complex objects. The inclusion of the vector class, in this document, merely establishes the fact that vector mathematics have been used - where appropriate - to simplify the coding.

The number of elements in the arrays in the diagram, indicated by the [1..n] limits, are not indicative of the actual length of the array. The actual storage needs are determined by the type of the problem (number of variables to store) and the nature of the geometry itself.

It should be noted that most of the arrays that hold geometry data are declared as dynamic arrays of pointers to objects (i.e. G_Class ** g_object_array;). This is not as flexible as a linked list storage but does have the advantage of allowing individual elements of the array to be replaced with new members if required. It is also possible to extend the storage quite simply by copying the pointers into a temporary array, extending the original array and then copying back the pointers. This is clearly a much more efficient strategy than copying the objects themselves, particularly for cases where the geometry objects contain significant amounts of data as member variables.

Some of the class functions (and data members) described in this document are for internal class use and should not be used by class users. This is particularly true of the Group_Class where garbage collection routines are defined.

VECTOR CLASS

General Cartesian vector class with an array for the 3 components for the x, y and z co-ordinate directions (Access using "X", "Y" and "Z" constants defined in "constants.h"). This class is used to simplify numeric routines (using the overloaded operators that hide some of the looping) and storage allocation and access (in other classes). Generally this class is used to define data members of other classes.

class Vector_Class {

public:

//==>Data members for vectors
// (3 component array with storage 0=x, 1=y and 2=z). Use the X, Y and Z constants for access.
Float_Type comp[3];

// Methods

//==>Constructor for a vector object. Sets default vector (0.0, 0.0, 0.0).
Vector_Class(Float_Type, Float_Type, Float_Type);

//==>Return the magnitude of the vector = sqrt(comp[X]^2 + comp[Y]^2 + comp[Z]^2).
Float_Type magnitude(void);

//==> Return the sum of the components of the vector.
Float_Type sum(void);

//==> Negate all of the components of a vector.
void negate(void);

//==> Display the vector components.
void show(void);

//==> Like the constructor this sets the components of a vector object.
void set(Float_Type, Float_Type, Float_Type);

//==> Overloaded assignment for vectors (copy).
Vector_Class& operator=(Vector_Class);

//==> Overloaded addition-assignment for vectors.
Vector_Class& operator+=(Vector_Class);

//==> Overloaded addition-assignment for vectors when adding a float.
Vector_Class& operator+=(Float_Type);

```
//==>  Overloaded subtraction-assignment for vectors.
                Vector_Class& operator-=( Vector_Class );

//==>  Overloaded subtraction-assignment for vectors when taking a float.
                Vector_Class& operator-=( Float_Type );

//==>  Overloaded multiplication-assignment for vectors.
                Vector_Class& operator*=( Vector_Class );

//==>  Overloaded multiplication-assignment for vectors when multiplying by a float.
                Vector_Class& operator*=( Float_Type );

//==>  Overloaded division-assignment for vectors.
                Vector_Class& operator/=( Vector_Class );

//==>  Overloaded division-assignment for vectors when dividing by a float.
                Vector_Class& operator/=( Float_Type );

//==>  Overloaded multiplication for vectors.
                Vector_Class operator*( Vector_Class );

//==>  Overloaded division for vectors divided by floats.
                Vector_Class operator/( Float_Type );

//==>  Overloaded division for vectors.
                Vector_Class operator/( Vector_Class );

//==>  Overloaded addition for vectors.
                Vector_Class operator+( Vector_Class );

//==>  Overloaded subtraction for vectors.
                Vector_Class operator-( Vector_Class );

//==>  Returns the vector that has components as magnitude of source vector.
                Vector_Class abs( void );

//==>  Overloaded multiplication for vectors with floats.
                Vector_Class operator*( Vector_Class, Float_Type );

};
```

POINT CLASS

Physical coordinate point class in 3 space dimensions. This class is used to simplify numerical routines and to tidy up storage in other objects. All of the mesh nodes (cell vertices) are allocated as an array of pointers to points as follows:

```
Point_Class ** point;
point = new Point_Class* [TOTAL_NUMBER_OF_POINTS+1];
for ( i=1; i <= TOTAL_NUMBER_OF_POINTS; i++ ){
   point[i] = new Point_Class( 0.0, 0.0, 0.0 );
}
```

// Example of class member access

// Setting point number 200 to have coordinates (x=0.5, y=1.3, z=2.7)

```
point[200]->set( 0.5, 1.3, 2.7 );
```

// Getting the displacement vector between point 25 and point 26

```
Vector_Class displace = (*point[26]) - (*point[25]);
```

class Point_Class {

public:

//==> Coordinate values for x, y and z.
Float_Type x, y, z;

// Methods

//==> Point constructor creates a default co-ordinate at (0.0, 0.0, 0.0).
Point_Class(Float_Type, Float_Type, Float_Type);

//==> Show the coordinates of a point.
void show(void);

//==> Set the coordinates of a point.
void set(Float_Type, Float_Type, Float_Type);

//==> Overloaded operator for point assignment (copy).
Point_Class& operator=(Point_Class);

//==> Overloaded addition of points (adds co-ordinates).
Point_Class operator+(Point_Class);

```
//==>  Overloaded addition-assignment for points (add and copy result).
                Point_Class& operator+=( Point_Class );

//==>  Overloaded division for points by a float (reduction).
                Point_Class operator/( Float_Type );

//==>  Overloaded division-assignment for points (subtract scalar and copy result).
                Point_Class& operator/=( Float_Type );

//==>  Overloaded multiplication-assignment for points (times by scalar and copy result).
                Point_Class& operator*=( Float_Type );

//==>  Calculate vector displacement between two points.
                Vector_Class operator-( Point_Class );

//==>  Overloaded multiplication for points and floats.
                friend Point_Class operator*( Point_Class, Float_Type );

};
```

FACE CLASS

Faces are objects in their own right. E.g. a single brick cell has 6 faces whereas two adjoining brick cells have only 11 individual faces because the common face exists only once in a separate global face array. This global face array is defined as an array of pointers to faces.

```
Face_Class ** face;
face = new Face_Class* [TOTAL_NUMBER_OF_FACES+1];
for ( i=1; i <= TOTAL_NUMBER_OF_FACES; i++ ){
   face[i] = new Face_Class( 4, 2 );
}
```

// Example of class member access

// Calculating the centre of face number 25

```
face[25]->calc_centre();
```

// Accessing some face data (Not used in current version of the code)

```
face[30]->access(WALL_SHEAR_STRESS) = 0.3;
```

class Face_Class {

private:

//==> The size of storage used (only used for debug mode bounds checking).
int array_size;

//==> The data array slots for storage of face properties.
Float_Type * data;

public:

//==> The number of points around a face.
int num_of_pts;

//==> The patch number associated with a face (or 0 if this face is internal).
Int_Type patch_num;

//==> The boundary cell indicator (0 if internal or non-zero is the cell number).
Int_Type bd_cell;

//==> Array of indices of face vertex points in a walk around the face.
Int_Type * pt_num;

```
//==>  The mid point of the face.
                Point_Class mid;

//==>  The scalar area of the face.
                Float_Type area;

// Methods

//==>  Constructor needs to know the number of face points and number of face properties.
                Face_Class( int, int );

//==>  Destructor.
                ~Face_Class( void );

//==>  Calculates the face centroid and puts the result in mid.
                void calc_centre( void );

//==>  Data access function gives back a reference to the requested data item.
                Float_Type & access( Var_Index_Type );

};
```

CELL CLASS

Cell objects are face bounded control volumes with their own collection of internal properties. Many of the properties used in the computation are cell centred however a few exist as cell-face properties where there will be a value (or item) for each of the faces in a cell. When using cell class access functions the local face indices are used rather than the global face numbering scheme. This means that internally the cell knows about faces 1 to n (where n is the maximum number of faces bounding this cell). The actual global indexed face numbers bounding this cell could be literally anything. The interrogation functions hide the index re-mapping to simplify access from the cell to the bounding faces. All of the separate cell-face arrays have the same indexing so that using cell-face index #1 inside a cell will refer to all of the cell-face properties of only one of the bounding faces. The cells are stored in an array of pointers to cells.

```
Cell_Class ** cell;
cell = new Cell_Class* [TOTAL_NUMBER_OF_CELLS+1];
for ( i=1; i <= TOTAL_NUMBER_OF_CELLS; i++ ){
   cell[i] = new Cell_Class( i, 8, 6, 23, 5, 2 );
}
```

// Example of class member access

// Setting the newest value of pressure in cell number 7

```
cell[7]->access(NEWEST,PRESSURE) = 2.5e-5;
```

// Getting the 2nd internal face area of cell number 15

```
float area = cell[15]->face_area( 2 );
```

// Finding the maximum Y coordinate of cell 53

```
float y_max = cell[53]->find_max_coord(Y);
```

class Cell_Class {

//==> The following private data members cannot be accessed directly. You must use the data access functions provided later. Some of these data access functions are declared as returning a reference type (&) in which case they can be used on either side of an assignment satement. See the above examples for details of using the data access function. This is necessary to hide the complexities of least memory array storage.

private:

//==> Array of data values for storing cell properties.
Float_Type * data;

//==> The number of face variables stored in a cell.
int face_vars;

//==> Array of face value data accessed by function below.
Float_Type ** face_data;

//==> Array of differentials stored at the cell centre for all solved vars.
Vector_Class * diff_data;

public:

//==> The number of vertex points of a cell.
int num_of_pts;

//==> The number of faces forming the boundary of a cell.
int num_faces;

//==> This cells number in the global index scheme.
Int_Type this_cell_num;

//==> The material property index for this cell.
Int_Type material;

//==> The group number which this cell belongs to.
Int_Type group_number;

//==> The volume source patch index for coefficient calculation.
Int_Type source_patch;

//==> Array of point indices for the vertex points which make up this cell.
Int_Type * pt_num;

//==> Array of face indices which make up this cell.
Int_Type * face_num;

//==> Array of adjacent cell indices (one for each cell face) (0 if boundary).
Int_Type * adj_num;

//==> The local face index in the adjacent cell that adjoins this cell.
Int_Type * cell_adj_to;

//==> The array of distances from the cell centroid to the middle of the cell faces.
Float_Type * dist_to_face;

//==> The normal (perpendicular) distance to the cell faces.
Float_Type * norm_dist_to_face;

//==> The cell centroid point.
Point_Class mid;

//==> Array of outward pointing normal vectors (for each cell face).
Vector_Class * normal;

//==> Array of convection fluxes through each face.
Float_Type * convection;

//==> The in-face displacement vector between the centre of the face and the intersection point where a line joining the cell centres passes through the face. (for each cell face).
Vector_Class * displacement;

// Methods

//==> Constructor for a cell. The arguments are used to allocate memory for internal cell storage. Arguments are as follows:-
The global cell number index of this cell,
The number of vertices of this cell,
The number of faces bounding this cell,
The total number of all property values to allocate for,
The number of solved variables to allocate for AND
The number of cell-face properties to allocate for.
Cell_Class(Int_Type, int, int, int, int, int);

//==> Destructor.
~Cell_Class(void);

//==> Internal error management routine when allocation fails.
void allocation_failure(char*);

//==> Copy the NEWEST variable value to the LAST storage slot for the same variable.
void copy_current_to_last(Var_Index_Type);

//==> Copy the NEWEST variable value to the OLD storage slot for the same variable.
void copy_current_to_old(Var_Index_Type);

//==> Copy the NEWEST value of a variable to another NEWEST variable slot.
void copy_data(Var_Index_Type, Var_Index_Type);

//==> Locate the indices of all of the vertices of this cell. List into pt_num.
Logical find_cell_pts(void);

//==> Check if a test point (argument) is within this cell volume.
Logical point_within_cell(Point_Class);

//==> Find if any of the cell faces meet a domain boundary and set internal properties accordingly.
void find_boundaries(void);

//==> Calculate the position of the cell centroid. Result into mid.
void calc_centre(void);

//==> Calculate the outward cell-face normals. Results into normal array.
void calc_normals(void);

//==> Calculate the current cell volume (N.B. volume is a property in the data array).
void calc_volume(void);

//==> Calculate all of the distances to the cell-faces. Results into dist_to_face array.
void calc_dist_to_face(int);

//==> Calculate the normal distances to the cell-faces. Results into norm_dist_to_face array.
void calc_norm_dist_to_face(const);

//==> Interpolates across a face for a particular variable and mode (E.g. interpolate(1,NEWEST,PRESSURE) would return the interpolated pressure across local cell-face 1 for the NEWEST PRESSURE property).
Float_Type interpolate(int, Mode_Type, Var_Index_Type);

//==> Interpolates the differentials for a variable.
Vector_Class interpolate_diffs(int, Var_Index_Type);

//==> Similar to interpolate except a simple 50 - 50 averaging scheme is used.
Float_Type average(int, Mode_Type, Var_Index_Type);

//==> Similar to interpolate_diffs except that a simple averaging scheme is used.
Vector_Class average_diffs(int, Var_Index_Type);

//==> Calculates the displacement vector for each face. Results into displacement array.
void calc_disp_vectors(void);

//==> Calculates the updated material property value for this cell. The result will be put into the correct slot in the data array.
void update_property(Var_Index_Type, Update_Mode);

//==> Return the maximum coordinate for this cell for a particular direction.
Float_Type find_max_coord(int);

//==> Return the minimum coordinate for this cell for a particular direction.
Float_Type find_min_coord(int);

//==> Calculate the flow-upwinded density across a particular cell-face.
Float_Type upwind_density(int);

//==> Calculates a vector velocity at the selected cell-face based on the velocity components in the centre this cell and the centre of the adjacent cell.
void calc_face_velocity(Vector_Class&, int);

//==> Calculates a correction vector based on non-orthogonal meshes.
void calc_conjunction_vector(Vector_Class&, int);

//==> Calculates the temperature for a cell given a knowledge of the material properties and the current enthalpy value. Result into the correct slot in the data array.
void calc_temperature(void);

//==> Calculates the turbulent viscosity value. Result into the data array.
void calc_turb_viscosity(void);

//==> Calculates the buoyancy term from the properties and temperature. Result is put into the correct slot in the data array.
void calc_buoyancy(void);

//==> Calculates the velocity magnitude for combined U_VELOCITY, V_VELOCITY, W_VELOCITY for a flow calculation.
Float_Type calc_velocity_magnitude(Mode_Type);

//==> Calculates the error in the sum of mass fluxes through the faces of this cell.
Float_Type calc_continuity_error(void);

//==> Interrogates the appropriate cell-face to find the area.
Float_Type & face_area(int);

//==> Interrogates the appropriate cell-face to find the face patch number (0 if internal).
Int_Type & face_patch_id(int);

//==> Numerical vector product routine consistently used in computations.
void add_differential_product(Var_Index_Type, int, Vector_Class);

//==> Calculates the dissipation rate differentials in this cell. Result into diff_data array.
void calc_dissipation_diffs(Mode_Type);

//==> Calculates the kinetic energy differentials in this cell. Result into diff_data array.
void calc_kinetic_diffs(Mode_Type);

//==> Calculates the enthalpy differentials in this cell. Result into diff_data array.
void calc_enthalpy_diffs(Mode_Type);

```
//==>  Calculates the generic var differentials in this cell. Result into diff_data array.
             void calc_generic_diffs( Mode_Type );

//==>  Calculates the velocity differentials in this cell. Result into diff_data array.
             void calc_velocity_diffs( Mode_Type );

//==>  Calculates the corrections to the velocities based on pressure changes. Results into the
     correct slots in the data array.
             Vector_Class calc_velocity_corrections( void );

//==>  Calculates the wall friction terms for a particular cell-face. Results are stored in the
     correct slot in the face_data array.
             void calc_friction_terms( int );

//==>  Calculates the turbulent generation rate for this cell. Result into the data array.
             void calc_turb_generation( void );

//==>  Calculates the pressure gradients in this cell. Results into the data array.
             void calc_pressure_grads( void );

//==>  Calculates cell-face convection. Results into the convection array.
             void calc_convection( void );

//==>  Adds the volume source contribution to the return float value for a particular variable.
             void add_volume_source( Var_Index_Type, Float_Type& );

//==>  The user data access function that refers to the items in the data array.
             Float_Type & access( Mode_Type, Var_Index_Type );

//==>  The user face data access function that refers to the items in the face_data array.
             Float_Type & access_face( int, Var_Index_Type );

//==>  Returns the effective in-cell viscosity for both laminar and turbulent cases.
             Float_Type effective_viscosity( Mode_Type );

//==>  The user differentials access function that refers to the diff_data array.
             Vector_Class & diffs( Var_Index_Type );

};
```

FACE PATCH CLASS

The boundary face patch class is used to collect and store data about a boundary to the domain. These properties are applied only to faces. The various properties indicate to the CFD code how the boundary properties should be used in the computation of new values.

```
class Face_Patch_Class {

        public:

//==>   The flow type indicator (WALL, WALL2, INLET, OUTLET or SYMMETRY).
                int flow_type;

//==>   The heat type indicator (DIRICHLET, NEUMAN, CONVECTIVE, ADIABATIC).
                int heat_type;

//==>   Extra variables boundary type indicator (0=symmetry 1=fixed value 2=fixed flux 3=linear
      flux).
                int * bound_type;

//==>   The property array that stores the quantities associated with the above conditions.
                Float_Type * prop;

// Methods

//==>   Constructor needs to know the maximum number of property values to be stored.
                Face_Patch_Class( int );

//==>   The destructor.
                ~Face_Patch_Class( void );

};
```

VOLUME PATCH CLASS

The volumetric source class stores data about sources that are applied within cells. These sources are applied to solved variables to indicate an influx (or decrease) of a particular property from the domain.

class Volume_Patch_Class {

private:

//==> The array of coefficient values (unused at present).
Float_Type * v_coeff;

//==> The array of volume source values (one for each solved variable).
Float_Type * v_value;

//==> Parameter used in time varying sources.
Float_Type * v_ramp_parameter;

//==> Indicator that this variable has a ramped source.
Logical * v_ramped;

//==> Indicator that a particular variable has a volume source available.
Logical * v_stored;

public:

//==> Constructor needs to know the number of solved variables.
Volume_Patch_Class(int);

//==> The destructor.
~Volume_Patch_Class(void);

//==> Interrogation function refers to the coefficient values.
Float_Type & coeff(Var_Index_Type);

//==> Interrogation function refers to the volume source values.
Float_Type & value(Var_Index_Type);

//==> Interrogation function refers to the storage indicator.
Logical & stored(Var_Index_Type);

//==> Interrogation function refers to the ramped source indicator.
Logical & ramped(Var_Index_Type);

```
//==>  Interrogation function refers to the ramp parameter.
            Float_Type & ramp_parameter( Var_Index_Type );

};
```

GROUP CLASS

The group class objects maintain a list of cell indices that constitute a special grouping that should have different solution techniques applied. The range of different conditions that can be applied is limited at present however these can still be useful in situations where the prevailing conditions within a group are radically different from another group.

class Group_Class {

public:

//==> The global number of this group.
Int_Type group_number;

//==> The total number of cells in this group.
Int_Type num_cells_in_group;

//==> An array of indices of all of the cells in this group.
Int_Type * group_cell;

// Methods

//==> Constructor needs to know the number of this group and the number of cells initially in the group.
Group_Class(Int_Type, Int_Type);

//==> The destructor.
~Group_Class(void);

//==> An internal function used to garbage collect in a modified group.
void compact_list(void);

//==> An internal function used to add more storage to a group.
void extend_storage(Int_Type);

//==> An internal garbage collection routine.
void contract_storage(Int_Type);

//==> A function to search for a global cell index and report its presence in this group.
Logical find_index(Int_Type, Int_Type&);

//==> Internal function adds the cell index to this group.
void add_cell(Int_Type);

//==> Internal function sets the group cell item to be a particular cell index.
void set_cell(Int_Type, Int_Type);

//==> Reports on the current contents of this group.
void show_cells(void);

//==> Function to move a cell from another group to this one (All memory allocation and garbage collection is handled internally).
void move_group_cell(Group_Class&, Int_Type);

// Access routines to data stored in the variables.

//==> Sets the number of group iterations for a particular variable.
void set_solver_group_iterations(Var_Index_Type, Int_Type);

//==> Sets the relaxation parameter for a particular variable.
void set_solver_group_relaxation(Var_Index_Type, Float_Type);

//==> Sets the residual value for a particular variable.
void set_solver_group_residual(Var_Index_Type, Float_Type);

//==> Sets the new group solver convergence tolerance for a particular variable.
void set_solver_group_tolerance(Var_Index_Type, Float_Type);

//==> Returns the current number of iterations for a particular variable.
Int_Type get_solver_group_iterations(Var_Index_Type);

//==> Returns the current relaxation parameter for a particular variable.
Float_Type get_solver_group_relaxation(Var_Index_Type);

//==> Returns the current solver residual for a particular variable.
Float_Type get_solver_group_residual(Var_Index_Type);

//==> Returns the current solver tolerance for a particular variable.
Float_Type get_solver_group_tolerance(Var_Index_Type);

};

11.6 SMARTFIRE : Interim development report on the Control and Blackboard Architecture used in the SMARTFIRE system.

SMARTFIRE

Interim development report on the Control Architecture and Blackboard used in the SMARTFIRE system

PURPOSE OF DOCUMENT:

- Information about the newly implemented CFD blackboard architecture.

- Information about access to data on the CFD blackboard.

- Information about the control layer of SMARTFIRE.

- List of data and control items available via the blackboard.

Points to note about usage of the blackboard:

- Most of the blackboard data and control items are accessed directly. Ideally data access should be functional with appropriate get and set functions that would allow all of the data to be private. This is a throwback to the nature of the original legacy code where the control and status flags were all, essentially, public.

- Because of the complexities of group solvers the blackboard DOES NOT maintain copies of the group information, which would be very difficult to maintain. The groups also have the potential to be very large and this would lead to inefficient memory usage in certain cases. Access functions have been provided that "punch" through the blackboard to the relevant data structures in SMARTFIRE. This is a temporary measure to allow research even as the group capabilities are being developed.

- The variable class and solved variable class structures are not easily duplicated in the blackboard. It has been decided to use an alternative approach using simple arrays to store the control and status information for individual variables. Accessing the data is correspondingly more awkward but the implementation is much easier. See the examples for details about accessing variable or solver information.

- Many of the control flags are linked to other control data. For example the "use_transient" logical flag is closely linked to the values of "delta_time" and "num_of_time_steps". Consequently changes to some blackboard items cannot be effected in isolation or unpredictable behaviour will result. This is one of the reasons why functional access is so desirable.

- Control must not be used to switch from a less complex state to a more complex one. For example the CFD code can not switch from processing as a steady state run to a transient run if it started out as a steady state run. This is because of memory allocation and initialisation optimisation that are tailored to the type of problem being run. Conversely it is generally and theoretically possible to switch from a more complex to a less complex processing state.

- Similarly to above point, the code cannot create variables at run time while it is processing. If extra variables (e.g. turbulent quantities) are needed at a later stage of the simulation then they must have been known when the run started (although - once created - they can be disabled until required).

Current blackboard architecture:

```
class CFD_Blackboard_Class {

   public :

//
// Constructor and Destructor.
//

      CFD_Blackboard_Class( void );
      ~CFD_Blackboard_Class( void );

//
// Functions to update the Blackboard or the CFD code control structures.
//

      update_cfd_to_blackboard( void );
      update_blackboard_to_cfd( void );

//
// Temporary "punch-through" code to allow groups to be examined and changed.
//

      void set_group_solver_iterations( const Int_Type group_num,
              const Var_Index_Type& var_point, const Int_Type num_iters );
      void set_group_solver_relaxation( const Int_Type group_num,
              const Var_Index_Type& var_point, const Float_Type relax_value );
      void set_group_solver_tolerance( const Int_Type group_num,
              const Var_Index_Type& var_point, const Float_Type tol_value );
      Int_Type   get_group_solver_iterations( const Int_Type group_num,
                    const Var_Index_Type& var_point );
      Float_Type get_group_solver_relaxation( const Int_Type group_num,
                    const Var_Index_Type& var_point );
      Float_Type get_group_solver_tolerance( const Int_Type group_num,
                    const Var_Index_Type& var_point );
      Float_Type get_group_solver_residual( const Int_Type group_num,
                    const Var_Index_Type& var_point );

//
// Copies of the domain, control and status objects from the CFD code.
//

      Domain_Class        bb_domain;
      Control_Class       bb_control;
      Status_Class        bb_status;

//
// Solved variable control and status arrays.
//

      Int_Type            * bb_solver_max_iterations;
      Int_Type            * bb_solver_max_inner_iterations;
      Int_Type            * bb_solver_max_group_iterations;
      Solver_Class_Type   * bb_solver_class;
      Solver_Type         * bb_solver_type;
      Float_Type          * bb_solver_relaxation;
      Float_Type          * bb_solver_false_time_step;
      Float_Type          * bb_solver_residual;
      Float_Type          * bb_solver_solution_error;
      Boolean             * bb_solver_use_false_time;
      Boolean             * bb_solver_use_solver_relax;
      Boolean             * bb_solver_used;
      Boolean             * bb_solver_use_value_solver;
```

```
//
// Calculated variable control and status arrays
//

      Res_Mode_Type       * bb_variable_residual_mode;
      Res_Method_Type     * bb_variable_residual_method;
      Float_Type          * bb_variable_tolerance;
      Float_Type          * bb_variable_relaxation;
      Float_Type          * bb_variable_residual;
      Float_Type          * bb_variable_monitor_value;
      Boolean             * bb_variable_source_available;
      Boolean             * bb_variable_use_linear_relax;
      Boolean             * bb_variable_used;

};


// cfd_blackboard MUST be a pointer to a blackboard object and the
// blackboard object MUST be "NEWed" after the problem set-up but
// prior to the first process_step. This is so that the sizes used
// for dynamic memory allocation (in the blackboard) are correct for
// the problem being run. The BB object should also be deleted before
// the program exits or returns.

LOCATION_OF_STORAGE CFD_Blackboard_Class    * cfd_blackboard;
```

Examples of access to the "cfd_blackboard":

(1) Setting the number of iteration sweeps to 100 for subsequent processing:

```
cfd_blackboard->bb_control.num_of_sweeps = 100;
```

(2) Getting the number of groups:

```
num_groups_used = cfd_blackboard->bb_domain.num_of_groups;
```

(3) Getting the mass error for a flow simulation:

```
if ( cfd_blackboard->bb_control.use_flow ){
   current_mass_error = cfd_blackboard->bb_status.mass_error;
}
```

(4) Putting a new relaxation into the U_VELOCITY solver:

```
cfd_blackboard->bb_solver_relaxation[SOL[U_VELOCITY]] = new_relax_value;
```

(5) Putting a new value of convergence tolerance into the PRESSURE:

```
cfd_blackboard->bb_variable_tolerance[VAR[PRESSURE]] = new_tolerance_value;
```

(6) Access to the group solver settings:

```
old_tolerance = cfd_blackboard->get_group_solver_tolerance( in_group, U_VELOCITY );
cfd_blackboard->set_group_solver_tolerance( in_group, U_VELOCITY, new_tolerance );
old_relaxation = cfd_blackboard->get_group_solver_relaxation( in_group, ENTHALPY );
cfd_blackboard->set_group_solver_tolerance( in_group, ENTHALPY, new_relaxation );
current_residual = cfd_blackboard->get_group_solver_residual( in_group, U_VELOCITY );
```

CFD code "high-level" code architecture in control layer:

```
// Other initialisation routines.
cfd_code_link( INITIALISE_CFD, process_status );
cfd_blackboard = new CFD_Blackboard_Class;
cfd_blackboard->update_cfd_to_blackboard();
// KBS call if required.
*--------]
|
|   // Implicit while loop ( actually this is implemented as repetitive
|   // calls from a dedicated event timer handler in the GUI code layer ).
|
|   while ( process_status == PROCESSING_CFD ){
|
|      *--------]
|       |
|       |   // Task management can be performed in this block.
|       |
|       |   do_pre_sweep_kbs_reasoning();   // KBS call before sweep.
|       |
|       |   cfd_blackboard->update_blackboard_to_cfd();
|       |
|       |   cfd_code_link( PROCESS_CFD, process_status );
|       |
|       |   cfd_blackboard->update_cfd_to_blackboard();
|       |
|       |   do_post_sweep_kbs_reasoning();  // KBS call after sweep.
|       |
|       *--------]
|
|   }
|
*--------]
// KBS call if required.
cfd_code_link( TERMINATE_CFD, process_status );
delete cfd_blackboard;
// Other tidy-up routines.
```

Control and status information in the DOMAIN, STATUS and CONTROL objects:

```
//
// Domain of problem class
//

class Domain_Class {

public :

   int         dimensions;             // Dimensions used ( always 3 in cwnn++ )
   Int_Type    num_of_cells;           // Number of cells in the domain
   Int_Type    num_of_points;          // Number of grid points in the domain
   Int_Type    num_of_faces;           // Number of faces in the domain
   Int_Type    num_of_groups;          // Number of cell groups
   int         max_cell_faces;         // Maximum number of faces in a cell
   int         max_cell_points;        // Maximum number of points in a cell
   int         max_face_points;        // Maximum number of points in a face
   int         max_band_width;         // Maximum width of sparse solver matrix storage
   int         max_face_values;        // Maximum number of face property values
   int         max_cell_face_vars;     // Maximum number of cell face variables
   int         max_cell_values;        // Maximum number of "slots" for cell values
   int         max_stored_variables;   // Max number of stored cell variables
   int         num_of_solved_vars;     // Number of solved variables
   int         num_of_calc_vars;       // Number of calculated variables
   Int_Type    num_of_properties;      // Number of different property variables
   Int_Type    num_of_face_patches;    // Number of different boundary patches specified
   Int_Type    num_of_volume_patches;  // Number of different volume patches specified
   Int_Type    num_of_materials;       // Number of different materials specified
   Int_Type    nx_cells;               // Cartesian domain cell extent in x direction
   Int_Type    ny_cells;               // Cartesian domain cell extent in y direction
   Int_Type    nz_cells;               // Cartesian domain cell extent in z direction
   Float_Type  reference_temperature;  // Reference temperature if specified
   Float_Type  reference_density;      // Reference density if specified
   Float_Type  reference_pressure;     // Reference pressure if specified
   Int_Type    pressure_ref_cell;      // Cell number for cell pressure reference
   Float_Type  external_pressure;      // The actual external pressure
   Float_Type  gravity;                // Value of gravity (-)ve is downwards
   Boolean     is_cartesian;           // Flag to indicate cartesian like domain
   Boolean     is_structured_mesh;     // Flag to indicate structured mesh

};


//
// Status of problem class
//

class Status_Class {

public :

   Int_Type    iteration_count;        // Current outer iteration sweep number
   Int_Type    last_saved_iteration;   // Last iteration number saved into restart database
   Int_Type    time_step;              // Current transient time step number (transient only)
   Float_Type  sim_time;               // Total simulation time for transient runs
   Float_Type  mass_error;             // Maximum mass error for this sweep
   Float_Type  elapsed_time;           // Total elapsed c.p.u. time
   Boolean     doing_restart;          // Flag to indicate the current run is a restart
   Boolean     memory_is_allocated;    // Status of memory allocation
   int         geom_version;           // Version of geometry specification file

};
```

```
//
// Control of problem class
//

class Control_Class {

private :

   Boolean           debug_item[ MAX_VARIABLES+1 ];

public :

   Float_Type        delta_time;              // Time step size for transient runs
   Diff_Scheme_Type  difference_scheme;       // Difference scheme to use
                                              // {i.e. UPWIND, HYBRID, POWER_LAW, EXPONENTIAL}
   Int_Type          num_of_time_steps;       // Number of time steps to use in transient run
   Int_Type          turbulence_model;        // Turbulence model to use (KE_MODEL only)
   Int_Type          probe_cell_number;       // Cell to look in for monitor information
   Int_Type          num_of_sweeps;           // Number of sweeps (outer iterations) to do
   Int_Type          num_of_flow_sweeps;      // Number of flow algorithm sweeps to do
   Int_Type          ke_source_method;        // Kinetic energy source method
   Int_Type          debug_start_iteration;   // Debug switched to a range of sweeps only
   Int_Type          debug_end_iteration;
   Int_Type          debug_start_time_step;   // Debug switched to a range of time steps only
   Int_Type          debug_end_time_step;
   Int_Type          debug_start_cell;        // Debug cell range
   Int_Type          debug_end_cell;
   Int_Type          print_frequency;         // Frequency of run time sweep info
   int               sweep_adapt_every;       // Mesh adaption and refinement parameters
   int               sweep_refine_every;
   Float_Type        tolerance;               // Global default convergence tolerance value
   Boolean           be_silent;               // Do not use the screen for output
   Boolean           use_heat;                // Activate the heat solver
   Boolean           use_flow;                // Activate the flow solvers
   Boolean           use_boussinesq;          // Switch to select buoyancy calculation method
   Boolean           use_solidification;      // Redundant switch to allow for solidification
   Boolean           use_cross_product;       // Activate cross-product diffusion terms
   Boolean           use_turbulence;          // Activate turbulence (solves turbulent vars)
   Boolean           use_output_var;          // Create a var result file
   Boolean           use_output_phi;          // Create a PHI result file
   Boolean           use_flowvis_phi;         // Use Flowvis format for PHI file
   Boolean           use_restart;             // Read restart info and continue processing
   Boolean           use_transient;           // Transient simulation (steady state otherwise)
   Boolean           use_wall_2;              // A boundary for PHOENICS compatibility
   Boolean           use_pressure_ref_point;  // Switch to use a cell for pressure reference
   Boolean           use_pressure_ref;        // Switch to use a pressure reference
   Boolean           use_density_ref;         // Switch to use a density reference
   Boolean           use_temperature_ref;     // Switch to use a temperature reference
   Boolean           use_external_pressure;   // Switch to use an external pressure value
   Boolean           create_restart;          // Create a restart file at the end of this run
   Boolean           use_refinement;          // Switch to allow mesh refinement (NA)
   Boolean           use_adaption;            // Switch to allow mesh adaption (NA)
   Boolean           use_volumetric_sources;  // Flag to indicate use of volume sources
   Boolean           use_debug_dump;          // Flag to write debug data to file
   Boolean           use_log_file;            // Keep a log file if active
   Boolean           use_restart_dumps;       // Flag to allow creation of restarts each sweep
   Boolean           use_table_file;          // Residual and monitor output to a graph file
   Boolean           started_table_file;
   int               restart_dump_interval;   // Frequency of debugs (1=every sweep)
   Algorithm_Type    flow_algorithm;          // Switch for flow algorithm (SIMPLE only)
   Boolean           show_storage_info;       // Switch for storage information debugging
   Boolean           check_mem_allocation;    // More exhaustive memory allocation information

   inline Boolean & use_debug( const Var_Index_Type& var_point ){
      return debug_item[ var_point ];
   }

};
```