School of Mathematical Sciences
Queensland University of Technology

Modelling Sea Water Intrusion in Coastal Aquifers Using Heterogeneous Computing

Benjamin Cumming
Bachelor of Applied Science (Mathematics)
Bachelor of Applied Science (Hons I)
Masters of Applied Science (Mathematics)

A thesis submitted for the degree of Doctor of Philosophy in the Faculty of Science and Technology, Queensland University of Technology according to QUT requirements.

Principal Supervisor:	Professor Ian Turner
Associate Supervisors:	Dr Timothy Moroney
	Associate Professor Malcom Cox
	Associate Professor Les Dawes
	Professor Vo Anh

2012

Abstract

The objective of this PhD research program is to investigate numerical methods for simulating variably-saturated flow and sea water intrusion in coastal aquifers in a high-performance computing environment. The work is divided into three overlapping tasks: to develop an accurate and stable finite volume discretisation and numerical solution strategy for the variably-saturated flow and salt transport equations; to implement the chosen approach in a high performance computing environment that may have multiple GPUs or CPU cores; and to verify and test the implementation.

The geological description of aquifers is often complex, with porous materials possessing highly variable properties, that are best described using unstructured meshes. The finite volume method is a popular method for the solution of the conservation laws that describe sea water intrusion, and is well-suited to unstructured meshes. In this work we apply a control volume-finite element (CV-FE) method to an extension of a recently proposed formulation (Kees and Miller, 2002) for variably saturated groundwater flow. The CV-FE method evaluates fluxes at points where material properties and gradients in pressure and concentration are consistently defined, making it both suitable for heterogeneous media and mass conservative. Using the method of lines, the CV-FE discretisation gives a set of differential algebraic equations (DAEs) amenable to solution using higher-order implicit solvers.

Heterogeneous computer systems that use a combination of computational hardware such as CPUs and GPUs, are attractive for scientific computing due to the potential advantages offered by GPUs for accelerating data-parallel operations. We present a C++ library that implements data-parallel methods on both CPU and GPUs. The finite volume discretisation is expressed in terms of these data-parallel operations, which gives an efficient implementation of the nonlinear residual function. This makes the implicit solution of the DAE system possible on the GPU, because the inexact Newton-Krylov method used by the implicit time stepping scheme can approximate the action of a matrix on a vector using residual evaluations. We also propose preconditioning strategies that are amenable to GPU implementation, so that all computationally-intensive aspects of the implicit time stepping scheme are implemented on the GPU.

Results are presented that demonstrate the efficiency and accuracy of the proposed numeric methods and formulation. The formulation offers excellent conservation of mass, and higher-order temporal integration increases both numeric efficiency and accuracy of the solutions. Flux limiting produces accurate, oscillation-free solutions on coarse meshes, where much finer meshes are required to obtain solutions with equivalent accuracy using upstream weighting. The computational efficiency of the software is investigated using CPUs and GPUs on a high-performance workstation. The GPU version offers considerable speedup over the CPU version, with one GPU giving speedup factor of 3 over the eight-core CPU implementation.

Statement of Original Authorship

The work contained in this thesis has not been previously submitted for a degree or diploma at any other higher educational institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

SIGNED:

DATE: 18 SEPTEMBER 2012

Contents

1 Introduction **1**

1.1 Literature Review . . . 3

1.2 Objectives of The Thesis . . . 11

1.3 Contribution of The Thesis . . . 15

1.4 Overview of The Thesis . . . 15

2 Problem Formulation **19**

2.1 Governing equations . . . 19

2.2 Closure Of The System . . . 22

2.3 Boundary Conditions . . . 26

2.4 Formulation . . . 30

2.5 Conclusions . . . 34

3 Computational Techniques **35**

3.1 The Mesh . . . 36

3.1.1 The Finite Element Mesh . . . 36

3.1.2 The Dual Mesh 37

3.1.3 Interpolation 42

3.2 The Control Volume-Finite Element Method 46

3.2.1 Accumulation Terms 48

3.2.2 Source Terms 51

3.2.3 Surface Fluxes 51

3.2.4 Discretised Equations 63

3.3 Temporal Solution 68

3.3.1 Solving the Linear System 71

3.3.2 Preconditioner 72

3.4 Conclusions 75

4 Algorithms and Data Structures 77

4.1 Time Stepping with IDA 78

4.2 Evaluating the Residual: The CV-FE Discretisation 80

4.2.1 Time Step Preprocessing 83

4.2.2 Interpolation 88

4.2.3 Edge-Based Weighting 89

4.2.4 Fluid Properties 90

4.2.5 Flux Assembly 94

4.2.6 Residual Assembly 96

4.3 Domain Decomposition 97

4.4 Preconditioners 99
4.4.1 The Global Matrix 100
4.4.2 Finding the Local Block 101
4.4.3 Preconditioning the Local Block 103
4.5 Conclusions 106

5 GPU Implementation 107

5.1 Using GPUs as Computational Accelerators 108
5.1.1 Fermi Architecture 110
5.2 The vectorlib Library 113
5.3 Sparse Matrices 116
5.4 Domain Decomposition 118
5.4.1 Sub-Dividing the Computer Into Processes 118
5.4.2 Mesh Generation and Domain Decomposition 119
5.4.3 Communication Between Processes 120
5.5 The IDA Library 124
5.6 The CV-FE Discretisation 126
5.6.1 Interface 126
5.6.2 Edge-Based Weighting 129
5.6.3 Interpolation 132
5.6.4 Fluid Properties 132
5.6.5 Flux Assembly 134
5.6.6 Residual Assembly 136

5.7 Mesh Renumbering To Optimise Indirect Indexing 138

5.7.1 Indirect Indexing in Computing Relative Permeability . 140

5.8 Implementation of Preconditioners 141

5.8.1 Forming the Preconditioner 142

5.8.2 Applying the Preconditioner 143

5.9 Conclusions 145

6 Model Verification 148

6.1 Test Cases 148

6.1.1 Richards' Equation: Infiltration Into Dry Heterogeneous Soil – The *dry_infiltration* Test Case 150

6.1.2 Richards' Equation: Transient Water Table Experiment – The *water_table* Test Case 152

6.1.3 Transport Model: Flow and Transport in Unsaturated Soil – The *unsaturated_transport* Test Case 153

6.1.4 Transport Model: Flow Tank Experiments – The *tank_steady*, *tank_plume* and *tank_tidal_plume* Test Cases of Zhang . 155

6.1.5 Transport Model: Leaching of a Contaminant Plume in a Shallow Aquifer – The *heap_leaching* Test Case . . 159

6.2 Richards' equation: the *dry_infiltration* test case 161

6.3 Richards' equation: the *water_table* test case 173

6.4 Transport Model: Unsaturated flow and transport 180

6.5 Transport Model: Zhang's Flow Tank Experiments 188

6.6 Transport Model: Heap Leaching Simulation 209

6.7 Time Stepping With IDA 217

6.7.1 The role of different preconditioners 217
6.7.2 Higher-Order Temporal Integration 221
6.8 Conclusions 224

7 Computational Performance 229
7.1 Test Setup 229
7.1.1 Test Hardware 231
7.2 Results 231
7.3 Conclusions 250

8 Conclusions 252
8.1 Summary and Discussion 252
8.2 Directions For Further Research 257

A Computing Derivative Coefficients 261
A.1 The PR formulation of Richards' Equation 261
A.2 The PC Formulation of the Full Transport model 262

B Verification of Hydrostatic Boundary Condition 265

C Shape Function Weights 268
C.1 Shape function interpolation on quadrilaterals 268
C.2 Shape function interpolation on hexahedra 269

D Transient Seepage Faces in IDA 271

E G Under Saturated Conditions **272**

List of Figures

2.1 Saturation and relative permeability curves determined using typical parameters for sand, clay and loam soil types. 24

3.1 Heterogeneous two-dimensional domain with mixed triangle-quadrilateral finite element mesh. 37

3.2 Description of elements used in this work: triangles and quadrilaterals in two dimensions; tetrahedra and hexahedra in three dimensions. 38

3.3 Construction of a sub-control volume in two dimensions for triangle and quadrilateral elements. 39

3.4 Construction of a two-dimensional control volume for a node that lies on the boundary. 40

3.5 Construction of a sub-control volume for a tetrahedral element. 41

3.6 Description of an edge and the control volume faces associated with it in two dimensions. 55

3.7 Schematic for the choice of upstream, downstream and 2up nodes for an edge. 61

4.1 Flow chart of the steps taken for one internal time step of IDA. 79

4.2 Diagram showing the two faces attached to an edge in a two-dimensional mesh. . . . 86

4.3 Diagram illustrating the method used to determine the 1up and 2up points. . . . 87

4.4 Approach for finding fluid properties in a domain with few distinct material properties. . . . 93

4.5 Domain decomposition of a mixed two-dimensional mesh. . . . 98

4.6 Storage of a distributed vector. . . . 99

4.7 The global iteration matrix with four sub-domains. . . . 100

5.1 Schematic of GPU workstation with two GPUs and two sockets. 109

5.2 Caching of global memory access for indirect indexing. . . . 111

5.3 Allocation of computational resources in the workstation in Figure 5.1 for a domain decomposition of two sub-domains. . . 119

5.4 Storage of a distributed vector with overlap. . . . 121

5.5 The nodes, edges and faces accessed by a half warp in the mapping of permeability, k_{rw}, values onto control volume faces in Listing 5.10. . . . 142

6.1 The domain for the *dry_infiltration* test case due to Forsyth et al. (1995). . . . 151

6.2 The domain for the *water_table* test case due to Vauclin et al. (1979). . . . 153

6.3 The domain for the *unsaturated_transport* test case. . . . 154

6.4 Height of the tide for the *tank_tidal_plume* experiment. . . . 156

6.5 The laboratory flow tank, and corresponding computational domain for the laboratory flow tank experiments by Zhang (2000). . . . 158

6.6 Top and side views of the domain for the *heap_leaching* test case. 160

6.7 Fine mesh solution for the *dry_infiltration* test case. 165

6.8 Mass balance errors of the PR and MPR formulations for the *dry_infiltration* test case with very dry initial conditions. . . . 166

6.9 Comparison of pressure head contours using upstream weighting and flux limiting for the *dry_infiltration* test case with *dry* initial conditions at 30 days. 171

6.10 Comparison of pressure head contours using upstream weighting and flux limiting for the *dry_infiltration* test case with very dry initial conditions at 30 days. 172

6.11 Reference solution for *water_table* test case. 176

6.12 Comparison of the reference solution for the *water_table* test case to upstream and flux limited solutions. 179

6.13 Comparison of pressure head contours obtained using different spatial weighting schemes at $t = 48$ hours for the *unsaturated_transport* test case 183

6.14 Comparison of concentration contours obtained using different spatial weighting schemes at $t = 48$ hours for the *unsaturated_transport* test case 184

6.15 Reference solutions for the *water_table* test case. 187

6.16 Experimental and numerical locations of the water table and the sea water interface for the *tank_steady* flow tank experiment. 190

6.17 Steady state solutions for the *tank_steady* test case. 192

6.18 Time step size before and after the injection of contaminant at $t = 0$ min in the *tank_plume* test case. 196

6.19 Comparison between experimental and numerical results (upstream and flux limited) on meshes 2 and 4 for the *tank_plume* experiment. 197

6.20 Experimental results for the *tank_plume* test case, where contaminant is first injected at $t = 0$ min. 198

6.21 Solution contours for the *tank_plume* test case determined using upstream weighting and Mesh 3. 199

6.22 The location of the $c_0 = 0.5$ isochlor for the *tank_plume* test case using the finest mesh (mesh 6 with 214,012 nodes) and upstream weighting. 200

6.23 Comparison between experimental and numerical results (upstream and flux limited) on meshes 1 and 3 for the *tank_tidal_plume* experiment. 204

6.24 Comparison of experimental results for the contaminant plume with and without tidal variation 205

6.25 Comparison of numerical results for the contaminant plume with and without tidal variation 206

6.26 Experimental results for the *tank_tidal_plume* test case. 207

6.27 Numerical results using flux limiting on the coarsest mesh for the *tank_tidal_plume* test case. 208

6.28 Concentration isosurfaces using of the reference solution for the *heap_leaching* test case: front view. 211

6.29 Concentration isosurfaces using of the reference solution for the *heap_leaching* test case after 81 days: back view. 212

6.30 Comparison of high-resolution solutions for the *heap_leaching* test case. 214

6.31 Concentration contours using upstream weighting for the *heap_leaching* test case after 81 days. 215

6.32 Concentration contours using flux limiting for the *heap_leaching* test case after 81 days. 216

6.33 Triangular mesh for testing the efficacy of higher-order integration methods on the *dry_infiltration* test case. 222

6.34 Mass balance error for the *dry_infiltration* test case in terms of computational work. 224

7.1 The strong scaling for the CPU version. 244

7.2 Breakdown of time spent in each part of the solver for CPU×8 244

7.3 Speedup of physics computation for GPU×1 and GPU×2 (red and blue respectively) relative to CPU×8 (black) as the number of nodes in the mesh increases for two dimensions (a) and three dimensions (b). 245

7.5 Speedup of each step in the residual evaluation due to renumbering of nodes, edges and faces to obtain better cache performance. 246

7.6 Proportion of time spent in each step of the residual evaluation on the GPU with and without node, edge and face renumbering for cache performance. 247

7.7 Speedup of IDA on the GPU relative to the eight-core CPU implementation. 247

7.8 Time spent in each part of the solver for GPU×1 when using the host preconditioner, with further breakdown of time spent in different parts of the preconditioner. 248

7.9 Average time to apply the preconditioner for each of the different preconditioners when using the GPU. 248

7.10 Speedup of the entire solver when using GPUs relative to CPU×8 as the number of nodes in the mesh increases for two and three dimensions. 249

B.1 Analysis of error in computed pressure head at a beach boundary as the length scale parameter L is varied. 267

List of Tables

2.1 Model parameters, their dimensions and definitions. 22

4.1 The fluid properties used in the PR and MPR formulations of Richards' equation. 91

6.1 Material properties for the *dry_infiltration* test case. 151

6.2 Material properties for the *water_table* test case. 152

6.3 Parameters for *unsaturated_transport* test case. 154

6.4 Parameters for experiments by Zhang (2000). 157

6.5 Parameters for *heap_leaching* test case. 160

6.6 Details of the meshes used to verify the *dry_infiltration* test case. 166

6.7 Statistics for the solution of the *dry_infiltration* test case with *dry* initial conditions ($\psi^0 = -7.34$) using each combination of spatial averaging and mesh resolution with the PR formulation. 167

6.8 Statistics for the solution of the *dry_infiltration* test case with *dry* initial conditions ($\psi^0 = -7.34$) using each combination of spatial averaging and mesh resolution with the MPR formulation. 168

6.9 Statistics for the solution of the *dry_infiltration* test case with *very dry* initial conditions ($\psi^0 = -100m$) using each combination of spatial averaging and mesh resolution with the PR formulation. 169

6.10 Statistics for the solution of the *dry_infiltration* test case with *very dry* initial conditions ($\psi^0 = -100m$) using each combination of spatial averaging and mesh resolution with the MPR formulation. 170

6.11 Details of the three-dimensional tetrahedral meshes used to test the *water_table* test case. 176

6.12 Statistics for the solution of the *water_table* case study using each combination of spatial averaging and mesh resolution with the PR formulation. 177

6.13 Statistics for the solution of the *water_table* test case using each combination of spatial averaging and mesh resolution with the MPR formulation. 178

6.14 Computational performance and mass balance results for the *unsaturated_transport* test case using different combinations of spatial weighting schemes for the PC formulation. 185

6.15 Computational performance and mass balance results for the *unsaturated_transport* test case using different combinations of spatial weighting schemes for the MMPC formulation. 186

6.16 Details of the triangular meshes used for testing the unsaturated flow and contaminant transport test case. 187

6.17 Details of the triangular meshes used to for the flow tank experiments. 189

6.18 Computational performance metrics for the *tank_steady* test case using upstream weighting and flux limiting as mesh resolution increases. 193

6.19 Details of the tetrahedral meshes used for testing the *heap_leaching* test case. 213

6.20 Computational performance metrics for the *heap_leaching* test case using upstream weighting and flux limiting as mesh resolution increases. 213

6.21 Performance of the different choices of local preconditioner for the *dry_infiltration* and *water_table* test cases, for *serial* and *distributed* runs. 218

6.22 Computational efficiency and mass balance error for the *dry_infiltration* test case for both the PR and MPR formulations. 223

7.1 The name and number of nodes in each of the meshes used to test the parallel performance of FVMPor. 230

7.2 The average time spent evaluating the residual and applying the preconditioner for baseline 8-core simulations performed using two-dimensional and three-dimensional meshes with a similar number of nodes. 235

List of Algorithms

4.1 The steps in evaluating the residual function. 81

4.2 Algorithm for determining the scalar flow direction indicator FDI(φ). 85

4.3 Method for determining the 1up nodes and corresponding flux for each control volume in the domain. 88

4.4 Algorithm for determining the density at each face using precomputed edge weights. 90

4.5 Greedy multi-colouring algorithm used to find the minimal independent column set. 102

4.6 Forward and backward substitution for sparse triangular factors that have been reordered according to a multi-coloring. 105

List of Source Listings

5.1 Definition of Pattern class. 122
5.2 The interface to the Communicator class 123
5.3 Replacing memory allocation in NVector library with a CUDA call. 125
5.4 Replacing an NVector routine with a CUDA kernel. 126
5.5 Definition of Physics class that implements the residual evaluation for the Richards' equation model. 128
5.6 Definition of the block variable for the MPR model, and physics for CPU and GPU implementation. 129
5.7 Code for computing fluid properties for Richards' equation. . . 133
5.8 Flux assembly for the fluid mass $[q^{\mathrm{w}}]_j$ in three dimensions. . . 135
5.9 Source code for residual assembly of the PR formulation of Richards' equation. 137
5.10 Code for computing relative permeability at control volume faces in a homogeneous medium. 141

Acknowledgements

Finishing a project in three years is not possible alone – it is a team effort.

To my principle supervisor Professor Ian Turner, thank you for showing me just how much fun numerical analysis is in my undergraduate days, for guiding me through my postgraduate studies, and being patient whenever I started talking about cache.

I owe an enormous debt to Dr Timothy Moroney, who always had time to chat about ideas, who taught me to talk object oriented, and without whose help vectorlib would not have happened. Thanks Tim!

The work with GPUs wouldn't have happened without the support of Professor Kevin Burrage, who generously provided a GPU computer when I mentioned that I thought GPUs might make my simulations run faster.

Thanks to Mark Harris and Maxim Neumov from NVIDIA for discussing the inner workings of the CUSPARSE library with me, and to Qi Zhang who dug up his old experimental results for me.

Thanks the many friends who have helped out, particularly towards the end when things got strange. Your many acts of kindness made all the difference.

Thank you to my parents, who have been understanding and supportive (often when they had a right not to be) over the years.

And thank you Joana. You have joined this adventure very late on, and already made many sacrifices. I love you.

Ben Cumming.

September 2012.

Chapter 1

Introduction

Coastal aquifers are an important source of fresh water for domestic and agricultural use in many parts of the world. The fresh water resources they provide are sensitive to over-extraction and contamination due to both sea water intrusion and anthropogenic pollution. These pressures are particularly acute in many parts of Australia, where the majority of the population lives in arid and semi-arid coastal regions.

Coastal groundwater resources require careful management to prevent over extraction and to both prevent and remediate contamination. Computational modelling is an important tool for providing insight into the complicated hydrodynamics of groundwater flow and contamination, and for making predictions to inform resource management decisions. However, computational modelling of groundwater flow and contaminant transport is challenging. There are several reasons for this:

- First, computational models are based on mathematical descriptions of groundwater flow and contaminant transport. A detailed model must account for density-dependent flow and transport in porous media, and in unconfined aquifers the presence of both water and air in the void space of the aquifer in the unsaturated region directly below the ground

surface must also be considered. Furthermore, the partial differential equations in the mathematical models can be very stiff, necessitating the use of computationally intensive implicit time stepping methods.

- Second, the geology of aquifers can be complex and heterogeneous with different materials such as sand, silt, clay and rock. The flow rates and constitutive relationships associated with the different materials can vary significantly. Unstructured meshes are useful for representing such complicated geometry, yet they are also more complicated and difficult than structured meshes to implement in software.

- Third, the regions to which models are applied are typically large, measured in the scales of hundreds of metres up to many kilometres. This necessitates very large meshes, particularly for three-dimensional models, to accurately represent the aquifer and model the relevant physical phenomena.

To handle the size and complexity of groundwater models, software for modelling groundwater flow requires significant hardware resources. One way to achieve this is to use clusters, which allow the computational problem to be distributed across many computational nodes. However, such computational resources are expensive and require specialist skills to maintain. As such, they are not available to many scientists and engineers.

The recent advent of multi-core CPUs and many-core processors such as GPUs[1] has greatly increased the computational capabilities of powerful desktop computers and small clusters. However, to effectively utilise this computational potential, algorithms must be designed and implemented to take advantage of multi-core hardware.

Hence, a significant challenge in developing modern software for simulating the hydrodynamics of coastal aquifers is that the software should not only accurately model the physical phenomena — it should also be able to run

[1]Graphics Processing Units, named as such because these devices were originally intended for accelerating graphics operations.

efficiently on available computational resources, be they those of a cluster, GPUs, or a combination thereof.

1.1 Literature Review

Spatial Discretisation

There are many different numerical strategies for the solution of the partial differential equations (PDEs) used to model density-dependent groundwater flow and contaminant transport. Finite difference methods implemented in MODFLOW and its descendants (Langevin and Guo, 2006) are widely used – see for example Brovelli et al. (2007). However, finite difference methods are not readily applicable to unstructured meshes.

The finite element method and finite volume method are the most widely-used spatial discretisations for unstructured meshes. Examples of codes that use finite elements include SUTRA (Voss, 1994), FEFLOW (Trefry and Muffels, 2007) and Hydrus (Simunek et al., 2008). The HydroGeoSphere code (Therrien et al., 2010) uses a hybrid control volume-finite element discretisation[2], and the TOUGH family of codes (Pruess, 2001) use a cell-centred finite volume discretisation.

Finite element and finite volume methods discretise the domain using a mesh composed of non-overlapping cells, with material properties typically taken to be uniform over each cell. Finite element and cell-centred finite volume methods form fluxes at cell edges where properties such as permeability may be discontinuous, which requires some form of averaging to approximate the permeability. This can have a large impact on accuracy and computational efficiency of the method (Miller et al., 1998).

[2]The control volume-finite element discretisation used in HydroGeoSphere (Forsyth and Kropinski, 1997) should not be confused with the alternative control volume-finite element method used in this work, which is sometimes referred to as a finite volume element (FVE) method (Ewing et al., 2002, Martinez, 2006)

Care must also be taken to ensure that fluxes across cell faces are consistent to ensure local conservation of mass. The standard Galerkin finite element method has discontinuous velocity fields across cell faces, which are not locally mass conservative, and require modification to in order ensure oscillation-free, mass-conservative solutions (Kees et al., 2008). The mixed hybrid finite element method (MHFE) and its variants ensure consistent fluxes, however they require a discrete approximation of the temporal derivative (Farthing et al., 2003), which excludes the use of general integration codes via the method of lines (MOL). Younes et al. (2009) and Fahs et al. (2009) use a lumped MHFE method which can be used with MOL to simulate density-driven flows and Richards' equation[3] respectively.

The control volume-finite element (CV-FE) discretisation (sometimes referred to as a finite volume element (FVE) method (Ewing et al., 2002, Martinez, 2006)) is a vertex-centred finite volume method formulated on a dual mesh that is based on an underlying finite element mesh. The method uses shape functions, defined on the underlying finite element mesh, for interpolation. The CV-FE method has been successfully applied to modelling heat and mass transfer in wood drying (Turner and Perré, 2001, Perré and Turner, 2002, Truscott, 2004), which is modelled with similar physics to that for groundwater flow. The method has also been tested for saturated groundwater flow and transport in two dimensions (Liu et al., 2002, Moroney and Truscott, 2008).

The CV-FE discretisation is attractive for modelling flow in heterogeneous porous media because fluxes are evaluated at quadrature points in the interior of elements, where material properties are continuous and consistent pressure and concentration gradients are reconstructed using finite element shape functions. This ensures conservation of mass (Martinez, 2006), and does not require the averaging of physical properties such as permeability.

[3]Richards' equation is the most widely used model for describing variably-saturated flow in porous media. It is a nonlinear parabolic equation derived by coupling a mass conservation equation with Darcy's Law (Bear, 1979).

The CV-FE method used in this research work should not be confused with another method with the same name: the control volume-finite element method proposed by Forsyth et al. (Forsyth, 1991, Forsyth and Kropinski, 1997) that is implemented in the HydroGeoSphere[4] software package. The approach taken by Forsyth et al. assigns material properties to each node, without explicit treatment of the interface between control volumes and their faces.

When the transport of the gas phase is considered unimportant in unsaturated media, constant gas phase pressure is assumed, and only the fluid phase is modelled explicitly using Richards' equation. Richards' equation is not particularly sensitive to monotonicity constraints, however under certain circumstances the use of central weighting on coarse meshes can lead to oscillatory, non physical solutions (Forsyth and Kropinski, 1997, Diersch and Perrochet, 1999). To avoid the possibility of this occurring, care must be taken when reconstructing the relative permeability (mobility term) in flux terms. Contaminant transport is modelled using a nonlinear advection-diffusion equation that is more sensitive to spatial weighting (Neumann et al., 2011), such that fine meshes must be used to obtain monotone solutions for advection-dominated flows.

Upwind weighting of the mobility and advected terms gives oscillation free solutions that are unconditionally monotone. However, the solutions can suffer from excessive numeric diffusion (Patankar, 1980). Numeric diffusion leads to the smearing of sharp interfaces, such as those formed by infiltrating wetting fronts in dry soils, and in advection-dominated contaminant transport. Flux limiting (van Leer, 1979) has been shown to significantly reduce numeric diffusion in the solution of multiphase flow and transport in groundwater (Forsyth et al., 1996, Unger et al., 1996), and with the CV-FE method for wood drying (Turner and Perré, 2001, Perré and Turner, 2002, Truscott, 2004).

The pressure head formulation of Richards' equation, which uses the pres-

[4] `hydrogeosphere.org`

sure head as the primary variable, is generally not mass conservative. The mixed formulation, proposed by Celia et al. (1990) is mass conservative when discretised using a mass-conservative spatial discretisation, and can be implemented easily using first-order implicit Euler temporal integration (Forsyth and Kropinski, 1997). Tocci et al. (1997) investigated using the method of lines (MOL) to formulating the discretised equations as a system of differential algebraic equations (DAEs) that were amenable to adaptive higher-order integration libraries. They showed that higher-order integration improved the mass balance and computational efficiency of the pressure head formulation, making it preferable to first-order integration of the mixed form.

Kees and Miller (2002) proposed an alternative mixed formulation for Richards' equation, whereby both fluid mass and pressure head are primary variables, and an additional algebraic equation is imposed at each node. The resultant DAE system was solved using higher-order implicit time stepping, and was shown to be more accurate and mass conservative than the equivalent pressure head formulation. The approach has been extended to mixed hybrid finite element (Farthing et al., 2003) and stabilised finite element methods (Kees et al., 2008) for solving Richards' equation. However, it has not yet been applied to coupled groundwater flow and contaminant transport.

Temporal Discretisation

Implicit temporal integration methods are used in groundwater modelling due to their stability for both fine and coarse meshes (Unger et al., 1996), and because they allow large time steps to be taken when solving the stiff Richards' equation. Adaptive first-order implicit time stepping is widely used: the popular software package FEFLOW (Trefry and Muffels, 2007) uses an adaptive predictor-corrector method; Hydrus (Simunek et al., 2008) uses a method that ensures that the Courant and Peclet numbers do not exceed preset limits; and HydroGeoSphere (Therrien et al., 2010) uses adaptive time stepping based on the rate of change of variables and heuristic measures

relating to convergence of the Newton algorithm.

Using the method of lines (MOL), the spatial discretisation can be decoupled from the temporal integration. This gives a system of semi-discrete differential algebraic equations (DAEs) with the general form

$$\boldsymbol{F}(t, \boldsymbol{y}, \boldsymbol{y}') = \boldsymbol{0} \tag{1.1a}$$

$$\boldsymbol{y}(t_0) = \boldsymbol{y}_0 \tag{1.1b}$$

$$\boldsymbol{y}'(t_0) = \boldsymbol{y}_0', \tag{1.1c}$$

where time $t \in [t_0, t_{\text{final}}]$, and $\boldsymbol{y}, \boldsymbol{y}' \in \mathbb{R}^N$ are vectors of primary variables and their derivatives at mesh nodes. The nonlinear function $\boldsymbol{F}$ in (1.1a) is referred to as the *nonlinear residual function*.

The system of DAEs in (1.1) can be solved using implicit, higher-order temporal integration methods for DAE systems. Such libraries have been shown to be well-suited to the solution of systems derived from the discretisation of Richards' equation (Tocci et al., 1997, Kees and Miller, 2002, Fahs et al., 2009) and coupled density-driven flow and transport (Younes et al., 2006, Liu et al., 2002, Moroney and Truscott, 2008). Furthermore, robust DAE solver codes such as DASPACK and IDA[5] (Hindmarsh et al., 2005) offer sophisticated time step, order selection and error control according to user-specified tolerances.

IDA is a higher-order implicit solver based on backwards differentiation formulae (BDFs), which is part of the SUNDIALS suite of solvers (Hindmarsh et al., 2005). IDA uses an inexact Newton-Krylov method to solve the nonlinear system at each time step. The solution of nonlinear system using Newtons method requires the solution of linear systems, for which the Krylov subspace method GMRES Saad and Schultz (1986) is used.

An important point with the GMRES method is that it only requires the operation of a matrix on a vector in the form of a matrix-vector product.

[5] `computation.llnl.gov/casc/sundials`

The matrix-vector product can be approximated using difference quotients of shifted nonlinear residual evaluations, in which case it is not necessary to explicitly generate the iteration matrix (Kelley, 1995).

Heterogeneous Computing

Heterogeneous computer systems are broadly defined as computer systems that use a combination of different computing units. In the context of scientific computing, heterogeneous computers often use a combination of CPUs (central processing units) and GPUs (graphics processing units) to accelerate intensive computational tasks. In this section I will first review previous work on implementing unstructured mesh codes on GPUs, then look at the challenges associated with bringing implicit time stepping methods to GPUs.

GPUs were originally designed for specialised tasks in rendering computer graphics. These tasks require the application of identical operations to large sets of data, referred to as single instruction multiple data (SIMD), or data parallel, operations. The use of GPUs in scientific computing is motivated by the observation that many computationally intensive algorithms in scientific computing are also data parallel.

To obtain optimal instruction and memory throughput on GPUs, the data-access patterns of concurrently executing threads on the device[6] must conform to strict guidelines. These guidelines are imposed by the hardware due to the difficulty of serving thousands of simultaneous memory requests from threads.

The connections between nodes, edges and faces in a mesh give rise to the data access patterns associated with operations on the mesh. For structured meshes, these connections have a regular structure, that give rise to regular strided data access patterns. Programmers can take advantage of this *a*

[6] In a heterogeneous computer that uses both CPU and GPUs, the GPU and its attached memory are referred to as the *device*, and the CPU and its memory are referred to as the *host*.

priori knowledge of data structure to optimise mesh storage and the order in which mesh elements are processed to meet these guidelines. Early adoption of GPUs for the solution of PDEs focussed on structured meshes for this reason (Göddeke et al., 2007, Walsh et al., 2009, Rostrup and De Sterck, 2010).

With unstructured meshes however, there is no *a priori* knowledge of the connections between nodes, edges and faces in the mesh. This necessitates the use indirect indexing to describe the connections. As a result, codes for unstructured meshes that use GPUs are considerably more challenging to implement efficiently that those for structured meshes. Examples of research into implementing solvers for PDEs on unstructured meshes include (Klöckner et al., 2009, Corrigan et al., 2010, Komatitsch et al., 2010, de la Asuncin et al., 2011). Each of these papers focussed on steps to improve memory throughput on unstructured meshes.

Corrigan et al. (2010) investigated a cell-centred finite volume method for solving the Euler equations. They assigned one thread on the GPU to each cell in the mesh. Each thread computed the temporal derivatives of local variables in its cell from variable data at each of its vertices. This approach minimises the limitations of memory bandwidth on GPUs by loading values into shared and register memory on the GPU, computing the residuals locally, then copying the results back without any intermediate copying. The downside of this approach is that it requires problem-specific and method-specific kernels written and hand-tuned for a specific combination of physics and hardware, which limits its portability.

Klöckner et al. (2009) implemented discontinuous Galerkin (DG) methods for linear hyperbolic conservation laws. The authors focussed on using higher-order spatial discretisations to improve memory throughput. Evaluation of the DG discretisation was broken into four distinct stages. Of these stages, three could be represented using dense matrix vector multiplication, an operation that is more efficient as the order of the method increases.

The research in (Klöckner et al., 2009, Corrigan et al., 2010, Komatitsch et al., 2010, de la Asuncin et al., 2011) focussed on the solution of hyperbolic conservation laws, for which explicit time stepping methods such as Runge-Kutta can be used. The main computational expense for explicit methods is evaluating the spatial discretisation to form the right hand side. The remaining operations are forming linear combinations of vectors, which can be efficiently implemented on GPUs.

Implicit methods are generally more complicated to implement than explicit methods because they require the solution of large, sparse linear systems. However, as was discussed earlier, the matrix-free Newton-Krylov method can use the nonlinear residual function to approximate matrix-vector products. If the nonlinear residual function can be implemented efficiently on the GPU, then the remaining level 1 BLAS[7] vector operations in the Newton and GMRES methods are readily implemented on the GPU.

In the field of computer vision, Wu et al. (2011) investigated such a Newton-Krylov solver using a preconditioned conjugate gradient method, with matrices that had a special structure for which it was possible to implement an efficient GPU preconditioner.

The linear systems that arise from the discretisation of the PDEs that govern groundwater flow are often ill-conditioned. A preconditioner for general matrices is required to ensure the timely convergence of Krylov subspace methods for such ill-conditioned matrices. Incomplete LU (ILU) factorisations are very popular preconditioners for general matrices, however the algorithms for factorising and applying sparse triangular factors are not inherently parallel.

Limited parallelism can be obtained for ILU methods by analysis of the sparsity pattern of the sparse factors, which must be known *a priori*. A

[7] Basic Linear Algebra Subprograms (BLAS) is an interface standard for libraries that implement basic low-level dense linear algebra operations. Level 1 BLAS contains vector operations such as linear combinations, dot products and norms. The dense linear algebra operations in BLAS are typically data-parallel, and can be implemented efficiently on GPUS.

multi-colouring of the matrix graph associated with the sparse factors gives sets of independent rows that can be computed concurrently (Saad, 2000). Li and Saad (2010) investigated implementing incomplete Cholesky factorisations without fill in on the GPU, in which case the sparse factors have the same sparsity pattern as the original matrix. Very recently, Heuveline et al. (2011a,b) applied this approach to general matrices, and factors that have the same sparsity pattern integer powers of the matrix. A sparse triangle solver for based on the same principles has also been implemented recently in the CUSPARSE library Naumov (2011).

1.2 Objectives of The Thesis

The objectives of this PhD research program will now be outlined.

Formulate a mass-conservative control volume finite element method for modelling variably-saturated groundwater flow and contaminant transport in heterogeneous porous media

The control volume-finite element method has been successfully used to model the drying of wood, a challenging heat and mass transfer problem governed by similar physics as those that govern the hydrological processes considered in this work (Turner and Perré, 2001). The method has also been used to model sea water intrusion in confined aquifers in two dimensions (Liu et al., 2002), however, to the best of our knowledge, there has been no detailed attempt to apply the CV-FE discretisation to variably-saturated groundwater flow and contaminant transport in two and three dimensions.

Due to the size of the domains used in groundwater models, it is necessary to use relatively coarse mesh resolutions, particularly in three dimensions. To ensure monotone solutions on coarse meshes, upstream weighting is typically used, however upstream weighting suffers from well-documented nu-

meric diffusion (Patankar, 1980). Flux limiting gives monotone solutions with significantly reduced numeric diffusion on coarse meshes, and has been used successfully for the CV-FE method in wood drying (Turner and Perré, 2001, Perré and Turner, 2002), and for other discretisations in multi-phase groundwater flows (Forsyth et al., 1996, Unger et al., 1996). An important part of this project will be to investigate flux limiters for the CV-FE discretisation of groundwater flow and contaminant transport.

Extend the mixed formulation for Richards' equation due to Kees and Miller (2002) to the CV-FE method and coupled contaminant transport

A modification of the mixed formulation for Richards' equation was proposed for finite difference methods by Kees and Miller (2002), and later applied to mixed hybrid finite element methods (Farthing et al., 2003). The formulation was shown to have superior accuracy and conservation of mass compared to the pressure head formulation. In this work, the formulation will be investigated in the context of the CV-FE discretisation, and extended to the coupled model for groundwater and contaminant transport.

An important feature of the modified mixed formulation is that, unlike the mixed formulation of Celia et al. (1990), it is amenable to solution using higher-order implicit methods for solving systems of DAEs. In this thesis IDA (Hindmarsh et al., 2005), a library for solving DAE systems, will be used. IDA uses a higher-order implicit time stepping scheme that uses a matrix-free Newton-Krylov method. The Newton-Krylov method is well-suited to solving very large problems, due to scalability of Krylov methods on distributed memory computers (Saad, 2000, Chapter 11).

Compared to the pressure head formulation for Richards' equation, the modified mixed formulation imposes an additional algebraic equation at each node of the mesh, which doubles the size of the iteration matrix. Previous work with the formulation (Kees and Miller, 2002, Farthing et al., 2003) showed

that size of the linear system can be reduced by half, to remove the computational overhead imposed by the additional equation. In this thesis this approach is applied to the preconditioner, so that the modified fixed formulation can be implemented in a matrix-free context.

Implement the proposed methods in a flexible software package that can be used on a desktop computer, and scale up to run on large clusters

The large meshes and implicit time stepping methods make the solution of detailed large-scale groundwater models computationally expensive. An aim of this work is to write an efficient, modular and extensible software package for density-dependent groundwater flow and contaminant transport.

The software package, called FVMPor (*F*inite *V*olume *M*ethod for *Por*ous Media) will be implemented in C++. Object oriented methods will provide modularity and flexibility in the code, while allowing for efficient implementation of performance-critical functions. Domain decomposition and the message passing library MPI will be used for implementation on clusters.

The software has the following requirements:

- The software will model both unsaturated flow via Richards' equation and the full coupled model of Richards' equation with contaminant transport.
- Handle both two-dimensional and three-dimensional models with the same code.
- Be able to be run on computers a wide range of computers, so that small to medium-sized problems can be run on desktop computers, and large-scale simulations on clusters.
- The code should be easy to extend to model other applications for multi-phase flow and transport problems in heterogeneous media. To

facilitate this, the back end such as the temporal integration, preconditioners and meshing should be independent of the model in question.

Investigate using GPUs to accelerate FVMPor

The vectorlib library[8] is a templated C++ library that provides an intuitive interface, familiar to users of MATLAB, for vector and matrix operations. An aim of this work is to extend the vectorlib library so that it can store and operate on data using different computational hardware. Two hardware back-ends will be implemented: one for multiple-core CPUs; and another that uses the CUDA programming language to utilise GPUs developed by NVIDIA.

The CV-FE spatial discretisation of the governing equations will be implemented using data-parallel operations provided by the vectorlib library. To improve the GPU performance of the indirect indexing operations necessitated by unstructured meshes, a numbering scheme for the nodes, edges and faces in the mesh will be investigated. The numbering scheme will aim to increase the correlation between the location of entities in space, and the location of data associated with them, in memory.

To perform the entire implicit time stepping scheme on the GPU, the DAE solution package IDA will be modified to use GPUs, and an ILU(0) preconditioner for general matrices will also be implemented on the GPU. The GPU implementation will then be benchmarked against the CPU code on a high-performance workstation.

[8] Originally developed by Dr Timothy Moroney.

1.3 Contribution of The Thesis

The original contributions of the research presented here are summarised below:

- The adaptation of the modified mixed formulation for Richards' equation for the CV-FE method, and extending the formulation to the coupled density dependent groundwater flow and contaminant transport, is new work, presented in Chapter 3. The preconditioning strategy presented in §3.3.2 for the modified mixed formulation, is also novel. The results of this work were published (Cumming et al., 2011).
- The GPU implementation of a matrix-free Newton-Krylov framework for implicit time stepping. To implement this, the popular open source package IDA (Hindmarsh et al., 2005) was modified, and the source code has been released under the BSD license, and can be downloaded from `github.com/bencumming/NVectorCUDA`.
- The novel numbering and storage scheme designed to improve instruction and data throughput on GPUs for unstructured meshes presented in §5.7.
- The GPU method for solving sparse triangle linear systems using multi-colouring, used to apply an ILU(0) preconditioner on the GPU.

1.4 Overview of The Thesis

Chapter 1 – Introduction

The motivation for using high-performance computing methods in modelling sea water intrusion is given. A review of the relevant literature is then presented. The chapter closes by stating the aims and novel contributions of this research program.

Chapter 2 – Problem Formulation: Groundwater Flow and Salt Transport

The mathematical model for variably-saturated groundwater flow and contaminant transport is given. The modified mixed formulation of Kees and Miller (2002) for Richards' equation is extended to include contaminant transport. The aim of this chapter is to clearly state the mathematical description of the physics in a form suitable for the methods introduced in the Chapter 3.

Chapter 3 – Computational Techniques: Control Volume-Finite Element Method with Implicit Time Stepping

The first part of this chapter introduces the control volume-finite element spatial discretisation, and formulate it for the variably-saturated flow and contaminant transport equations. Close attention is paid to the treatment of accumulation terms in the different formulations, and to methods for accurately and consistently calculating fluxes at cell faces using flux limiting and finite element shape functions.

The second part of the chapter gives an overview of the multi-step implicit solution method employed by the IDA library for solving the system of differential algebraic equations (DAEs) that arises from the finite volume discretisation. An efficient preconditioner suitable for the modified mixed formulations is proposed.

Chapter 4 – Implementation: Algorithms and Data Structures

This chapter presents a high-level discussion of the algorithms, data structures and of the CV-FE discretisation and implicit time stepping implemented in FVMPor. The chapter starts with an overview of steps taken in an internal time step of IDA, followed by a discussion of the CV-FE spatial

discretisation. Then the method of domain decomposition used to distribute the model over different computational nodes is presented. Finally methods are discussed for obtaining fine-grained and coarse-grained parallelism of preconditioners based on sparse factorisations.

Chapter 5 – Implementation on GPU Clusters using C++, MPI and CUDA

This chapter presents a description of the implementation of the algorithms developed in Chapter 4 to run efficiently in on multi-core and GPU computer sytems. This chapter starts with a discussion of general-purpose computing on GPUs, with a focus on the issues that arise when using unstructured meshes and an implicit time stepping scheme. Then the vectorlib library, which implements data parallel operations on both multi-core CPU and GPU hardware will be introduced. The rest of the chapter will discuss in detail the implementation of the CV-FE discretisation, the IDA time stepping library and preconditioners.

Chapter 6 – Model Verification

In this chapter the accuracy and numerical performance of the proposed methods are investigated along with the computational performance of the implementation.

In the first part of this chapter, the CV-FE discretisation and implicit solution strategy proposed in Chapter 3 are verified. Benchmark problems for Richards' equation, and for the full coupled model from the literature are used to investigate accuracy of the solutions using the modified mixed formulation, mass balance errors, and the accuracy of upstream weighting and flux limiting.

The second part of this chapter investigates the performance of the higher-

order time stepping scheme. The efficacy of different preconditioners for the serial and cluster implementations are investigated. Then, the effect of the order of the time stepping method and error tolerances on mass balance errors is discussed in detail.

Chapter 7 – Computational Performance

This chapter investigates the computational performance of the implementation. Residual evaluation, which involves computing the finite volume formulation, is the main computational bottleneck of the CPU code. The GPU version of the code performs this task very well, such that application of the preconditioner in the Krylov method becomes the bottleneck. We investigate different preconditioning strategies, and show that a combination of preconditioning on CPU and GPU depending on the problem size and properties produces very good speedup for the GPU over the whole solution process.

Chapter 8 – Conclusion and Discussion

The findings of this research project are discussed, and extensions of the work are proposed.

Chapter 2

Problem Formulation: Groundwater Flow and Salt Transport

The physics and associated mathematical descriptions of variably-saturated flow coupled with salt transport in aquifers are well-established. In this chapter we state the relevant conservation equations, boundary conditions and closure relations that govern the flow phenomena considered in this thesis, in a form that is amenable to solution using the numerical methods proposed in Chapter 3. Except for the MMPC formulation presented in §2.4, the material presented in this chapter is not original work, and the interested reader can find detailed derivations of the equations in the definitive books by Bear et al. (Bear, 1979, Bear and Verruijt, 1987, Bear and Cheng, 2010), and the excellent review of numerical modelling for density dependent flow and transport by Diersch and Kolditz (2002).

2.1 Governing equations

Flow in the shallow, unsaturated vadose zone is treated as a two-phase flow, with a liquid (water) phase and a gas (air) phase. It is certainly possible to

model both phases explicitly using a two-phase formulation (Forsyth et al., 1995, Kees and Miller, 2002). However, if the vadose zone is thin relative to the depth of the aquifer it is reasonable to make the simplifying assumption of constant gas-phase pressure (Bear and Cheng, 2010), such that only the water phase, or wetting phase[1], needs to be modelled explicitly.

Richard's Equation is the most widely used model for variably-saturated flow in porous media, derived by coupling a mass balance law with Darcy's law for flow in porous media. We consider the following mixed form of (Kees and Miller, 2002, Farthing et al., 2003)

$$\frac{\partial(\rho\theta)}{\partial t} = -\boldsymbol{\nabla}\cdot(\rho\boldsymbol{q}) + \rho S, \tag{2.1}$$

where ρ is the density of the water, θ is the moisture content (see equation (2.13)) and S is a source term that accounts for the volumetric rate of extraction, or injection of water.

The flow is assumed to have low Reynolds number, that is, the flow is slow and the viscous terms outweigh the momentum terms. As such, the momentum of the fluid is ignored, and the volumetric flux of fluid, $\boldsymbol{q}$, is assumed to be driven by a pressure gradient and gravity alone

$$\boldsymbol{q} = -\frac{\mathsf{k}_s\rho_0 g k_{\mathrm{rw}}}{\mu}\left(\boldsymbol{\nabla}\psi + \frac{\rho}{\rho_0}\boldsymbol{\nabla}z\right). \tag{2.2}$$

Equation (2.2) is often referred to as Darcy's law, in which the gravitational force, with acceleration g, is in the direction of the negative z-axis; ρ_0 is the density of fresh water; μ is the viscosity of water; k_s and k_{rw} are the absolute and relative permeability of the porous medium; and the pressure head is defined in terms of fluid pressure p:

$$\psi = \frac{p}{\rho_0 g}. \tag{2.3}$$

[1]The water phase is referred to as the *wetting* phase because in the presence of both phases, the water phase adheres to, or wets, the solid matrix, and the air phase occupies the remaining void space in the porous medium.

The volumetric flux $\boldsymbol{q}$ described by Darcy's law in equation (2.2) can be expressed in the simplified form

$$\boldsymbol{q} = -\mathsf{K} k_{\mathrm{rw}} \left(\boldsymbol{\nabla} \psi + \hat{\rho} \boldsymbol{\nabla} z \right), \tag{2.4}$$

where $\mathsf{K} = \mathsf{k}_s \rho_0 g / \mu$ is the hydraulic conductivity, and $\hat{\rho} = \rho / \rho_0$ is the scaled (dimensionless) density.

The mass balance for salt is described by an advection-dispersion equation

$$\frac{\partial (c\theta)}{\partial t} = -\boldsymbol{\nabla} \cdot (c\boldsymbol{q}) + \boldsymbol{\nabla} \cdot (\theta \mathsf{D}_h \boldsymbol{\nabla} c) + cS, \tag{2.5}$$

where c is the salt concentration, defined as the mass of salt per unit volume of water, and the final term on the right hand side is a source term. The flux of salt is composed of advective and dispersive components. The first term on the right hand side of equation (2.5) is the advective component, which accounts for the bulk transport of salt at the water's average velocity. The second term accounts for the dispersive components, which are described in detail in §2.2.

By defining the flux of water mass, $\boldsymbol{q}^{\mathrm{w}}$, and flux of solute mass, $\boldsymbol{q}^{\mathrm{s}}$, respectively as

$$\boldsymbol{q}^{\mathrm{w}} = \rho \boldsymbol{q}, \tag{2.6a}$$

$$\boldsymbol{q}^{\mathrm{s}} = c\boldsymbol{q} - \theta \mathsf{D}_h \boldsymbol{\nabla} c, \tag{2.6b}$$

the mass balance equations can be written in the following form that is suitable for use in developing the finite volume spatial discretisation presented in Chapter 3:

$$\frac{\partial (\rho\theta)}{\partial t} = -\boldsymbol{\nabla} \cdot \boldsymbol{q}^{\mathrm{w}} + \rho S; \tag{2.7a}$$

$$\frac{\partial (c\theta)}{\partial t} = -\boldsymbol{\nabla} \cdot \boldsymbol{q}^{\mathrm{s}} + cS. \tag{2.7b}$$

Parameter	Dimensions	Description
α	$L\,T^2M^{-1}$	coefficient of soil compressibility
α_L	L	longtidudinal dispersivity
α_T	L	transverse dispersivity
α_{vg}	L^{-1}	emperical parameter for van Ganuchten-Mualem model
β	$L\,T^2M^{-1}$	coefficient of compressibility
η	$M^{-1}L^3$	density coupling coefficient
ϕ_0	1	porosity of porous medium under no strain
ρ_0	$M\,L^{-3}$	density of fresh water, taken as $1000\,\mathrm{kg\,m^{-3}}$
μ	$M\,L^{-1}\,T^{-1}$	dynamic viscosity, taken as $10^{-3}\,\mathrm{kg\,m^{-1}\,s^{-1}}$.
c_0	$M\,L^{-3}$	reference salt concentration, taken as $0\,\mathrm{kg\,m^{-3}}$.
D_m	L^2T^{-1}	molecular diffusivity
g	$L\,T^{-2}$	acceleration due to gravity, taken as $9.82\,\mathrm{m\,s^{-2}}$
k_{rw}	1	relative permeability
k_s	L^2	absolute permeability
K	$L\,T^{-1}$	hydraulic conductivity
n_{vg}	1	emperical parameter for van Ganuchten *psk* model
m_{vg}	1	emperical parameter for van Ganuchten *psk* model, $m_{vg} = 1 - n_{vg}$
p_0	$M\,L^{-1}T^{-2}$	reference pressure, taken as $0\,kg\,m^{-1}s^{-2}$
S_r	1	residual saturation level.

Table 2.1: Model parameters, their dimensions and definitions. The dimensions are given in terms of length (L), time (T) and mass (M).

2.2 Closure Of The System

The governing equations in the previous section refer to quantities, such as k_{rw}, D_h, and ρ, that must be expressed by phenomenological laws, equations of state and constitutive functions.

The saturation of water is the proportion of void space in the porous medium

occupied by water, defined as

$$S_w = S_r + (1 - S_r)S_e, \tag{2.8}$$

where S_r is the residual moisture content, and S_e is the effective saturation. Various analytical descriptions have been proposed for the relationship between fluid pressure and effective saturation, including those of Brooks and Corey (1966) and van Genuchten (1980). In this work we use the description of van Genuchten (1980)

$$S_e = \begin{cases} 1 & \text{for } \psi \geq 0 \\ [1 + (\alpha_{\text{vg}}|\psi|)^{n_{\text{vg}}}]^{-m_{\text{vg}}} & \text{for } \psi < 0 \end{cases}, \tag{2.9}$$

where the parameters α_{vg}, n_{vg} and m_{vg} are specific to the porous medium, determined empirically by way of experimental data.

The relative permeability, k_{rw}, of the porous medium is dependent on the effective saturation, with the following relationship proposed by Mualem (1976)

$$k_{\text{rw}} = S_e^{1/2} \left[1 - (1 - S_e^{1/m_{\text{vg}}})^{m_{\text{vg}}}\right]^2. \tag{2.10}$$

The functional relationships between saturation and pressure head, and relative permeability and pressure head in (2.10) by way of (2.9), are very nonlinear, and are not Lipschitz continuous for $1 \leq n_{\text{vg}} < 2$ (Miller et al., 1998). The relationships also vary significantly between different materials in a heterogeneous aquifer, as illustrated in Figure 2.1.

When pressure head is greater than or equal to zero ($\psi \geq 0$), the wetting phase completely occupies the void space of the porous medium – which is said to be fully saturated – and saturation, relative permeability, and effective

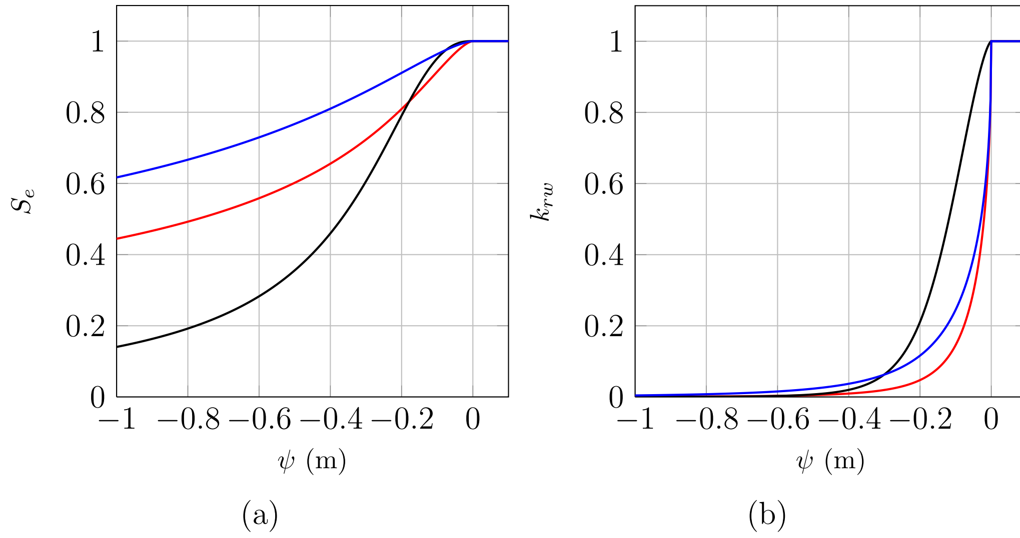


Figure 2.1: Typical saturation (a) and relative permeability (b) curves for different soil types: sand (black), clay (red) and loam (blue). The van Ganuchten parameters for each soil type were taken from Hodnett and Tomasella (2002).

saturation are unity:

$$\begin{aligned} S_e &= 1, \\ S_w &= 1, \\ k_{\mathrm{rw}} &= 1. \end{aligned}$$

Density, ρ, depends on both salt concentration c and water pressure p, with the general form

$$\rho = \rho_0 \mathrm{e}^{\beta(p-p_0)+\eta(c-c_0)},$$

where p_0 and c_0 are a reference pressure and reference concentration respectively, β is the coefficient of fluid compressibility, and η is the density coupling coefficient. For the range of pressures and concentrations considered in coastal aquifers, the parameters β and η are treated as small constants, and the following linear form is assumed (Diersch and Kolditz, 2002)

$$\rho = \rho_0 \left(1 + \beta(p - p_0) + \eta(c - c_0)\right).$$

Taking the reference pressure to be atmospheric pressure, equivalent to $\psi = 0$, and the reference concentration to be that of fresh water, equivalent to $c_0 = 0$, and recalling the relationship between pressure head and pressure in equation (2.3), density can be expressed as a function of pressure head and concentration:

$$\rho = \rho_0 \left(1 + \rho_0 g \beta \psi + \eta c\right). \tag{2.11}$$

In the general case, the solid matrix is also assumed to be deformable. If only small volume changes are considered, then the matrix can be assumed to behave like an elastic material, which gives the following functional form for the porosity

$$\phi = 1 - (1 - \phi_0)\mathrm{e}^{-\alpha(p - p_0)},$$

where α is the coefficient of matrix compressibility. Again, taking reference pressure head to be atmospheric, and linearising gives the following approximation for porosity as a function of pressure head

$$\phi = \phi_0 + (1 - \phi_0)\alpha \rho_0 g \psi. \tag{2.12}$$

The moisture content, θ, is the volumetric proportion of the medium occupied by water

$$\theta = \phi S_w, \tag{2.13}$$

and is a function of pressure head only. Given the moisture content, we define two further variables, the mass of fluid per unit volume, M, and mass of salt per unit volume, C:

$$M = \rho\theta, \tag{2.14a}$$

$$C = c\theta, \tag{2.14b}$$

which are functions of both pressure and salt concentration.

The transport of salt is affected by both dispersive and diffusive fluxes. The combined effect of these fluxes is accounted for in the coefficient of hydrody-

namic dispersion, D_h, a symmetric tensor defined by

$$\mathsf{D}_h = \mathsf{D} + \mathsf{D}_m^*, \tag{2.15}$$

where D is the coefficient of dispersion and D_m^* is the coefficient of molecular diffusion.

The dispersive flux is due to small variations from the average flux of salt arising from variations in the fluid velocity due to the structure of the porous medium. The coefficient of dispersion is dependent on the average fluid velocity, described using the Bear–Scheidegger dispersion relationship

$$\mathsf{D} = (a_L - a_T)\frac{\boldsymbol{v}\boldsymbol{v}^T}{\|\,\boldsymbol{v}\,\|_2} + a_T \,\|\,\boldsymbol{v}\,\|_2\, \delta_{ij}, \tag{2.16}$$

where a_L and a_T are the longitudinal and transversal dispersivity respectively; δ_{ij} is the Kronecker delta; and $\boldsymbol{v}$ is the average fluid velocity, which has the following relationship with the Darcy flux

$$\boldsymbol{q} = \theta \boldsymbol{v}. \tag{2.17}$$

Diffusion of salt due to the random motion of salt molecules in the water is modelled as a linear Fickian process, with the coefficient of molecular diffusion defined

$$\mathsf{D}_m^* = D_m \mathsf{T}, \tag{2.18}$$

where D_m is the molecular diffusivity, and the tensor T is the tortuosity of the porous medium.

2.3 Boundary Conditions

At the boundary, two boundary conditions are assigned: one for the fluid mass-balance equation, and one for the salt mass-balance equation. Detailed

derivations of the boundary conditions discussed here are given by Bear et al. (Bear and Verruijt, 1987, Bear and Cheng, 2010). For the mass-balance equations defined on a domain Ω with boundary Γ, we define the following five boundary conditions. Types 1–3 pertain to the fluid-mass balance equation (2.1), and Types 4–6 to the salt mass-balance equation (2.5).

Type 1: Prescribed Pressure Head

The Dirichlet boundary condition on pressure head has the general form

$$\psi(\boldsymbol{x}, t) = \psi_b(\boldsymbol{x}, t) \quad \text{on } \Gamma, \tag{2.19}$$

where the prescribed pressure head ψ_b is a function of space and time. If the boundary is in contact with a body of water, a *hydrostatic* boundary condition prescribes the pressure head as a function of the depth of the water at the boundary:

$$\psi_b(z, t) = (1 + \eta c_b)(h(t) - z), \tag{2.20}$$

where $h(t)$ is the height of the surface of the external water body, η is the density coupling coefficient, and c_b is the concentration of salt in the external water body.

Type 2: Prescribed Fluid Flux

When the flux of fluid normal to the boundary is known, such as rainfall on the surface or recharge from an adjacent aquifer, the normal component of the volumetric flux is prescribed using a Neumann boundary condition

$$\boldsymbol{q} \cdot \boldsymbol{n} = q_b(\boldsymbol{x}, t) \quad \text{on } \Gamma, \tag{2.21}$$

where $\boldsymbol{n}$ is the outwards-facing unit normal to the boundary. Because the normal is outwards-facing, a positive q_b indicates flow leaving the domain, and a negative q_b indicates influx of fluid.

A special case of the prescribed flux boundary condition is the no-flux condition:

$$\boldsymbol{q} \cdot \boldsymbol{n} = 0 \quad \text{on } \Gamma, \tag{2.22}$$

which is imposed on impermeable boundaries.

Type 3: Mixed Beach

The hydrostatic boundary condition in (2.20) is only physically realistic at points below the surface external water body, that is when $z > h(t)$. When modelling the shore line next to a body of water, such as a beach, parts of the boundary may be above the surface of the external water body. Under such conditions a mixed boundary condition is imposed: below the external water level the hydrostatic Dirichlet condition in equation (2.20) is imposed, with the sea level specified by $h(t)$; and the no flow boundary condition in (2.22) is imposed above the external water level, that is, for $z > h(t)$.

When the water level changes over time, for example due to tidal fluctuations, imposing two separate boundary conditions can cause convergence difficulties for numerical time stepping schemes. An alternative way of formulating the boundary condition is using a Cauchy condition, where the flux over the boundary is formulated as

$$\boldsymbol{q} \cdot \boldsymbol{n} = R_b(\psi - \psi_b(z,t)), \tag{2.23}$$

where the prescribed pressure head at the boundary, ψ_b, is dictated by the height of the tide, namely

$$\psi_b(z,t) = (1+\eta)(h(t) - z). \tag{2.24}$$

The penalty term R_b in equation (2.23) is chosen as follows

$$R_b = \begin{cases} 0, & z > h(t) \\ \frac{K}{L}, & z \leq h(t) \end{cases}, \tag{2.25}$$

where K is hydraulic conductivity (under the assumumption of isotropic conditions), and L is the length coupling scale (Chui and Freyberg, 2009). At points above the tide, that is when $z > h(t)$, the penalty term R_b is zero so that a no flow boundary condition is imposed in equation (2.23), and there is a flux over the boundary at points below the level of the tide. This approach is equivalent to applying a free drainage boundary condition at nodes on or below the level of the tide, where the value of R_b corresponds to an equivalent conductance (Therrien et al., 2010). Small values of the length coupling scale L give a large value of R_b, and enforce the Dirichlet boundary condition more strictly. Analysis of the accuracy of this boundary condition is presented in Appendix B.

Type 4: Prescribed Solute Concentration

A Dirichlet boundary condition can be applied when the salt concentration is known at the boundary, with the general form:

$$c(\boldsymbol{x}, t) = c_b(\boldsymbol{x}, t) \quad \text{on } \Gamma. \tag{2.26}$$

Type 5: Advective Solute Exchange

If fluid is allowed to pass through the boundary, and the concentration of salt in the fluid external to the domain is known, such as at the sea–land interface, the following Cauchy boundary condition can be imposed

$$\boldsymbol{q}^{\mathrm{s}} \cdot \boldsymbol{n} = c^{*}(\boldsymbol{x}, t)\boldsymbol{q} \cdot \boldsymbol{n} \quad \text{on } \Gamma, \tag{2.27}$$

where the interchange concentration c^* is defined by

$$c^*(\boldsymbol{x},t) = \begin{cases} c(\boldsymbol{x},t) & \boldsymbol{q}\cdot\boldsymbol{n} \geq 0 \\ c_b(\boldsymbol{x},t) & \boldsymbol{q}\cdot\boldsymbol{n} < 0 \end{cases} .$$

Thus, if the flux of fluid is out of the domain, the concentration of the fluid is that of the interior of the domain. Conversely, if flux is into the domain, the concentratation of the fluid flowing over the boundary is that of the fluid in the external body of water, c_b. This type of boundary condition assumes that the dispersive flux is small relative to the flux of solute due to advection at the boundary, and is often written $\partial c/\partial n = 0$.

2.4 Formulation

The choice of primary variables used in formulating the mass balance equations (2.1) and (2.5) dictates the formulation of the numerical method used to solve the equations. In this work, we seek formulations that are amenable to solution using the method of lines (MOL). The method of lines requires that the temporal derivative in the accumulation term of each mass balance equation is in terms of a primary variable. For example, the pressure head formulation of Richards' equation expresses the accumulation term in (2.1) as a function of pressure head and its temporal derivative only (Tocci et al., 1997). The pressure head formulation is not mass-conservative when solved with first order time stepping, so the mixed formulation is used with first order implicit time stepping (Celia et al., 1990). Tocci et al. (1997) showed that the pressure head formulation is mass conservative when solved using higher-order time stepping.

Kees and Miller (2002) proposed a variation on the mixed formulation of Richards' equation, that we call the *modified mixed formulation*. The proposed formulation is amenable to solution using the method of lines and higher-order implicit time stepping schemes, and is more accurate and mass

conservative than the pressure head formulation (Kees and Miller, 2002, Kees et al., 2008). In this section the pressure head and modified mixed formulations of Richards' equation due to Kees and Miller (2002) are first presented. Then, both formulations are extended to the coupled flow and salt transport equations.

PR: The pressure head formulation for Richards' equation

To derive the pressure head formulation of Richards' equation, the chain rule is applied to the accumulation term in the mixed formulation given in equation (2.7a)

$$\begin{aligned}\frac{\partial(\rho\theta)}{\partial t} &= \frac{\mathrm{d}(\rho\theta)}{\mathrm{d}\psi}\frac{\partial\psi}{\partial t} \\ &= \left(\theta\frac{\mathrm{d}\rho}{\mathrm{d}\psi} + \rho\frac{\mathrm{d}\theta}{\mathrm{d}\psi}\right)\frac{\partial\psi}{\partial t} \\ &= n(\psi)\frac{\partial\psi}{\partial t},\end{aligned} \tag{2.28}$$

where the functional form of the storage term

$$n(\psi) = \theta\frac{\mathrm{d}\rho}{\mathrm{d}\psi} + \rho\frac{\mathrm{d}\theta}{\mathrm{d}\psi}, \tag{2.29}$$

can be determined from the closure relations in §2.2 (see Appendix A for details). Substituting (2.28) for the accumulation term in (2.7a) gives the pressure head form of Richards' equation

$$n(\psi)\frac{\partial\psi}{\partial t} = -\boldsymbol{\nabla}\cdot\boldsymbol{q}^{\mathrm{w}} + \rho S. \tag{2.30}$$

The pressure head formulation as given by equation (2.30) is referred to throughout this thesis as the PR formulation.

MPR: The modified mixed formulation for Richards' equation

Kees and Miller (2002) proposed a formulation for the mixed form of Richards' equation, where the variable $M = \rho\theta$, defined in (2.14a), is a primary variable in addition to pressure head. The accumulation term in the mass balance equation (2.7a) is then the derivative of the new variable

$$\frac{\partial M}{\partial t} = -\boldsymbol{\nabla} \cdot \boldsymbol{q}^{\mathrm{w}} + \rho S, \tag{2.31}$$

and an additional algebraic relationship is also imposed, namely

$$M = \rho\theta. \tag{2.32}$$

This particular formulation for Richards' equation is referred to as the modified mixed formulation, or the MPR formulation, throughout this thesis.

PC: The two-variable formulation for the coupled flow and transport model

Similarly to the PR formulation in equation (2.30), which formulates Richards' equation in terms of pressure head in equation, the coupled flow and transport equations (2.7a) and (2.7b) can be formulated in terms of pressure head and concentration alone (Liu et al., 2006, Boufadel et al., 2011). The chain rule is applied to the accumulation term of equation (2.7a)

$$\begin{aligned}\frac{\partial(\rho\theta)}{\partial t} &= \frac{\partial(\rho\theta)}{\partial\psi}\frac{\partial\psi}{\partial t} + \frac{\mathrm{d}(\rho\theta)}{\mathrm{d}c}\frac{\partial c}{\partial t} \\ &= n(\psi, c)\frac{\partial\psi}{\partial t} + \theta\eta\rho_0\frac{\partial c}{\partial t},\end{aligned} \tag{2.33}$$

and to the accumulation term in equation (2.7b):

$$\frac{\partial(c\theta)}{\partial t} = ca(\psi)\frac{\partial\psi}{\partial t} + \theta\frac{\partial c}{\partial t}. \tag{2.34}$$

A detailed derivation of the analytical expressions for the coefficients, such as $n(\psi, c) = \partial(\rho\theta)/\partial\psi$ and $a(\psi) = \mathrm{d}\theta/\mathrm{d}\psi$, of the temporal derivatives in equations (2.33) and (2.34) are given in Appendix A.

Substituting (2.33) and (2.33) into the mass balance equations (2.7a) and (2.7b) gives the pressure–concentration of the coupled flow and transport system

$$n(\psi, c)\frac{\partial \psi}{\partial t} + \theta\eta\rho_0\frac{\partial c}{\partial t} = -\boldsymbol{\nabla} \cdot \boldsymbol{q}^{\mathrm{w}} + \rho S, \tag{2.35a}$$

$$ca(\psi)\frac{\partial \psi}{\partial t} + \quad \theta\frac{\partial c}{\partial t} = -\boldsymbol{\nabla} \cdot \boldsymbol{q}^{\mathrm{s}} + cS. \tag{2.35b}$$

The two variable formulation in (2.35) is referred to as the PC formulation throughout this thesis.

MMPC: The modified mixed formulation for the coupled flow and transport model

The modified mixed formulation for Richards' equation was proposed by Kees and Miller (2002) so that it would be possible to solve the mixed formulation using the method of lines (MOL) in conjunction with higher-order implicit time stepping. In this work, the formulation is extended to the transport model described in equations (2.7).

The MPR formulation for Richards' equation introduced the primary variable M. To extend the mixed formulation to the coupled transport model, we introduce primary variables for both the fluid mass and solute, M and C respectively, defined in equation (2.14). The mass balance equations in (2.7) are formulated explicitly in terms of the new primary variables, along with pressure head and concentration

$$\frac{\partial M}{\partial t} = -\boldsymbol{\nabla} \cdot \boldsymbol{q}^{\mathrm{w}} + \rho S, \tag{2.36a}$$

$$\frac{\partial C}{\partial t} = -\boldsymbol{\nabla} \cdot \boldsymbol{q}^{\mathrm{s}} + cS. \tag{2.36b}$$

These equations are coupled with the following algebraic expressions for mass and concentration

$$M = \rho\theta, \tag{2.37a}$$

$$C = c\theta. \tag{2.37b}$$

This modified mixed formulation for the coupled flow and transport model has four variables and four equations, and is referred to as the MMPC formulation throughout this thesis.

2.5 Conclusions

This chapter introduced a set of governing equations and boundary conditions used to model variably-saturated fluid flow and coupled contaminant transport. The equations were expressed in a form that is amenable to solution using the control volume-finite element spatial discretisation and time stepping scheme that will be introduced in the following chapter.

Chapter 3

Computational Techniques: Control Volume-Finite Element Method with Implicit Time Stepping

Numerical solution of the transient mass balance equations for variably-saturated flow and contaminant transport specified in Chapter 2 involves both spatial and temporal discretisation. This chapter introduces these discretisations, along with their associated numerical techniques, and is organised as follows. In §3.1 the finite element mesh and the dual mesh used in the spatial discretisation are introduced. The control volume-finite element spatial discretisation of the mass balance equations is formulated in §3.2, to give semi-discrete systems of differential algebraic equations by means of the method of lines. The final part of the chapter, §3.3, discusses a higher-order, implicit time stepping scheme that uses an inexact Newton-Krylov method.

3.1 The Mesh

The control volume-finite element (CV-FE) method discretises the continuous mass balance formulations[1] in space. To this end, it employs a discrete geometric representation of the domain and its boundary using a mesh. More precisely, the CV-FE formulation is carried out using a dual mesh, which is itself derived from a finite element mesh. In this section, the finite element mesh is first presented, followed by a description of the dual mesh.

3.1.1 The Finite Element Mesh

An example of a two-dimensional domain comprising two different porous media and a corresponding finite element mesh of triangle and quadrilateral elements is illustrated in Figure 3.1. The finite element mesh discretises the domain Ω into a set of non-overlapping convex polyhedra, called elements, such that

$$\Omega = \bigcup_{j=1}^{n_\epsilon} \epsilon_j, \tag{3.1}$$

where $\mathcal{E} = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_{n_\epsilon}\}$ is the set of all elements, and n_ϵ is the number of elements, in the mesh. The vertices of the elements are called nodes, and the set of all nodes is labelled $\mathcal{N} = \{n_1, n_2, \ldots, n_{n_n}\}$, where n_n is the number of nodes in the mesh. The nodes that form the vertices of each element are connected by one-dimensional line segments, called edges[2]. The set of all edges in the domain is labelled $E = \{e_1, e_2, \ldots, e_{n_e}\}$, where n_e is the number of edges in the domain.

Finite element meshes can be constructed using a wide variety of element types. The software developed in this thesis, called FVMPor, uses the element types illustrated in Figure 3.2: triangle and quadrilateral elements in

[1] The PR and MPR formulations for Richards' equation, and the PC and MMPC formulations for the coupled flow and transport model, as formulated in §2.4.

[2] It is important to note that edges are one-dimensional line segments between adjacent nodes in the mesh in both two and three dimensions.

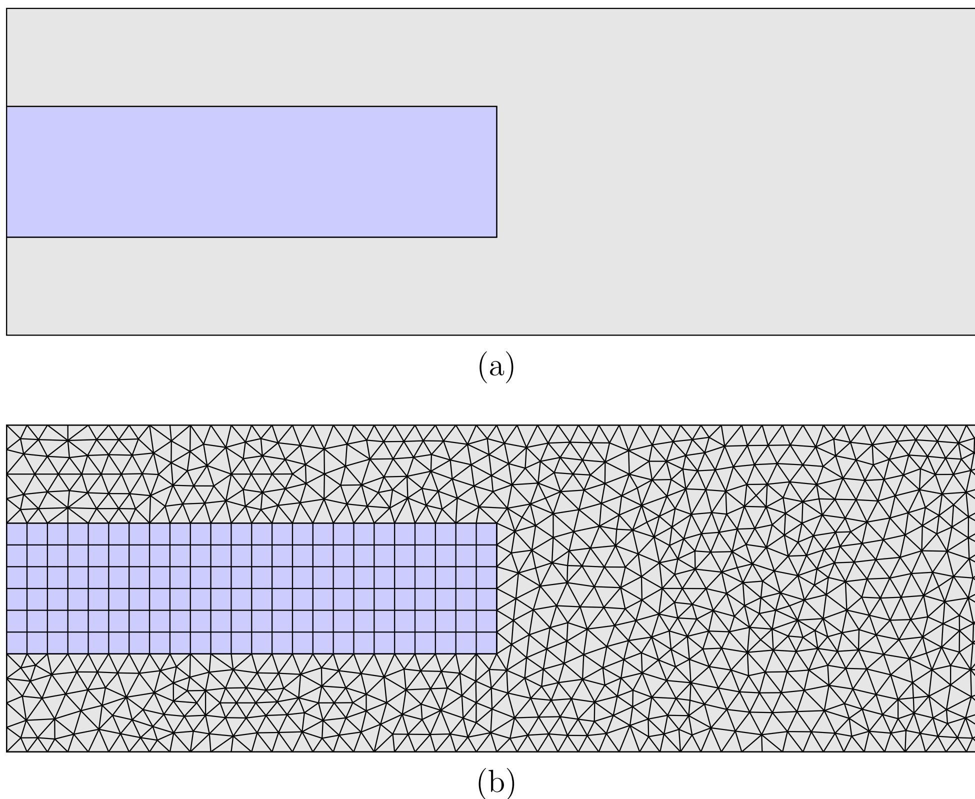


Figure 3.1: (a) A heterogeneous two-dimensional domain, where the colour represents the material. (b) A finite element mesh of the domain, composed of both quadrilaterals and triangles. Material properties are specified on a per-element basis, such that the material properties are uniform in each element.

two dimensions; and tetrahedral and hexahedral elements in three dimensions. These are the most widely-used element types supported by mesh generation software, and can be used to form both structured and unstructured representations of domains.

3.1.2 The Dual Mesh

The CV-FE method is a vertex-centred finite volume method, formulated on a dual mesh composed of control volumes. The first step in constructing the dual mesh is to form sub-control volumes and their faces. In two dimensions,

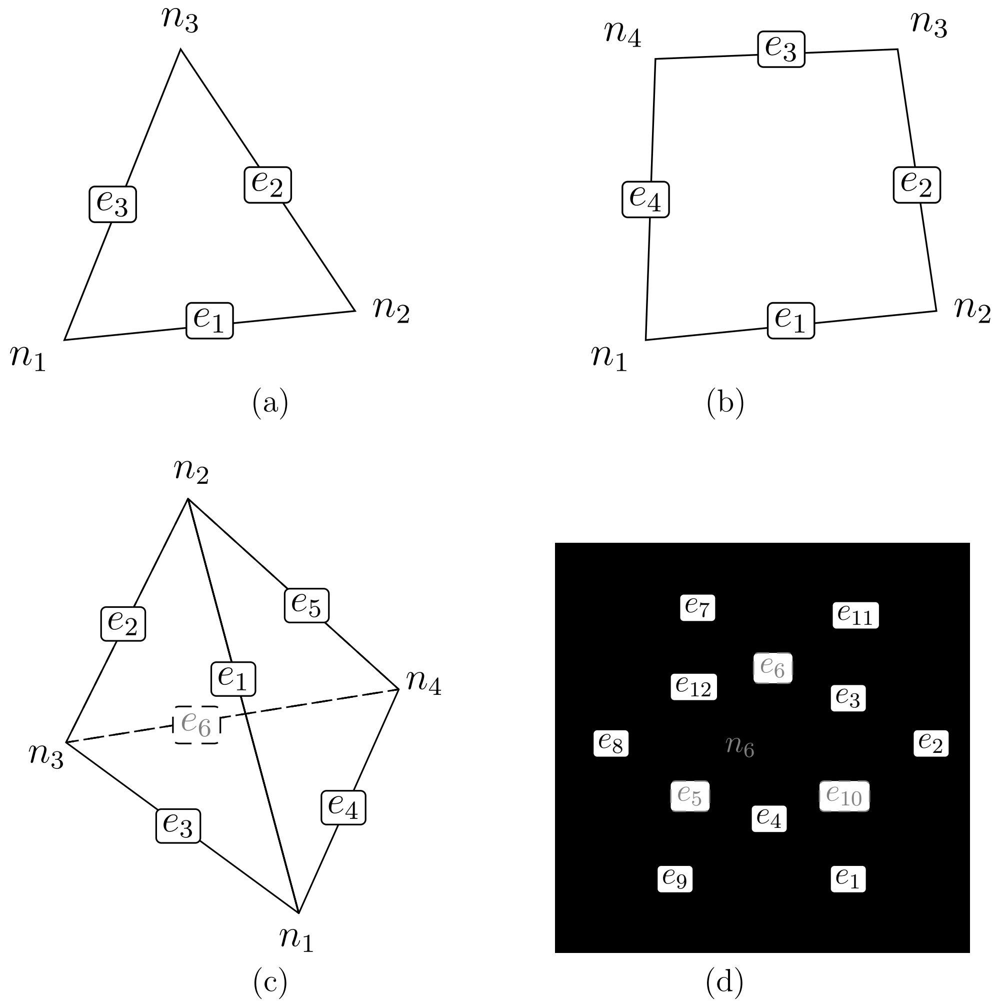


Figure 3.2: Elements with nodes and edges marked. Note that for both two-dimensional and three-dimensional elements, edges are one-dimensional line segments that join two ajacent nodes in an element. (a) A two-dimensional triangle. (b) A two-dimensional quadrilateral. (c) A three-dimensional tetrahedron. (d) A three-dimensional hexahedron.

a sub-control volume is a quadrilateral formed by joining the centroid of an element with the midpoint of each of the element's edges, as illustrated in Figure 3.3(a)–(b). The sub-control volume faces are the line segments that join the element centroid and edge midpoints. In three dimensions, control volume faces are quadrilaterals, formed by joining edge midpoints, face centroids and the centroid of an element, and the resultant sub-control

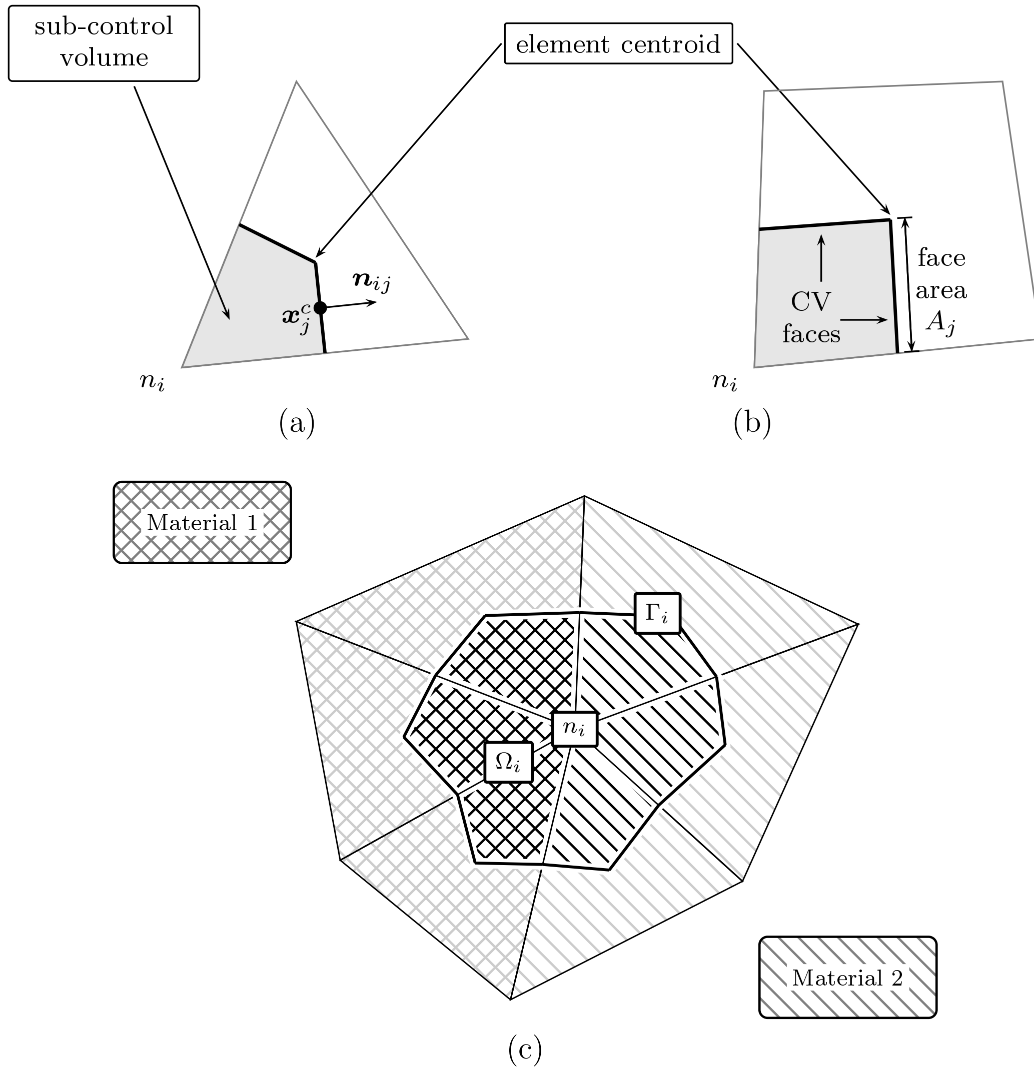


Figure 3.3: Construction of a sub-control volume around a node, n_i, for triangular (a) and quadrilateral (b) elements. Control volume faces are formed by joining the centroid of the element with the midpoint of the element edges, and the centre of face f_j is denoted $\boldsymbol{x}_j^c$ with outward facing normal $\boldsymbol{n}_{ij}$. A control volume Γ_i and its boundary Γ_i are illustrated in (c). Each sub-control volume is assigned the material properties of its element.

volume is a hexahedron as illustrated Figure 3.5. Sub-control volumes that are adjacent to the domain boundary are formed in the same manner as for interior sub-control volumes, with additional faces on the boundary, as illustrated in Figure 3.4.

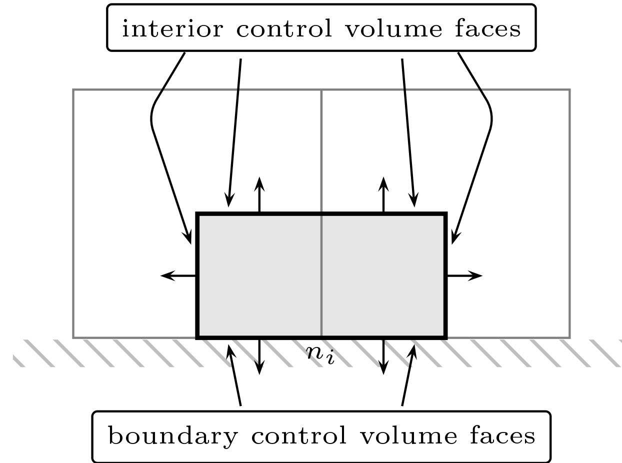


Figure 3.4: Construction of a control volume for a node n_i that lies on the boundary. Extra boundary control volume faces are defined on boundary edges.

A control volume is then formed around each node in the finite element mesh. The control volume for a node is composed of the sub-control volumes that share the node as a vertex. For example, the two-dimensional control volume in Figure 3.3(c) is composed of six sub-control volumes – one from each of the elements that share the node as a vertex. Material properties of the porous media are defined on a per-element basis, so that material properties are constant in each element, and each sub-control volume and control volume face has the same material properties as its parent element. In this manner, material properties are defined in a piecewise constant manner over the volume and surface of each control volume, as illustrated in Figure 3.3(c).

The set $\mathcal{F} = \{f_1, f_2, \ldots, f_{n_f}\}$ is the set of all control volume faces in the dual mesh, where n_f is the number of control volume faces in the dual mesh. The set of interior control volume faces (those that do not lie on the boundary) is denoted $\mathcal{F}_I$, while the set of boundary faces is denoted $\mathcal{F}_B$. The set of all sub-control volumes in the domain is denoted $\mathcal{S} = \{s_1, s_2, \ldots, s_{n_s}\}$, where n_s is the number of sub-control volumes. The sets of control volume faces and sub-control volumes that define the control volume Ω_i and its boundary Γ_i are labelled $\mathcal{F}_i$ and $\mathcal{S}_i$ respectively.

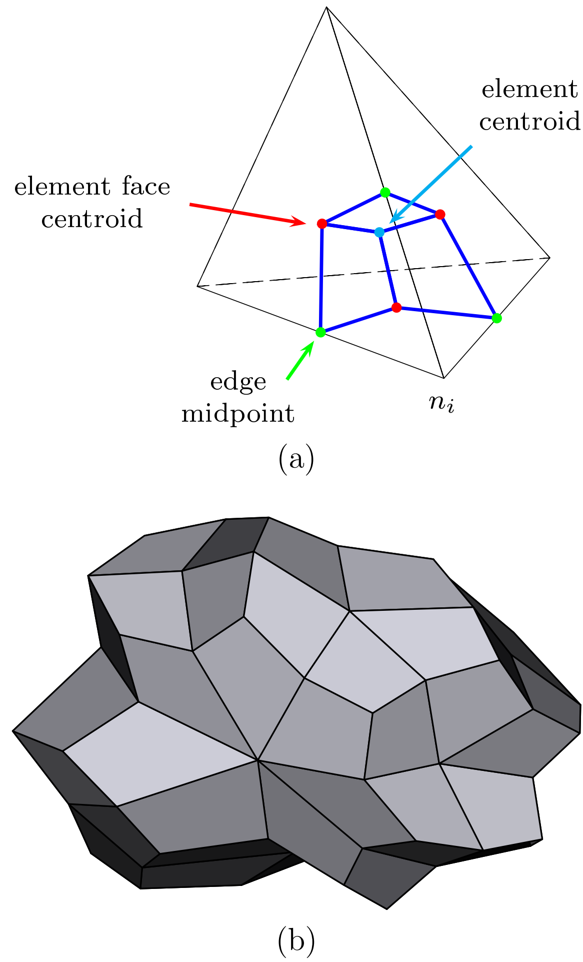


Figure 3.5: (a) Construction of a sub-control volume for a tetrahedral element. The sub-control volume has three quadrilateral faces whose edges are formed by joining the centre of the element, its faces, and its edges. Sub-control volumes have the same hexahedral shape for both tetrahedral and hexahedral three-dimensional meshes. (b) A control volume for an unstructured tetrahedral mesh.

The volume of the control volume Ω_i is the sum of the volumes of its sub-control volumes

$$\Delta_i = \sum_{s_j \in \mathcal{S}_i} \delta_j, \tag{3.2}$$

where Δ_i is the volume of Ω_i, and δ_j is the volume of the sub-control volume s_j. Additionally, the area of the control volume face f_j is A_j. We note that to allow the mesh description to apply in both two and three dimensions,

we have used semantic flexibility when assigning the properties of "volume" and "area" to two-dimensional control volumes and faces respectively (these quantities actually describe area and length.)

An important property of the dual mesh is that the description of sub-control volumes and their faces is independent of the element type in the finite element mesh; in two dimensions sub-control volumes are quadrilaterals with line segment faces, and in three dimensions sub-control volumes are hexahedra with quadrilateral faces. This simplifies the use of mixed meshes, composed of different element types, such as that illustrated in Figure 3.1.

3.1.3 Interpolation

The CV-FE method uses interpolation to reconstruct values of pressure head and concentration, along with their gradients, at the centre of each control volume face. To this end it uses finite element shape functions defined on the finite element mesh (Pepper and Heinrich, 2006).

The nodal shape functions, $N_i(\boldsymbol{x})$, are defined for each node of the element ϵ_ℓ, such that the following constraints are satisfied:

$$N_i(\boldsymbol{x}_j) = \delta_{ij}, \qquad \text{at node } n_j, \tag{3.3a}$$

$$\sum_{k \in \mathcal{N}_{\epsilon_\ell}} N_k(\boldsymbol{x}) = 1, \qquad \text{everywhere in the solution domain,} \tag{3.3b}$$

where $\boldsymbol{x}_j$ is the location of node n_j, δ_{ij} is the Kronecker delta, and $\mathcal{N}_{\epsilon_\ell}$ is the set of node indices for the vertices of the element ϵ_ℓ.

Shape function interpolation seeks to reconstruct a continuous function

$$\varphi(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega, \tag{3.4}$$

given the value of φ at the nodes of the finite element mesh. For each element ϵ_ℓ of the mesh, we define an interpolation function $s_\ell(\boldsymbol{x})$ that reconstructs

the value of φ inside the element as a linear combination of the value of φ at the vertices of the element, with weights given by the shape functions:

$$s_\ell(\boldsymbol{x}) = \sum_k \varphi_k N_k(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in \epsilon_\ell, \tag{3.5}$$

where $\boldsymbol{\varphi} = [\varphi_1, \varphi_2, \ldots, \varphi_{n_n}]^T$ is a vector of the nodal values of φ. In (3.5) the summation is implicitly over the vertices of ϵ_ℓ, that is $k \in \mathcal{N}_\ell$, for brevity of exposition. The interpolation function (3.5) has the important property that it recovers the value of φ exactly at each vertex of ϵ_ℓ by virtue of the constraint in (3.3a)

$$s_\ell(\boldsymbol{x}_i) = \varphi_i, \quad \forall i \in \mathcal{N}_{\epsilon_\ell}, \tag{3.6}$$

Furthermore, the gradient $\boldsymbol{\nabla}\varphi(\boldsymbol{x})$ can then be approximated by taking gradients of the individual shape functions in (3.6):

$$\boldsymbol{\nabla} s_\ell(\boldsymbol{x}) = \sum_k \varphi_k \boldsymbol{\nabla} N_k(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in \epsilon_\ell. \tag{3.7}$$

We will now derive shape function interpolation weights for triangular and tetrahedral elements. The weights for quadrilaterals and hexahedra are given in Appendix C.

Shape function interpolation on triangles

For a triangular element we use the following linear interpolation function

$$s_\ell(\boldsymbol{x}) = \alpha_1 + \alpha_2 x + \alpha_3 y, \tag{3.8}$$

where $\boldsymbol{x} = (x, y)^T$ in two dimensions. By specifying that the interpolation function (3.8) recovers the values of φ at the three vertices of the triangle

according to (3.6), the following system of equations is obtained

$$\begin{aligned} \varphi_1 &= \alpha_1 + \alpha_2 x_1 + \alpha_3 y_1 \\ \varphi_2 &= \alpha_1 + \alpha_2 x_2 + \alpha_3 y_2 \ , \\ \varphi_3 &= \alpha_1 + \alpha_2 x_3 + \alpha_3 y_3 \end{aligned} \tag{3.9}$$

where the subscripts denote a local numbering of coordinates and variables at the vertices of the triangle. The linear system in (3.9) has the following matrix form

$$\begin{bmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} \tag{3.10a}$$

or

$$\boldsymbol{\varphi}_{\epsilon_\ell} = C_{\epsilon_\ell} \boldsymbol{\alpha}_{\epsilon_\ell}, \tag{3.10b}$$

where $\boldsymbol{\varphi}_{\epsilon_\ell}$ is a vector of φ at the vertices of ϵ_ℓ, and $\boldsymbol{\alpha}_{\epsilon_\ell}$ is a vector of the coefficients in the interpolation function (3.8). From (3.8) and (3.10), the value of φ can be reconstructed at a point $\boldsymbol{x} \in \epsilon_\ell$ as follows

$$s_{\epsilon_\ell}(\boldsymbol{x}) = \begin{bmatrix} 1 & x & y \end{bmatrix} C_{\epsilon_\ell}^{-1} \boldsymbol{\varphi}_{\epsilon_\ell}. \tag{3.11}$$

This defines the shape function weights

$$\boldsymbol{N}(\boldsymbol{x}) = \begin{bmatrix} 1 & x & y \end{bmatrix} C_{\epsilon_\ell}^{-1}, \tag{3.12}$$

where $\boldsymbol{N}(\boldsymbol{x})$ is a vector of the shape function weights. Writing the weights as a vector in (3.12) makes it possible to express the summation in (3.5) as a dot product

$$s_\ell(\boldsymbol{x}) = \boldsymbol{N}(\boldsymbol{x}) \cdot \boldsymbol{\varphi}_{\epsilon_\ell} \quad \forall \boldsymbol{x} \in \epsilon_\ell. \tag{3.13}$$

It is interesting to note that the shape function weights in (3.12) are dependent only on the geometry of the element ϵ_ℓ, and not the value of the

function φ. This means that the same weights can be used to interpolate different variables (for example pressure head and concentration in the models considered in this thesis.)

The partial derivatives of φ can be found by differentiating (3.12)

$$\frac{\partial \boldsymbol{N}}{\partial x} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} C_{\epsilon_\ell}^{-1} \tag{3.14a}$$

$$\frac{\partial \boldsymbol{N}}{\partial y} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} C_{\epsilon_\ell}^{-1}, \tag{3.14b}$$

which are constant in ϵ_ℓ, and as such the gradient may not be continuous at the interface between elements. However, the CV-FE discretisation uses shape functions to determine gradients at the centroid of control volume faces, where the reconstructed gradient is continuous.

Shape function interpolation on tetrahedra

For a tetrahedron the following linear interpolation function is used

$$s_\ell(\boldsymbol{x}) = \alpha_1 + \alpha_2 x + \alpha_3 y + \alpha_4 z, \tag{3.15}$$

where $\boldsymbol{x} = (x, y, z)^{\mathrm{T}}$ in three dimensions. The coefficient matrix in this case is

$$C_{\epsilon_\ell} = \begin{bmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{bmatrix}, \tag{3.16}$$

and the shape functions and their derivatives are defined by

$$\boldsymbol{N}(\boldsymbol{x}) = \begin{bmatrix}1 & x & y & z\end{bmatrix} C_{\epsilon_\ell}^{-1} \tag{3.17a}$$

$$\frac{\partial \boldsymbol{N}}{\partial x} = \begin{bmatrix}0 & 1 & 0 & 0\end{bmatrix} C_{\epsilon_\ell}^{-1} \tag{3.17b}$$

$$\frac{\partial \boldsymbol{N}}{\partial y} = \begin{bmatrix}0 & 0 & 1 & 0\end{bmatrix} C_{\epsilon_\ell}^{-1} \tag{3.17c}$$

$$\frac{\partial \boldsymbol{N}}{\partial z} = \begin{bmatrix}0 & 0 & 0 & 1\end{bmatrix} C_{\epsilon_\ell}^{-1} \tag{3.17d}$$

where, as was the case for triangles, the gradient weights are constant, resulting in a piecewise constant gradient reconstruction.

3.2 The Control Volume-Finite Element Method

The CV-FE discretisation proceeds by integrating the mass balance laws for the fluid and solute, equations (2.7a) and (2.7b) respectively, over each control volume

$$\int_{\Omega_i} \frac{\partial(\rho\theta)}{\partial t}\,\mathrm{d}V + \int_{\Omega_i} \boldsymbol{\nabla}\cdot\boldsymbol{q}^w\,\mathrm{d}V = \int_{\Omega_i} \rho S\,\mathrm{d}V, \tag{3.18a}$$

$$\int_{\Omega_i} \frac{\partial(c\theta)}{\partial t}\,\mathrm{d}V + \int_{\Omega_i} \boldsymbol{\nabla}\cdot\boldsymbol{q}^s\,\mathrm{d}V = \int_{\Omega_i} cS\,\mathrm{d}V. \tag{3.18b}$$

Gauss' divergence theorem is applied to the integrals of the flux terms in (3.18) to express them as surface integrals

$$\int_{\Omega_i} \boldsymbol{\nabla}\cdot\boldsymbol{q}^\alpha\,\mathrm{d}V = \int_{\Gamma_i} \boldsymbol{q}^\alpha\cdot\boldsymbol{n}\,\mathrm{d}\sigma, \tag{3.19}$$

where $\boldsymbol{n}$ is the unit outward-facing normal to the control volume surface Γ_i. Equations (3.18a) and (3.18b) can be represented in the following general

form

$$\overbrace{\underbrace{\int_{\Omega_i} \frac{\partial X^\alpha}{\partial t}\,\mathrm{d}V}_{\text{accumulation term}}}^{\text{volume integral}} + \overbrace{\underbrace{\int_{\Gamma_i} \boldsymbol{q}^\alpha \cdot \boldsymbol{n}\,\mathrm{d}\sigma}_{\text{surface flux}}}^{\text{surface integral}} = \overbrace{\underbrace{\int_{\Omega_i} S^\alpha\,\mathrm{d}V}_{\text{source term}}}^{\text{volume integral}}, \tag{3.20}$$

where X^α is the conserved quantity ($X^w = M = \rho\theta$ for the water, and $X^s = C = c\theta$ for salt), $\boldsymbol{q}^\alpha$ is the flux ($\boldsymbol{q}^w$ and $\boldsymbol{q}^s$ in equation (2.6)), and S^α is the appropriate source term ($S^w = \rho S$ and $S^s = cS$).

The finite volume method seeks to approximate the integrals in the conservation equation (3.20) using the values of the variables at the nodes, and information provided by boundary conditions for control volumes that lie on the boundary (Ferziger and Perić, 2002).

Volume averaging

Before discussing the discretisation of the accumulation and source terms, the concept of a volume average must be introduced. The volume average of a quantity γ is the average value of γ over a control volume, which has the integral definition:

$$\overline{\gamma}_i = \frac{1}{\Delta_i}\int_{\Omega_i} \gamma\,\mathrm{d}V, \tag{3.21}$$

where Δ_i is the volume of the control volume Ω_i. In the most general case, γ is a function of pressure head, concentration and material properties $\gamma(\psi, c; \mathcal{P})$, where $\mathcal{P}$ denotes the material properties. The volume average in (3.21) can be approximated in terms of nodal values for pressure head and concentration:

$$\overline{\gamma}_i \approx \frac{1}{\Delta_i}\int_{\Omega_i} \gamma(\psi_i, c_i; \mathcal{P})\,\mathrm{d}V. \tag{3.22}$$

If γ is not dependent on material properties (as is the case for density ρ, which is only dependent on concentration and pressure head), the approximation

to the volume average in (3.22) simplifies to

$$\begin{aligned}\overline{\gamma}_i &\approx \gamma(\psi_i, c_i) \\ &= \gamma_i,\end{aligned} \tag{3.23}$$

where the value of γ at the node is taken as the volume average.

If γ is dependent on material properties (for example moisture content θ), the approximation to the volume average in (3.22) is a weighted sum of the average value in each of its constituent sub-control volumes

$$\begin{aligned}\overline{\gamma}_i &= \frac{1}{\Delta_i} \sum_{s_j \in \mathcal{S}_i} \int_{s_j} \gamma \, \mathrm{d}V \\ &\approx \frac{1}{\Delta_i} \sum_{s_j \in \mathcal{S}_i} \delta_j \gamma(\psi_i, c_i; \mathcal{P}_j),\end{aligned} \tag{3.24}$$

where δ_j is the volume of sub-control volume s_j, and $\gamma(\psi_i, c_i; \mathcal{P}_j)$ is a function of pressure head and concentration at node n_i and the material properties defined for the sub-control volume s_j. In the heterogeneous case, the value of $\gamma(\psi_i, c_i; \mathcal{P}_j)$ is not defined at the node, because material properties $\mathcal{P}$ are discontinuous at that point, and the volume average is used instead to define the variable at the node

$$\gamma_i \equiv \frac{1}{\Delta_i} \sum_{s_j \in \mathcal{S}_i} \delta_j \gamma(\psi_i, c_i; \mathcal{P}_j). \tag{3.25}$$

3.2.1 Accumulation Terms

The treatment of the accumulation terms is dependent on the formulation. First, we consider the modified mixed formulations: the MPR and MMPC formulations for Richards' equation and the coupled transport model respectively. For both of these formulations, the temporal derivative can be taken

outside the integral

$$\int_{\Omega_i} \frac{\partial X^\alpha}{\partial t} \, \mathrm{d}V = \frac{\mathrm{d}}{\mathrm{d}t} \int_{\Omega_i} X^\alpha \, \mathrm{d}V, \tag{3.26}$$

to which we apply the volume average definition in (3.21)

$$\int_{\Omega_i} \frac{\partial X^\alpha}{\partial t} \, \mathrm{d}V = \Delta_i \frac{\mathrm{d}}{\mathrm{d}t} \overline{X_i^\alpha}. \tag{3.27}$$

We define a primary variable for the discretised system X_i as the volume average as per (3.24)

$$X_i^\alpha = \frac{1}{\Delta_i} \sum_{j \in \mathcal{S}_i} \delta_j X^\alpha(\psi_i, c_i; \mathcal{P}_j); \tag{3.28}$$

specifically, we define the primary variables M_i and C_i:

$$M_i = X_i^w = \frac{\rho_i}{\Delta_i} \sum_{j \in \mathcal{S}_i} \delta_j \theta(\psi_i; \mathcal{P}_j) \tag{3.29}$$

$$C_i = X_i^s = \frac{c_i}{\Delta_i} \sum_{j \in \mathcal{S}_i} \delta_j \theta(\psi_i; \mathcal{P}_j). \tag{3.30}$$

Then, the accumulation terms for the modified mixed formulation in equations (3.18a) and (3.18b) can be expressed as derivatives of the primary variables:

$$\frac{1}{\Delta_i} \int_{\Omega_i} \frac{\partial (\rho\theta)}{\partial t} \, \mathrm{d}V \approx \frac{\mathrm{d}M_i}{\mathrm{d}t} \tag{3.31a}$$

$$\frac{1}{\Delta_i} \int_{\Omega_i} \frac{\partial (c\theta)}{\partial t} \, \mathrm{d}V \approx \frac{\mathrm{d}C_i}{\mathrm{d}t}. \tag{3.31b}$$

The discretisation of the accumulation terms is different for the reduced PR and PC formulations, where the conserved quantities X^α are not primary variables. The accumulation term for the PR formulation in equation (2.30),

is partitioned into the contributions from each sub-control volume

$$\int_{\Omega_i} n(\psi;\mathcal{P})\frac{\partial \psi}{\partial t}\,\mathrm{d}V = \sum_{s_k\in\mathcal{S}_i}\int_{s_k} n(\psi;\mathcal{P}_k)\frac{\partial \psi}{\partial t}\,\mathrm{d}V.$$

The accumulation term is then approximated by using the nodal pressure head value ψ_i over the control volume:

$$\begin{aligned}\int_{\Omega_i} n(\psi;\mathcal{P})\frac{\partial \psi}{\partial t}\,\mathrm{d}V &\approx \sum_{s_k\in\mathcal{S}_i}\int_{s_k} n(\psi_i;\mathcal{P}_k)\frac{\partial \psi_i}{\partial t}\,\mathrm{d}V\\ &= \sum_{s_k\in\mathcal{S}_i}\delta_k n(\psi_i;\mathcal{P}_k)\frac{\mathrm{d}\psi_i}{\mathrm{d}t}.\end{aligned}$$

This is in the form of the volume average in (3.25), so that the discretised accumulation term is

$$\int_{\Omega_i} n(\psi;\mathcal{P})\frac{\partial \psi}{\partial t}\,\mathrm{d}V \approx \Delta_i n_i\frac{\mathrm{d}\psi_i}{\mathrm{d}t}, \tag{3.32}$$

where we note that the volume average of the storage term n_i is determined using equation (3.25).

Similarly, for the PC formulation, the discrete form of the accumulation term for the fluid mass balance equation (2.35a) is

$$\int_{\Omega_i}\left(n(\psi,c)\frac{\partial \psi}{\partial t} + \eta\rho_0\theta\frac{\partial c}{\partial t}\right)\mathrm{d}V \approx \Delta_i\left(n_i\frac{\mathrm{d}\psi_i}{\mathrm{d}t} + \eta\rho_0\theta_i\frac{\mathrm{d}c_i}{\mathrm{d}t}\right), \tag{3.33a}$$

and for the solute conservation law (2.35b)

$$\int_{\Omega_i}\left(ca(\psi)\frac{\partial \psi}{\partial t} + \theta\frac{\partial c}{\partial t}\right)\mathrm{d}V \approx \Delta_i\left(c_i a_i\frac{\mathrm{d}\psi_i}{\mathrm{d}t} + \theta_i\frac{\mathrm{d}c_i}{\mathrm{d}t}\right). \tag{3.33b}$$

The volume average of a_i, n_i and θ_i are dependent on material properties, and are determined using equation (3.25). The nodal value of concentration, c_i, is used for the volume average, and the parameters η and ρ_0 are independent of material properties.

3.2.2 Source Terms

The volume integrals of the source terms are independent of material properties, and can be approximated as

$$\frac{1}{\Delta_i}\int_{\Omega_i} \rho S \,\mathrm{d}V = \overline{(\rho S)_i} \approx \rho_i^* S_i, \tag{3.34a}$$

$$\frac{1}{\Delta_i}\int_{\Omega_i} c S \,\mathrm{d}V = \overline{(c S)_i} \approx c_i^* S_i, \tag{3.34b}$$

where S_i is the volume average of the volumetric source term for water, and ρ_i^* and c_i^* are selected dependent on whether there is a source or sink, for example with the concentration:

$$c_i^* = \begin{cases} c_i & S_i < 0, \\ \text{c of source fluid} & S_i \geq 0. \end{cases} \tag{3.35}$$

3.2.3 Surface Fluxes

The surface integral of flux over the control volume surface in (3.20) is the sum of contributions from individual control volume faces

$$\int_{\Gamma_i} \boldsymbol{q}^\alpha \cdot \boldsymbol{n} \,\mathrm{d}\sigma = \sum_{f_j \in \mathcal{F}_i} \int_{f_j} \boldsymbol{q}^\alpha \cdot \boldsymbol{n}_{ij} \,\mathrm{d}\sigma, \tag{3.36}$$

where $\boldsymbol{n}_{ij}$ is the outward-facing normal to the face f_j. The integral over each face in (3.36) is approximated using the midpoint rule

$$\int_{f_j} \boldsymbol{q}^\alpha \cdot \boldsymbol{n}_{ij} \,\mathrm{d}\sigma \approx A_j \left[\boldsymbol{q}^\alpha\right]_j \cdot \boldsymbol{n}_{ij}, \tag{3.37}$$

where $\left[\boldsymbol{q}^\alpha\right]_j$ is the flux at $\boldsymbol{x}_j^c$ the barycentre of face f_j.

At this point we introduce some new notation to distinguish between quantities that are known at $\boldsymbol{x}_j^c$, and quantities that must be approximated from

variables defined at nodes:

$$\begin{aligned} &\lambda_j, && \lambda \text{ known at } \boldsymbol{x}_j^c, \\ &[\lambda]_j\,, && \lambda \text{ approximated at } \boldsymbol{x}_j^c. \end{aligned} \tag{3.38}$$

This notation was used when specifying the midpoint approximation of the flux integral in equation (3.37), where the face area A_j is known, and the flux $[\boldsymbol{q}^\alpha]_j$ is approximated.

The functional forms of the fluxes defined in equations (2.6a)–(2.6b), are repeated here for reference

$$\text{fluid mass flux:} \qquad \boldsymbol{q}^w = \rho \boldsymbol{q}, \tag{3.39a}$$

$$\text{solute flux:} \qquad \boldsymbol{q}^s = c\boldsymbol{q} - \theta \mathsf{D}_h \boldsymbol{\nabla} c, \tag{3.39b}$$

where $\boldsymbol{q}$ is the Darcy flux

$$\boldsymbol{q} = -k_{\text{rw}} \mathsf{K} \left(\boldsymbol{\nabla} \psi + \hat{\rho} \boldsymbol{\nabla} z \right). \tag{3.40}$$

To approximate the fluid and solute fluxes in equation (3.39) at each control volume face f_j, the value and gradient of pressure head and concentration, secondary variables, and parameters associated with the material properties must be determined at the centroid, $\boldsymbol{x}_j^c$, of the face.

First, we consider parameters associated with material properties of the porous medium. The material properties are defined in a piecewise-constant manner, with discontinuities at the interface between elements. Each face centroid $\boldsymbol{x}_j^c$ is on the interior of an element, where material properties are continuous and may expressed exactly using the notation in (3.38) (for example, the hydraulic conductivity K_j). This is an advantage of the CV-FE method over finite element and cell-centred finite volume methods, which evaluate fluxes at element edges, where material properties may be discontinuous. Some form of averaging is required to determine the material properties for these methods, which can have a considerable negative impact on the convergence of the chosen numerical scheme (Miller et al., 1998).

The gradient of pressure head and concentration is reconstructed using the nodal shape function interpolation function defined in (3.6)

$$[\boldsymbol{\nabla}\psi]_j = \sum_k \psi_k \boldsymbol{\nabla} N_k(\boldsymbol{x}_j^c), \tag{3.41a}$$

$$[\boldsymbol{\nabla}c]_j = \sum_k c_k \boldsymbol{\nabla} N_k(\boldsymbol{x}_j^c). \tag{3.41b}$$

The shape function interpolation function defined in (3.5) is also used to approximate the scaled density in the buoyancy term, $\hat{\rho} = \rho/\rho_0$

$$[\hat{\rho}]_j = \frac{1}{\rho_0} \sum_k \rho_k N_k(\boldsymbol{x}_j^c). \tag{3.42}$$

Given the material properties, the pressure head gradient in (3.41a) and the density in the buoyancy term in (3.42), the Darcy flux at face f_j can be written

$$[\boldsymbol{q}]_j = -\left[k_{\mathrm{rw}}\right]_j \mathsf{K}_j \left(\sum_k \psi_k \boldsymbol{\nabla} N_k(\boldsymbol{x}_j^c) + \frac{1}{\rho_0} \left(\sum_k \rho_k N_k(\boldsymbol{x}_j^c) \right) \boldsymbol{\nabla} z \right), \tag{3.43}$$

where the relative permeability $[k_{\mathrm{rw}}]_j$ is to be determined using edge-based weighting discussed later in this section.

We note that the approximation of the Darcy flux in equation (3.43) is consistent by virtue of both the material properties and the gradient being continuous at the point $\boldsymbol{x}_j^c$. The consistent flux approximations make the CV-FE discretisation mass conservative (Martinez, 2006).

Dispersion terms

The dispersion tensor D_h defined in (2.15)–(2.18) is a function of average fluid velocity $\boldsymbol{v}$. In equation (3.39b) the dispersion tensor is scaled by moisture content. The resultant tensor, $\theta\mathsf{D}_h$, can be expressed in terms of the Darcy

flux by recalling the relationship between flux and velocity in (2.17), namely $\boldsymbol{q} = \theta \boldsymbol{v}$:

$$\theta \mathsf{D}_h = \theta D_m \mathsf{T} + (\alpha_L - \alpha_T) \frac{\boldsymbol{q}\boldsymbol{q}^T}{\| \boldsymbol{q} \|_2} + \alpha_T \| \boldsymbol{q} \|_2 \mathsf{I},$$

which is approximated at the face f_j

$$[\theta \mathsf{D}_h]_j = [\theta]_j D_{mj} \mathsf{T}_j + (\alpha_{Lj} - \alpha_{Tj}) \left[\frac{\boldsymbol{q}\boldsymbol{q}^T}{\| \boldsymbol{q} \|_2} \right]_j + \alpha_{Tj} \left\| [\boldsymbol{q}]_j \right\|_2 \mathsf{I}. \tag{3.44}$$

Turner and Perré (2001) investigated harmonic and arithmetic averaging for approximating the moisture content $[\theta]_j$, and Truscott (2004) used shape functions. They found that there was little difference between these approaches, so in this work we use the arithmetic average

$$[\theta]_j = \frac{1}{2} \left([\theta]_j^{\text{front}} + [\theta]_j^{\text{back}} \right), \tag{3.45}$$

where $[\theta]_j^{\text{front}}$ and $[\theta]_j^{\text{back}}$ are the moisture contents determined at the front and back nodes of the edge to which face f_j is attached[3], as illustrated in Figure 3.6.

Advection and mobility terms

Careful treatment of advection terms is essential to obtain physically realistic numerical solutions. There are three such terms in the flow and transport models considered in this thesis: the mobility term $[k_{\text{rw}}]_j$, and the advected variables $[\rho]_j$ and $[c]_j$. If shape function interpolation is used to determine these values at control volume faces, as they are for the gradients and the buoyancy term in (3.41) and (3.42), the solution can exhibit physically unrealistic, non-monotone behaviour (Patankar, 1980, Turner and Perré, 2001, Truscott, 2004). *Edge-based* spatial weighting schemes, such as upstream weighting and flux limiting, are used instead to ensure physically realistic

[3]This approach is valid in both two and three dimensions, because in both cases edges are one-dimensional line segments.

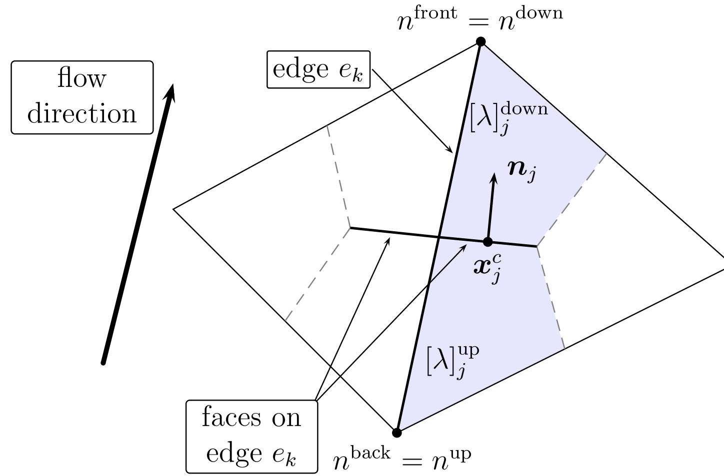


Figure 3.6: An edge and the control volume faces attached to it in a two-dimensional mesh. The edge has a front and back node, which are assigned as upstream or downstream according to the net flow along the edge.

solutions.

In §3.1.2 control volume faces were constructed by joining the centroid of an element to the midpoint of one of the element's edges, such that each control volume face is attached to only one edge. This is illustrated for a two-dimensional mesh in Figure 3.6, where two control volume faces are attached to each edge[4].

Edge-based methods take the weighted sum of the value at the front and back nodes of the edge to which the face f_j is attached, so that the value of a variable λ constructed at a control volume face is

$$[\lambda]_j = w_k^{\mathrm{front}} [\lambda]_j^{\mathrm{front}} + w_k^{\mathrm{back}} [\lambda]_j^{\mathrm{back}}, \tag{3.46}$$

with edge weights w_j^{front} and w_j^{back}, and $[\lambda]_j^{\mathrm{front}}$ and $[\lambda]_j^{\mathrm{back}}$ are the values of λ

[4]The process is more complicated in three dimensions (see Figure 3.5(a)), where the quadrilateral control volume face is formed by joining the element centroid with an edge midpoint and the two element faces adjacent to the edge. However, as for the two-dimensional case, each control volume face is attached to only one edge.

at the front and back nodes[5].

Flow direction indicators

In this thesis three edge-based spatial weighting schemes are investigated: central weighting, upstream weighting and flux limiting. The upstream and flux limiting methods use information about the direction of flow along each edge. To this end, a flow direction indicator (FDI) is employed to indicate whether the direction of flow is from the front node to the back node, or *vice-versa*.

The FDI is a scalar quantity computed for each edge, the sign of which indicates the direction of flow along the edge as follows:

$$\begin{array}{lll} \mathrm{FDI}_k > 0 & : & \begin{cases} \text{flow from back to front} \\ n^{\mathrm{up}} = n^{\mathrm{back}} \quad \text{and} \quad n^{\mathrm{down}} = n^{\mathrm{front}} \end{cases} \\ \mathrm{FDI}_k < 0 & : & \begin{cases} \text{flow from front to back} \\ n^{\mathrm{up}} = n^{\mathrm{front}} \quad \text{and} \quad n^{\mathrm{down}} = n^{\mathrm{back}} \end{cases} \end{array}, \tag{3.47}$$

where FDI_k is the FDI for edge e_k. One possible choice of indicator is the net flux of fluid over the faces attached to the edge e_k

$$\mathrm{FDI}_k(\boldsymbol{q}) = \sum_{f_j \in \mathcal{F}_{e_k}} A_j \, [\boldsymbol{q}]_j \cdot \boldsymbol{n}_j, \tag{3.48}$$

where $\mathcal{F}_{e_k}$ is the set of control volume faces on the edge, and the normal $\boldsymbol{n}_j$ to the face f_j is chosen to point from the back to the front nodes of the edge, as shown in Figure 3.6.

It is not possible to compute the flow direction indicator $\mathrm{FDI}(\boldsymbol{q})$ using up to date information, because the relative permeability $[k_{\mathrm{rw}}]_j$ – used to de-

[5]The subscript j is used instead of subscript k on the front and back values, for example $[\lambda]_j^{\mathrm{front}}$ because the value of λ may be dependent on material properties, which are uniquely defined for each face, such that two faces that share an edge may have different material properties

termine the flux $[\boldsymbol{q}]_j$ in (3.43) – is itself dependent on the value of FDI($\boldsymbol{q}$). To overcome this, Turner and Perré (2001) and Truscott (2004) use the $[\boldsymbol{q}]_j$ computed at the previous Newton iteration, however this value is not available in the software library used for time stepping in this thesis[6]. Instead, to determine the value of $[\boldsymbol{q}]_j$ for finding the FDI in (3.48), arithmetic averaging is used to determine the relative permeability at f_j, namely

$$[k_{\mathrm{rw}}]_j = \frac{1}{2}\left([k_{\mathrm{rw}}]_j^{\mathrm{front}} + [k_{\mathrm{rw}}]_j^{\mathrm{back}}\right), \tag{3.49}$$

which is equivalent to using central weighting (see equation (3.54)).

Truscott (2004) proposed several alternative edge-based flow direction indicators, two of which we will investigate in this work. Both indicators are based on the water phase potential φ, whose gradient is defined

$$\boldsymbol{\nabla}\varphi = \boldsymbol{\nabla}\psi + \hat{\rho}\boldsymbol{\nabla} z. \tag{3.50}$$

The first indicator uses the gradient of the phase potential

$$\mathrm{FDI}_k(\boldsymbol{\nabla}\varphi) = \sum_{j\in\mathcal{F}_{e_k}} -A_j\,[\boldsymbol{\nabla}\varphi]_j\cdot\boldsymbol{n}_j, \tag{3.51}$$

where $[\boldsymbol{\nabla}\varphi]_j$ is determined using shape function interpolation. The second proposed indicator uses the difference in potential between the back and front nodes of the edge

$$\mathrm{FDI}_k(\varphi) = \varphi_{e_k}^{\mathrm{front}} - \varphi_{e_k}^{\mathrm{back}}. \tag{3.52}$$

To determine the potential at the nodes in (3.52), it is assumed that the ratio of density and the reference density is unity, that is $\hat{\rho} = \rho/\rho_0 \approx 1$, such that

$$\varphi = \psi + z, \tag{3.53}$$

which is the *hydraulic head*, H. FDI(φ) is only used for Richards' equation

[6]The IDA library is used for time stepping in this project, details of which are given in §3.3.

because the assumption that $\hat{\rho} = 1$ is reasonable in the absence of salt, where the dependence of density on pressure head is very weak[7]. However, for density driven flows due to salinity the assumption in (3.53) does not hold, giving erroneous results for the full transport model.

Central weighting and upstream weighting

Central weighting reconstructs a value λ at the face f_j by taking an average of the value at the front and back nodes of the edge to which the face is attached

$$[\lambda]_j = \frac{1}{2}\left([\lambda]_j^{\text{back}} + [\lambda]_j^{\text{front}}\right), \tag{3.54}$$

which is equivalent to setting the edge weights in equation (3.46) to $w_k^{\text{front}} = w_k^{\text{back}} = 1/2$. Because the edge weights are constant, the central weighting method does not use flow direction information. However, central weighting requires the use of fine meshes to avoid non-physical oscillations in the solution of both Richards' equation (Forsyth and Kropinski, 1997) and advection-dominated contaminant transport (Neumann et al., 2011).

Upstream weighting uses the value of the variable defined at the upstream node as the representative value at the control volume face f_j

$$[\lambda]_j = [\lambda]_j^{\text{up}}, \tag{3.55}$$

where the upstream point is chosen according to the flow direction indicator as in (3.47). Upstream weighting is guaranteed to give monotone solutions, however these solutions can exhibit excessive numerical diffusion and smearing, particularly on coarse meshes (Patankar, 1980).

[7]The parameter β in equation (2.11) is typically very small.

Flux limiting

Flux limiting is a more general method for determining the edge weights in equation (3.46), which has been shown to produce oscillation-free solutions with sharp saturation fronts for coarse meshes in the solution of multiphase porous flows (Forsyth et al., 1996, Unger et al., 1996, Turner and Perré, 2001, Perré and Turner, 2002, Truscott, 2004).

Flux limiting reconstructs a value λ at the control volume face as a weighted average of front and back values

$$[\lambda]_j = [\lambda]_j^{\text{up}} + \frac{\sigma(r)}{2}\left([\lambda]_j^{\text{down}} - [\lambda]_j^{\text{up}}\right), \tag{3.56}$$

where r is the smoothness sensor and σ is the limiter function.

The smoothness sensor r is calculated using information from the upstream and downstream nodes, as well as information from a second upstream (2up) node. The 2up node can be determined using a *geometric* approach based on the direction of flow, or by the *maximum potential* approach that chooses the 2up node as the neighbouring node of the upstream node with the maximum flow into the upstream node, as illustrated in Figure 3.7. Both geometric and maximum potential approaches were investigated by Forsyth et al. (1996) for two-phase flow. The maximum potential method was found to be more accurate and efficient than the geometric approach, and easier to implement on unstructured meshes. For these reasons, the maximum potential method is used here.

There are various methods available for determining the smoothness sensor, and here we consider three approaches for the mobility term, and one for the advection terms. For the mobility term, the choice of smoothness sensor is based on the flow direction indicator (Truscott, 2004). The first sensor for

the mobility term is based on FDI($\boldsymbol{q}$), and takes the ratio of net flux

$$r = \frac{\sum_{j \in \mathcal{F}_{e2\mathrm{up}}} A_j \, [\boldsymbol{q}]_j \cdot \boldsymbol{n}_j}{\sum_{j \in \mathcal{F}_{ek}} A_j \, [\boldsymbol{q}]_j \cdot \boldsymbol{n}_j}. \tag{3.57}$$

The second sensor, based on FDI(φ), takes the ratio of the difference between the difference in potential

$$r = \frac{\varphi^{2\mathrm{up}} - \varphi^{\mathrm{up}}}{\| \boldsymbol{x}^{2\mathrm{up}} - \boldsymbol{x}^{\mathrm{up}} \|} \Big/ \frac{\varphi^{\mathrm{up}} - \varphi^{\mathrm{down}}}{\| \boldsymbol{x}^{\mathrm{up}} - \boldsymbol{x}^{\mathrm{down}} \|}. \tag{3.58}$$

The third sensor for the mobility term is based on FDI($\boldsymbol{\nabla}\phi$), and is found by replacing the flux in (3.57) with the gradient of the potential

$$r = \frac{\sum_{j \in \mathcal{F}_{e2\mathrm{up}}} A_j \, [\boldsymbol{\nabla}\varphi]_j \cdot \boldsymbol{n}_j}{\sum_{j \in \mathcal{F}_{ek}} A_j \, [\boldsymbol{\nabla}\varphi]_j \cdot \boldsymbol{n}_j}. \tag{3.59}$$

The sensor for the advection term takes the ratio of concentration values

$$r = \frac{c^{2\mathrm{up}} - c^{\mathrm{up}}}{\| \boldsymbol{x}^{2\mathrm{up}} - \boldsymbol{x}^{\mathrm{up}} \|} \Big/ \frac{c^{\mathrm{up}} - c^{\mathrm{down}}}{\| \boldsymbol{x}^{\mathrm{up}} - \boldsymbol{x}^{\mathrm{down}} \|}. \tag{3.60}$$

The limiter function, $\sigma(r)$ in equation (3.56), is selected such that the spatial weighting is total variation diminishing (TVD), which implies that the solution is non-osciliatory (van Leer, 1974). Furthermore, for the scheme to be second order accurate, the limiter must be chosen as a weighted convex average of a centrally-weighted scheme and a two-point upwind weighted scheme (Sweby, 1984). Here we investigate two limiter functions that satisfy these criteria: the van Leer limiter (van Leer, 1979) and the parabolic limiter (Arminjon and Dervieux, 1993).

The van Leer flux limiter has the following form

$$\sigma(r) = \frac{2r}{1+r}, \tag{3.61}$$

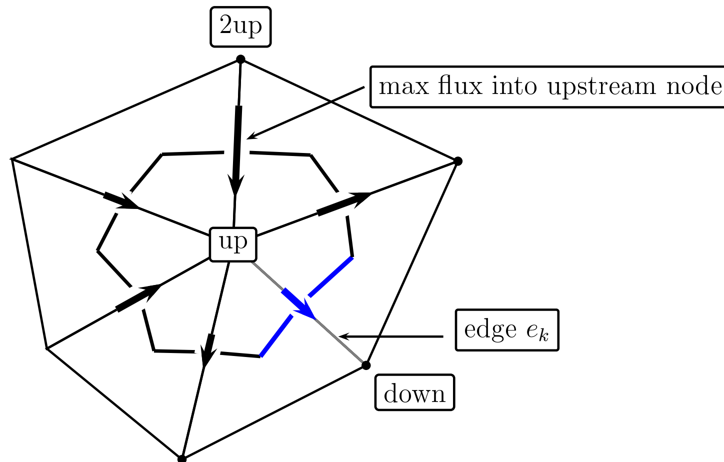


Figure 3.7: Schematic for the choice of upstream, downstream and 2up nodes for the edge e_k, which has faces marked blue. The upstream and downstream nodes are chosen according to the direction of flow along the edge (direction and magnitude indicated by arrows). The 2up node is then chosen using the maximum potential method, with the node corresponding to the maximum flux into the control volume around the node n_{up}.

which has range $\sigma \in [0, 2)$. We note the for the van Leer limiter (3.61), when the sensor $r > 1$ the weighting will be biased towards the downstream node. The parabolic limiter uses the following limiter function

$$\sigma(r) = \begin{cases} r(2-r), & r < 1 \\ 1, & r \geq 1 \end{cases}, \tag{3.62}$$

which has a range $\sigma \in [0, 1]$.

Fluxes at the boundary

The formulation of fluxes above is for interior control volume faces, i.e. faces that are not on the boundary. The flux at boundary faces is prescribed, either directly or indirectly, by the boundary conditions. At boundary faces where the fluid flux is specified by a Neumann boundary condition, the fluid

flux is specified according to (2.21) as a function of space and time

$$[\boldsymbol{q}]_j \cdot \boldsymbol{n}_{ij} = q_b(t, \boldsymbol{x}). \tag{3.63}$$

The flux is not explicitly specified at boundaries subject to Dirichlet boundary conditions on pressure head (Type 1 boundary condition in §2.3). To determine the flux over the boundary, we assume that the prescribed pressure head at the boundary is constant (or changes slowly over time)

$$\psi_i = \psi_b. \tag{3.64}$$

Under the assumption in (3.64), the moisture content is constant, which implies that the net volumetric flux of fluid over the surface Γ_i is zero

$$\int_{\Gamma_i} \boldsymbol{q} \cdot \boldsymbol{n} \, \mathrm{d}\sigma = 0.$$

The net flux over the control volume surface Γ_i can be broken into flux over the Dirichlet and non-Dirichlet parts of the surface, Γ_i^D and $\Gamma_i^{\not D}$ respectively, to get the following expression for flux over the Dirichlet boundary

$$\int_{\Gamma_i^D} \boldsymbol{q} \cdot \boldsymbol{n} \, \mathrm{d}\sigma = -\int_{\Gamma_i^{\not D}} \boldsymbol{q} \cdot \boldsymbol{n} \, \mathrm{d}\sigma. \tag{3.65}$$

The flux over each of the non-Dirichlet faces is then approximated using the midpoint rule according to (3.37), so the net flux over the Dirichlet boundary of Γ_i can be expressed in terms of the fluxes over the non-Dirichlet faces

$$\int_{\Gamma_i^D} \boldsymbol{q} \cdot \boldsymbol{n} \, \mathrm{d}\sigma \approx -\sum_{j \in \mathcal{F}_j^{\not D}} A_j \, [\boldsymbol{q}]_j \cdot \boldsymbol{n}_{ij}, \tag{3.66}$$

where $\mathcal{F}_j^{\not D}$ is the set of non-Dirichlet faces of Γ_i. The net volumetric flux computed using equation (3.66) can be used as a volumetric source term in equation (3.34) when computing mass balances, and for the well-mixed advective solute transport boundary condition defined in equation (2.27).

Assembling the flux

We can now write an expression for the the surface flux integral in the general form of the conservation equation (3.20) using the expressions for the different components of the flux at control volume faces. The scaled surface flux over the control volume surface Γ_i is defined by

$$Q_i^\alpha = \frac{1}{\Delta_i} \int_{\Gamma_i} \boldsymbol{q}^\alpha \cdot \boldsymbol{n} \, \mathrm{d}\sigma,$$

which can be approximated as the sum of discrete fluxes over each face according to equations (3.36)–(3.37)

$$Q_i^\alpha \approx \frac{1}{\Delta_i} \sum_{j \in \mathcal{F}_i} A_j \left[\boldsymbol{q}^\alpha\right]_j \cdot \boldsymbol{n}_{ij}. \tag{3.67}$$

The fluid and solute fluxes, $\boldsymbol{q}^w$ and $\boldsymbol{q}^s$ respectively, can be approximated at the control volume face f_j using the volumetric flux (3.43), advected density and concentration determined using flux limiting (3.56), dispersion (3.44), and the concentration gradient (3.41b):

$$\left[\boldsymbol{q}^w\right]_j = \left[\rho\right]_j \left[\boldsymbol{q}\right]_j, \tag{3.68}$$

$$\left[\boldsymbol{q}^s\right]_j = \left[c\right]_j \left[\boldsymbol{q}\right]_j - \left[\theta \mathsf{D}_h\right]_j \left[\boldsymbol{\nabla} c\right]_j. \tag{3.69}$$

3.2.4 Discretised Equations

Given the approximations for the accumulation terms, source terms and flux integrals derived in §3.2.1, §3.2.2 and §3.2.3 respectively, we can derive finite volume discretisations for the conservative form of the governing equations that are defined in (3.18a)–(3.18b). In this section we apply the CV-FE discretisation to each of the equations in the four formulations presented in §2.4, to generate a semi-discrete system of differential algebraic equations (DAEs) for each formulation.

Richards' equation: the PR formulation

For the PR formulation of Richards' equation, the mass balance equation (2.30) is discretised as per the integral form in (3.20). Then, the approximations for the accumulation term (3.32), the source term (3.34a) and the flux (3.67) are applied. The resultant semi-discrete system has one implicit differential equation for each node

$$f_i(t, \boldsymbol{\psi}, \psi_i') = n_i(\psi_i)\psi_i' + Q_i^w(\boldsymbol{\psi}) - \rho_i S_i = 0, \tag{3.70}$$

with one variable, pressure head ψ_i, to be solved for at each node. In (3.70) $\boldsymbol{\psi}$ is a vector of the nodal pressure head values, and ψ_i' is the derivative of pressure head at the node. The variable ψ_i is a differential variable in (3.70) because its derivative ψ_i' is explicitly part of the formulation. Indeed, the equation can be written explicitly as an ordinary differential equation

$$\psi_i' = -\frac{1}{n_i(\psi_i)} \left(Q_i^w(\boldsymbol{\psi}) - \rho_i S_i \right), \tag{3.71}$$

so long as $n_i(\psi_i) \neq 0$. However, under fully saturated conditions the value of $n_i(\psi_i)$ is very small, and is zero if furthermore the porous medium is inelastic and the fluid is incompressible. It is known that division by $n_i(\psi_i)$ causes numeric difficulties when attempting to solve (3.71) using an explicit ODE solver (Tocci et al., 1997) where $n_i(\psi_i)$ is small.

Instead, the system of equations is written in the form of a general DAE system

$$\boldsymbol{F}(t, \boldsymbol{\psi}, \boldsymbol{\psi}') = \boldsymbol{0}, \tag{3.72}$$

where $\boldsymbol{F} = [f_1, f_2, \ldots, f_{n_n}]^T$ is the vector formed by the nodal equations in (3.70).

Because the system is formulated as a DAE, it is possible to impose Dirichlet boundary conditions by replacing the mass balance equation (3.70) with the

following algebraic equation

$$f(t, \psi_i) = \psi_i - \psi_b(\boldsymbol{x}_i, t) = 0, \tag{3.73}$$

which is the discrete analogue of the definition of the Dirichlet boundary condition in (2.19).

Richards' equation: the MPR formulation

There are two primary variables, fluid mass M_i and pressure head ψ_i, in the MPR formulation of Richards' equation, and two equations per node. The first equation is the mass balance equation (2.31), to which the CV-FE discretisation is applied to give the following differential equation at each node

$$f_i(t, \boldsymbol{\psi}, M_i') = M_i' + Q_i^w(\boldsymbol{\psi}) - \rho_i S_i = 0. \tag{3.74}$$

This ordinary differential equation is analogues to equation (3.70) for the PR formulation, except the temporal derivative is in terms of fluid mass M_i, which is the differential variable. The second equation in the MPR formulation is the algebraic equation (2.32), which is integrated over the control volume to obtain the following equation

$$g_i(t, \psi_i, M_i) = M_i - \rho_i \theta_i = 0. \tag{3.75}$$

Pressure head ψ_i is an algebraic variable in this formulation because its temporal derivative does not appear explicitly in the formulation of equation (3.74) or (3.75).

The set of equations at all nodes in the mesh from (3.74) and (3.75) is formulated as a semi-explicit DAE system

$$\boldsymbol{F}(t, \boldsymbol{y}, \boldsymbol{y}') = \begin{bmatrix} \boldsymbol{f}(t, \boldsymbol{\psi}, \boldsymbol{M}') \\ \boldsymbol{g}(t, \boldsymbol{\psi}, \boldsymbol{M}) \end{bmatrix} = \boldsymbol{0}, \tag{3.76}$$

where $\boldsymbol{f} = [f_1, f_2, \ldots, f_{n_n}]^T$ and $\boldsymbol{g} = [g_1, g_2, \ldots, g_{n_n}]^T$ are vectors of the differential and algebraic equations defined in (3.74) and (3.75). The global variable vector $\boldsymbol{y}$ in (3.76) is ordered with the pressure head values first, followed by the fluid mass

$$\boldsymbol{y} = \begin{bmatrix} \boldsymbol{\psi} \\ \boldsymbol{M} \end{bmatrix}. \tag{3.77}$$

Transport model: the PC formulation

The discretisation of the two mass balance equations for the fluid and the solute for the PC formulation follows the same process as for the fluid mass balance equation (3.70) for the PR formulation. Pressure head and concentration, ψ_i and c_i, are the two primary variables at each node. Applying the CV-FE discretisation to equations (2.35a) and (2.35b) gives two coupled semi-implicit differential equations per node, namely

$$f_i^w(t, \boldsymbol{y}, \boldsymbol{y}') = n_i \psi_i' + \eta \rho_0 \theta_i c_i' + Q_i^w(\boldsymbol{y}) - \rho_i S_i, \tag{3.78a}$$

$$f_i^s(t, \boldsymbol{y}, \boldsymbol{y}') = c_i a_i \psi_i' + \theta_i c_i' + Q_i^s(\boldsymbol{y}) - c_i S_i, \tag{3.78b}$$

where both ψ_i and c_i are differential variables.

The equations in the resultant DAE system are ordered as follows

$$\boldsymbol{F}(t, \boldsymbol{y}, \boldsymbol{y}') = \begin{bmatrix} \boldsymbol{f^w}(t, \boldsymbol{y}, \boldsymbol{y}') \\ \boldsymbol{f^s}(t, \boldsymbol{y}, \boldsymbol{y}') \end{bmatrix} = \boldsymbol{0}, \tag{3.79}$$

where $\boldsymbol{f}^w = [f_1^w, f_2^w, \ldots, f_{n_n}^w]^T$ and $\boldsymbol{f}^s = [f_1^s, f_2^s, \ldots, f_{n_n}^s]^T$ are vectors of the nodal equations for fluid and salt mass balance in (3.78a) and (3.78b) respectively. The global variable vector $\boldsymbol{y}$ in (3.76) is ordered with the pressure head values first, followed by the concentration values

$$\boldsymbol{y} = \begin{bmatrix} \boldsymbol{\psi} \\ \boldsymbol{c} \end{bmatrix}. \tag{3.80}$$

Transport model: the MMPC formulation

The mixed formulation for the coupled flow and transport model has two additional primary variables for the conserved quantities: fluid mass and solute, M_i and C_i respectively. The mass balance equations for fluid (2.36a) and solute (2.36b) in the MMPC formulation are discretised using the CV-FE method to give two coupled differential equations for each node

$$f_i^w(t, \boldsymbol{y}, \boldsymbol{y}') = M_i' + Q_i^w(\boldsymbol{y}) - \rho_i S_i = 0, \tag{3.81a}$$

$$f_i^s(t, \boldsymbol{y}, \boldsymbol{y}') = C_i' + Q_i^s(\boldsymbol{y}) - c_i S_i = 0. \tag{3.81b}$$

The differential equations in (3.81) correspond to (3.78), where the temporal derivatives are of the volume average variables for the conserved quantities, M_i and C_i, which are the differential variables in this formulation. To close the system, the two algebraic equations (2.37a) and (2.37b) in the MMPC formulation are integrated over each control volume:

$$g_i^w(t, \boldsymbol{y}) = M_i - \rho_i \theta_i = 0, \tag{3.82a}$$

$$g_i^s(t, \boldsymbol{y}) = C_i - c_i \theta_i = 0. \tag{3.82b}$$

Pressure head ψ_i and c_i are algebraic variables in this formulation because their derivatives are not used explicitly in the discrete equations in (3.81) and (3.82).

The discrete equations in (3.81) and (3.82) for all nodes in the mesh are formulated as a DAE system, with the equations ordered with the differential equations first, followed by the algebraic equations

$$\boldsymbol{F}(t, \boldsymbol{y}, \boldsymbol{y}') = \begin{bmatrix} \boldsymbol{f}(t, \boldsymbol{y}, \boldsymbol{y}') \\ \boldsymbol{g}(t, \boldsymbol{y}, \boldsymbol{y}') \end{bmatrix} = \boldsymbol{0}, \tag{3.83}$$

where the differential and algebraic equations are ordered as follows:

$$\boldsymbol{f}(t,\boldsymbol{y},\boldsymbol{y}') = \begin{bmatrix} \boldsymbol{f}^{\boldsymbol{w}}(t,\boldsymbol{y},\boldsymbol{y}') \\ \boldsymbol{f}^{\boldsymbol{s}}(t,\boldsymbol{y},\boldsymbol{y}') \end{bmatrix} \quad \text{and} \quad \boldsymbol{g}(t,\boldsymbol{y},\boldsymbol{y}') = \begin{bmatrix} \boldsymbol{g}^{\boldsymbol{w}}(t,\boldsymbol{y}) \\ \boldsymbol{g}^{\boldsymbol{s}}(t,\boldsymbol{y}) \end{bmatrix}.$$

The vector $\boldsymbol{y}$ of nodal values for each of the primary variables is ordered with the algebraic variables, pressure head and concentration, first, followed by the differential variables, fluid mass and solute:

$$\boldsymbol{y} = \begin{bmatrix} \boldsymbol{\psi} \\ \boldsymbol{c} \\ \boldsymbol{M} \\ \boldsymbol{C} \end{bmatrix}. \tag{3.84}$$

3.3 Temporal Solution

The CV-FE discretisation of each formulation gives a system of semi-discrete differential algebraic equations with the general form

$$\boldsymbol{F}(t,\boldsymbol{y},\boldsymbol{y}') = \boldsymbol{0} \tag{3.85a}$$

$$\boldsymbol{y}(t_0) = \boldsymbol{y}_0 \tag{3.85b}$$

$$\boldsymbol{y}'(t_0) = \boldsymbol{y}_0', \tag{3.85c}$$

where time $t \in [t_0, t_{\text{final}}]$, and $\boldsymbol{y}, \boldsymbol{y}' \in \mathbb{R}^N$ are vectors of primary variables and their derivatives at mesh nodes. The nonlinear function $\boldsymbol{F}$ in (3.85a) is referred to as the *nonlinear residual function.*

Software libraries for the solution of general DAE systems that use implicit, multi-step methods have been shown to be well-suited for solving the stiff systems that typically arise from the discretisation of Richards' equation (Tocci et al., 1997, Kees and Miller, 1999, 2002, Farthing et al., 2003, Fahs et al., 2009). In this work the IDA library, which is part of the SUNDIALS suite of solvers (Hindmarsh et al., 2005), is used to solve the initial value

problem in (3.85). A detailed derivation and analysis of the algorithms used by IDA is presented in Chapter 5 of Brenan et al. (1996). Here the basic algorithm is presented, with a focus on the parts of the method that concern our application.

A backwards differentiation formula (BDF) approximates the derivative of the solution vector at the nth time step, $t = t_n$, as a linear combination of the solution at t_n and previous time steps:

$$\boldsymbol{y}'_n = \alpha_n \boldsymbol{y}_n + \boldsymbol{\beta}_n, \tag{3.86}$$

where

$$\alpha_n = \alpha_0/\tau_n \qquad \text{and} \qquad \boldsymbol{\beta}_n = \sum_{j=1}^{k} \alpha_j \boldsymbol{y}_{n-j}, \tag{3.87}$$

and $\tau_n = t_n - t_{n-1}$, and α_j, $\{j = 0, 1, \ldots, k\}$ are the coefficients of the order-k BDF.

Substituting the derivative approximation (3.86) into the DAE system (3.85a) gives the discrete algebraic system

$$\boldsymbol{F}(t, \boldsymbol{y}, \alpha\boldsymbol{y} + \boldsymbol{\beta}) = \boldsymbol{0}, \tag{3.88}$$

where all subscript notation has been dropped for clarity, and all variables are evaluated at $t = t_n$. The system of discrete algebraic equations in (3.88) is solved using an inexact Newton method, with the mth Newton update vector, $\boldsymbol{y}^{(m+1)} = \boldsymbol{y}^{(m)} + \boldsymbol{s}^{(m)}$ determined by solving the linear equation

$$G\boldsymbol{s}^{(m)} = -\boldsymbol{F}(t, \boldsymbol{y}^{(m)}, \alpha\boldsymbol{y}^{(m)} + \boldsymbol{\beta}). \tag{3.89}$$

The iteration matrix G is defined by

$$G = \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}} + \alpha \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{y}'}, \tag{3.90}$$

where we use the notation $\partial \boldsymbol{F}/\partial \boldsymbol{y}$ to indicate the Jacobian of $\boldsymbol{F}$ with respect

to $\boldsymbol{y}$.

Testing for convergence of the inexact Newton method is accomplished using a weighted root mean squared norm, defined by

$$\| \boldsymbol{x} \|_{rms} = \left(\frac{1}{N} \sum_{i=1}^{N} \left(\frac{x_i}{w_i} \right)^2 \right)^{1/2}, \tag{3.91}$$

where the weight w_i incorporates the values in the solution vector $\boldsymbol{y}^{(0)}$ at the beginning of the step and the user-specified absolute and relative error tolerances, τ_a and τ_r:

$$w_i = \tau_r |y_i^{(0)}| + \tau_a. \tag{3.92}$$

The Newton iterations are terminated when the following condition, due to Shampine (1980), is satisfied

$$\frac{\rho}{1-\rho} \left\| \boldsymbol{y}^{(m+1)} - \boldsymbol{y}^{(m)} \right\|_{rms} \leq 1/3, \tag{3.93}$$

where ρ is an estimate of the rate of convergence. The error tolerances are contained in the definition of the norm (3.91), so that the required accuracy is obtained when the left hand side of (3.93) is less than one. The value of 1/3 is a safety factor chosen to ensure that the desired accuracy has been achieved, because the GMRES method used to solve the linear system is not exact (Hindmarsh et al., 2005).

The estimated rate of convergence ρ used in (3.93) is calculated when two or more corrector iterations have been performed

$$\rho = \left(\frac{\left\| \boldsymbol{y}^{(m+1)} - \boldsymbol{y}^{(m)} \right\|_{rms}}{\| \boldsymbol{y}^{(1)} - \boldsymbol{y}^{(0)} \|_{rms}} \right)^{1/m}. \tag{3.94}$$

3.3.1 Solving the Linear System

The linear system (3.89) is solved using a preconditioner GMRES solver (Saad and Schultz, 1986). The iteration matrix G is only felt through its multiplication by a vector, which can be approximated using forward differences (Kelley, 1995):

$$G\boldsymbol{v} \approx \frac{\boldsymbol{F}(t, \boldsymbol{y} + \epsilon\boldsymbol{v}, \alpha(\boldsymbol{y} + \epsilon\boldsymbol{v}) + \boldsymbol{\beta}) - \boldsymbol{F}(t, \boldsymbol{y}, \alpha\boldsymbol{y} + \boldsymbol{\beta})}{\epsilon}. \quad (3.95)$$

The value of the residual function $\boldsymbol{F}(t, \boldsymbol{y}, \alpha\boldsymbol{y} + \boldsymbol{\beta})$ is computed at the start of the each Newton iteration, and hence computing the matrix-vector product in (3.95) requires one additional residual evaluation. The small parameter ϵ in (3.95) is chosen based on the values in $\boldsymbol{y}$ and $\boldsymbol{y}'$ so as to reduce round-off error in the forward difference (Hindmarsh et al., 2005). This is called a matrix-free method, because the iteration matrix G is not explicitly required to solve the linear system (with the possible excpetion of periodically computing G to determine a preconditioner, see §3.3.2).

An advantage of this matrix-free approach is that up-to-date matrix information is always used, by virtue of the approximation for the matrix-vector product in (3.95) performing the approximation with the value of $\boldsymbol{y}$ from the start of the time step. This is advantageous, because using out-of-date matrix information can require more Newton iterations to resolve the nonlinear system, and has been shown to cause premature termination of the Newton iterations when solving Richards' equation (Kelley et al., 1998). If the iteration matrix G is used explicitly, a Chord method is used whereby the matrix is recomputed only periodically, to avoid the considerable computational expense of forming G (Kelley, 1995).

3.3.2 Preconditioner

Preconditioning is required to ensure the timely convergence of the GMRES iterations. The role of the preconditioner is to transform the linear system such that the spectral properties of the transformed linear system are more amenable to solution using GMRES (Saad, 2000). A copy of the iteration matrix is required to form the preconditioners that are considered in this thesis, so that the iteration matrix must be recomputed each time the preconditioner is formed. IDA attempts to minimise the number of times the preconditioner is formed by recomputing the preconditioner only when the value of α in (3.90) changes (that is, when the time step size or integration order changes), or when an out-of-date preconditioner fails to accelerate GMRES convergence acceptably.

The iteration matrix is computed using the forward difference approximation for matrix-vector multiplication in equation (3.95). The jth column of the iteration matrix can be approximated using the following formula

$$G(:,j) = \frac{\boldsymbol{F}(t, \boldsymbol{y} + \epsilon \boldsymbol{e}_j, \alpha(\boldsymbol{y} + \epsilon \boldsymbol{e}_j) + \boldsymbol{\beta}) - \boldsymbol{F}(t, \boldsymbol{y}, \alpha \boldsymbol{y} + \boldsymbol{\beta})}{\epsilon}, \tag{3.96}$$

which is equivalent to multiplying the matrix by $\boldsymbol{e}_j$, the jth canonical vector in $\mathbb{R}^N$. In §4.4.2 an efficient method that exploits the sparsity in G to form G with a small number of residual evaluations will be presented.

Efficient preconditioners for the modified mixed formulations

The modified-mixed formulations of the governing equations, the MPR formulation (3.76) and MMPC formulation (3.83), have twice as many variables as their reduced counterparts, the PR formulation (3.72) and PC formulation (3.79) respectively. An important novel development in this thesis (Cumming et al., 2011) is a preconditioning method that reduces the computational cost of the mixed formulations to that of the reduced forms. Similar observations were made by Kees and Miller (2002) and Farthing et al. (2003) for the direct

solution of the linear systems, however the approach proposed here allows the solution of the mixed formulations in general purpose time stepping packages.

The approach is based on analysis of the structure of the iteration matrix for the mixed forms. The variables for the MPR formulation in (3.77) and MMPC formulation in (3.84) are ordered such that the algebraic variables are first

$$\boldsymbol{y} = \begin{bmatrix} \boldsymbol{y}_a \\ \boldsymbol{y}_d \end{bmatrix}, \tag{3.97}$$

where $\boldsymbol{y}_a$ and $\boldsymbol{y}_d$ are vectors of the algebraic and differential variables respectively[8]. The DAE system is semi-explicit (Brenan et al., 1996), with differential equations first, followed by the algebraic equations

$$\boldsymbol{F}(t, \boldsymbol{y}, \boldsymbol{y}_d') = \begin{bmatrix} \boldsymbol{f}(t, \boldsymbol{y}_a, \boldsymbol{y}_d') \\ \boldsymbol{g}(\boldsymbol{y}_a, \boldsymbol{y}_d) \end{bmatrix}. \tag{3.98}$$

To determine the structure of the global iteration matrix G, we substitute the variable and residual definitions, (3.97) and (3.98) respectively, into the general equation for the iteration matrix (3.90)

$$G = \begin{bmatrix} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}_a} & \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}_d} \\ \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{y}_a} & \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{y}_d} \end{bmatrix} + \alpha \begin{bmatrix} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}_a'} & \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}_d'} \\ \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{y}_a'} & \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{y}_d'} \end{bmatrix}$$

which is simplified due to the semi-eplicit formulation to

$$\begin{aligned} G &= \begin{bmatrix} \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}_a} & 0 \\ \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{y}_a} & I \end{bmatrix} + \begin{bmatrix} 0 & \alpha I \\ 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} A & \alpha I \\ D & I \end{bmatrix}. \end{aligned} \tag{3.99}$$

We see from (3.99), that only the columns of the matrix corresponding to differentiation with respect to algebraic variables (the D and A blocks) need to

[8] For the MPR formulation $\boldsymbol{y}_a = \boldsymbol{\psi}$ and $\boldsymbol{y}_d = \boldsymbol{M}$, and for the MMPC formulation $\boldsymbol{y}_a = [\boldsymbol{\psi}; \boldsymbol{c}]$ and $\boldsymbol{y}_d = [\boldsymbol{M}; \boldsymbol{C}]$.

be found, which reduces the number of residual evaluations to the equivalent of those required for the PR and PC formulations.

The sub-matrix $A = \partial \boldsymbol{F}_d / \partial \boldsymbol{y}_a$ in (3.99) is the Jacobian matrix associated with the spatial discretisation of the mass balance equations. For the MPR formulation the matrix $D = \partial \boldsymbol{F}_a / \partial \boldsymbol{y}_a$ is a diagonal matrix, with diagonal entries defined

$$d_{ii} = \frac{\partial g_i}{\partial \psi_i}, \tag{3.100}$$

and for the MMPC formulation the D matrix can be reordered as a block diagonal matrix with diagonal nonzero 2×2 blocks

$$D_{ii} = \begin{bmatrix} \frac{\partial g_i^w}{\partial \psi_i} & \frac{\partial g_i^s}{\partial \psi_i} \\ \frac{\partial g_i^w}{\partial c_i} & \frac{\partial g_i^s}{\partial c_i} \end{bmatrix}. \tag{3.101}$$

The cost of computing and applying the preconditioner for the mixed system (3.99) can be reduced to be similar to that of preconditioning for just the A block. We consider the linear system

$$\begin{bmatrix} A & \alpha I \\ D & I \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_a \\ \boldsymbol{x}_d \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_a \\ \boldsymbol{b}_d \end{bmatrix}. \tag{3.102}$$

Under the assumption that the Schur Complement $A - \alpha D$ is nonsingular, the solution of (3.102) can be solved in two steps as follows

$$\text{solve for } \boldsymbol{x}_a: \qquad \boldsymbol{x}_a = (A - \alpha D)^{-1} (\boldsymbol{b}_a - \boldsymbol{b}_d), \tag{3.103a}$$

$$\text{solve for } \boldsymbol{x}_d: \qquad \boldsymbol{x}_d = \boldsymbol{b}_d - D \boldsymbol{x}_a. \tag{3.103b}$$

Forming the Schur complement involves only an update to the diagonal of the matrix A. Thus, to calculate the preconditioner for the full system, we only need to compute a preconditioner for the Schur complement. When applying the preconditioner, the second step (3.103) is trivial to perform because D is a diagonal matrix, and the first step (3.103a) is the dominant cost, equivalent to the cost of applying the preconditioner for the reduced

PR and PC formulations.

3.4 Conclusions

In this chapter the control volume-finite element (CV-FE) spatial discretisation was applied to the governing equations for variably-saturated flow and contaminant transport in porous media. The CV-FE discretisation was chosen due to its suitability for heterogeneous media and unstructured meshes. The method computes consistent fluxes by virtue of material properties and gradients being continuous at control volume faces, where the fluxes are evaluated. This ensures local conservation of mass, and avoids any approximation of material properties in the flux formulation.

Upstream weighting is typically used for mobility and advection terms in groundwater modelling, but can suffer from excessive numerical diffusion. In this chapter we discussed flux limiting methods that have been successfully applied problems in wood drying[9] in the context of our CV-FE discretisation for groundwater flow and contaminant transport. In Chapter 6 the accuracy of flux limiters will be investigated for test cases that suffer from numerical diffusion when upstream weighting is used.

The mixed modified formulation due to Kees and Miller (2002) was adapted to the CV-FE discretisation for Richards' equation, and extended to the coupled flow and transport model. The resultant "modified mixed" formulations were expressed naturally in the finite volume framework, by using volume averages of the conserved quantities (fluid mass M and solute C) as primary variables in the semi-discrete system of differential-algebraic equations (DAEs) that arises from applying the spatial discretisation in conjunction with the method of lines (MOL).

The system of DAEs is amenable to solution using DAE solvers that use

[9]Wood drying is described by similar physics to groundwater flow

advanced adaptive time stepping methods. We gave an overview of the methods used by one such library, IDA, to solve the nonlinear system of equations at each time step using a matrix-free Newton Krylov method. The modified mixed formulations have twice as many variables and equations as their reduced counterparts, however we presented a simple Schur-complement method for reducing the size of the system that is preconditioned by half. This allows the efficient solution of the modified mixed formulations in general solver packages, with little computational overhead relative to the formulations with fewer variables.

The two chapters that follow, Chapter 4 and Chapter 5, will investigate the implementation of the spatial and temporal discretisations presented in this chapter. It will be shown that the CV-FE discretisation, and the matrix-free Newton-Krylov solver, can be implemented efficiently on clusters that use multi-core CPU and many-core GPU processors.

Chapter 4

Implementation: Algorithms and Data Structures

This chapter presents a high-level discussion of the algorithms, data structures and of the CV-FE discretisation and implicit time stepping for cluster computing implemented in FVMPor, which is the software package developed in this thesis (see §1.2). The discussion here will be kept general, without details for any specific hardware implementation, and the following chapter will outline the implementation of the algorithms and approaches discussed here for multi-core and GPU hardware platforms in C++.

The chapter starts with an overview of steps taken in an internal time step of IDA in §4.1, followed by a discussion of the CV-FE spatial discretisation in §4.2. The method of domain decomposition used to distribute the model over different computational nodes is presented in §4.3. Finally, in §4.4, methods are discussed for obtaining fine-grained and coarse-grained parallelism of preconditioners based on sparse factorisations.

4.1 Time Stepping with IDA

IDA[1] is an open source code for solving initial value problems in differential-algebraic equations (DAEs) (Hindmarsh et al., 2005). It uses an implicit higher-order time stepping method based on backwards differentiation formulae (BDFs). IDA was chosen for solving the DAE systems that arise from discretising the governing equations in this thesis for the following reasons:

- Implicit, higher-order solvers that use BDFs have been shown to be well-suited for solving the stiff systems that typically arise from the discretisation of Richards' equation (Tocci et al., 1997, Kees and Miller, 1999, 2002, Farthing et al., 2003, Fahs et al., 2009) and for density-dependent flow and transport problems (Liu et al., 2002, Younes et al., 2009).
- The robustness and accuracy of the solver has been tested on a wide range of applications, and its sophisticated methods for selecting time step size, integration order and error control based on user-specified tolerances have been verified for variably saturated flow in porous media (Tocci et al., 1997).
- The library has a very modular design. The main numerical operations performed by IDA are on data vectors, which are implemented in a separate library. The vector library performs memory allocation and parallel communication where necessary, so that it is relatively straightforward to add support for GPUs in IDA by replacing this library with a GPU library that supports the same interface.
- Support for cluster implementations is provided using the message passing library MPI. The parallel implementation uses an inexact Newton-Krylov method, which is highly scalable on large clusters, and is amenable to implementation with GPUs.

[1]IDA is part of the SUNDIALS suite of codes for solving ODE and DAE systems: `computation.llnl.gov/casc/sundials/main.html`.

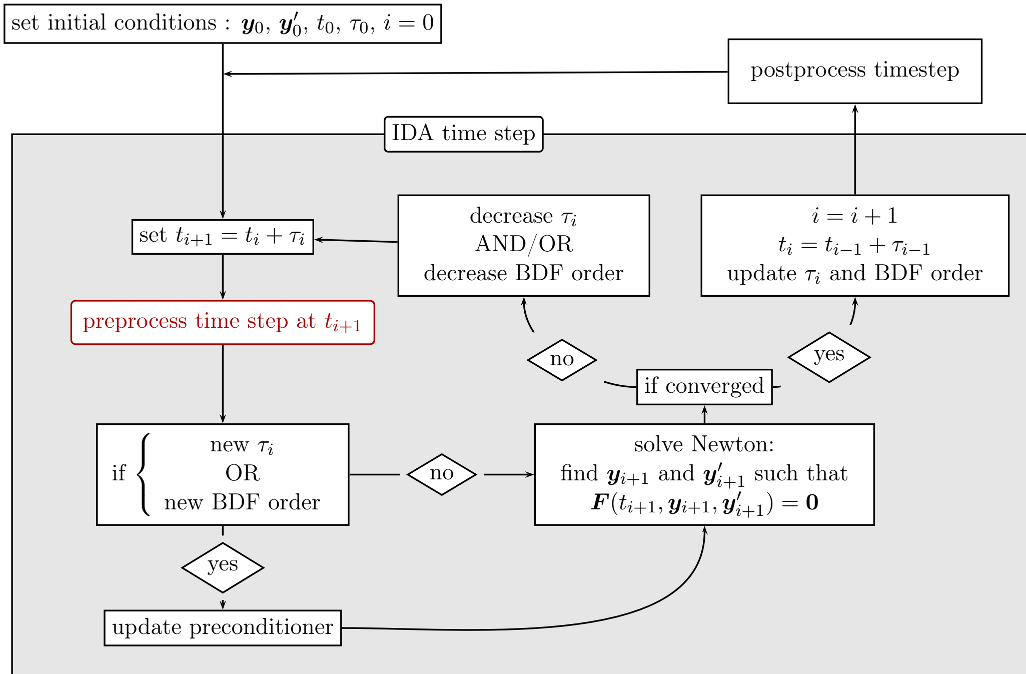


Figure 4.1: Flow chart of the steps taken for one internal time step of IDA (see §3.3 for definition of mathematical symbols). The step *preprocess time step* is highlighted because it is a modification used in this thesis.

IDA interacts with user code through calls to a user-supplied nonlinear residual function, and user-supplied routines for computing, and applying, a preconditioner. The solution is advanced in time by calling IDA to take internal time steps until the final solution time has been reached. The flowchart in Figure 4.1 gives an overview of the internal steps taken by IDA when performing a time step. IDA can be treated as a "black box" solver, however an understanding of the steps taken within IDA is necessary to utilise it effectively.

IDA attempts to take a time step of size τ_{i+1}, however if the Newton iterations fail to converge[2], IDA will reduce the time step and the order of the BDF,

[2]The Newton iterations are judged to have failed if the nonlinear residual is not reduced sufficiently within a maximum number of iterations; or if the convergence is not superlinear; or if the GMRES method fails to converge at any point. See Hindmarsh et al. (2005) for a detailed discussion.

then retry. This process continues until the Newton iterations converge, or until the time step size falls below a minimum threshold value. If the Newton iterations are successful, IDA will increase the time step or integration order if possible in preparation for the next time step, before returning control to the calling code.

Each time step performed by FVMPor has user-defined preprocessing and postprocessing steps. The postprocessing step performs user-defined operations on the solution, such as computing mass balances, and is undertaken outside the time step loop when IDA returns the solution to the calling code. The preprocessing step is performed inside the IDA time step loop, because it sets state information[3] that may depend on the target time t_{i+1}.

4.2 Evaluating the Residual: The CV-FE Discretisation

In this section, we consider the algorithms and data structures used in each step of the nonlinear residual evaluation. The discussion is kept at a high-level to facilitate clear discussion of the methods. However, most of the data structure and algorithm choices are made with the GPU implementation in mind, so references are made to the hardware implementation described in Chapter 5 where it motivates the algorithm and data structure choices.

The same steps are taken in evaluating the residual for all of the PR, MPR, PC and MMPC formulations, however the formulations for Richards' equation (the PR and MPR formulations) are simpler to describe because they have fewer equations. Hence, for clarity of exposition, the residual function for the PR and MPR formulations of Richards' equation are discussed here, with reference to the PC and MMPC formulations where clarification is required.

[3] State information is held constant through throughout a time step, see §4.2.1.

Algorithm 4.1: The steps in evaluating the residual function.

Input : time t, the primary variable vector $\boldsymbol{y}$ and its derivative $\boldsymbol{y}'$
Output: the residual vector $\boldsymbol{r}$
if $t \neq t_{old}$ **then**
  preprocess time step **(1)**;
  $t_{\text{old}} = t$;
interpolate to find gradient $\boldsymbol{\nabla}\psi$ at faces **(2)**;
set edge weights for flux limiting **(3)**;
find fluid properties ρ, θ, k_{rw}, ...etc. **(4)**;
form fluxes $\boldsymbol{q}^{\alpha}$ **(5)**;
gather fluxes and form residual r **(6)**;

Consider the residual function at each node for the PR formulation of Richards' equation (3.70):

$$f_i(t, \boldsymbol{y}, \boldsymbol{y}') = n_i(\psi_i)\psi_i' + \frac{1}{\Delta_i}\sum_{j \in \mathcal{F}_i} A_j \left[\boldsymbol{q}^w\right]_j \cdot \boldsymbol{n}_{ij} - \rho_i S_i. \tag{4.1}$$

Evaluation of the residual function for the PR formulation, $\boldsymbol{F}$ in equation (3.72), sees equation (4.1) evaluated at each node. This entails evaluating the accumulation term, fluxes and source term in the spatial discretisation given the nodal values of pressure head and its derivative specified in $\boldsymbol{y}$ and $\boldsymbol{y}'$. This in turn requires the computation of intermediate values, such as interpolated gradients at control volume faces to evaluate fluxes.

The residual function evaluation is broken into a sequence of steps or stages, shown in Algorithm 4.1, and summarised below:

1. **Time Step Preprocessing**: Set state variables that are fixed during the time step. In Figure 4.1, the preprocess time step task is highlighted because it is not explicitly performed by IDA, but is instead performed the first time that the residual function is called in the time step.

2. **Interpolation**: Interpolate values at nodes to values and gradients at control volume faces using shape function interpolation.

3. **Spatial Weighting**: Compute spatial weightings for advected quantities due to flux limiting.

4. **Secondary Variables**: Compute secondary variables such as density $\rho(\psi, c)$, moisture content $\theta(\psi)$ and relative permeability $k_{\mathrm{rw}}(\psi)$.

5. **Compute Fluxes**: Compute the flux over each control volume face using the values computed in steps 3, 4 and 5.

6. **Assemble Residual**: Compute the residual function for each node by gathering the fluxes for each control volume face, and adding with the accumulation and source terms, as per the PR formulation in equation (4.1).

The remainder of this section will discuss step 1 to step 6 of the residual evaluation in more detail.

Data structures

Before discussing the implementation, we briefly discuss the data structures used within the spatial discretisation. The approach taken here was motivated by the requirement that the software should run on both multi-core CPUs and GPUs without modification. To this end, the residual evaluation is expressed in terms of basic vector-vector operations, and sparse matrix-vector operations, for which efficient multi-core CPU and GPU implementations are provided in the vectorlib library described in §5.2.

To facilitate the implementation using vector and sparse matrix operations, a *flat data model* is used, in which values associated with components of the dual mesh are stored in vectors. For example, the values of density for each control volume in the mesh are stored in vectors of length n_n, and the values of density at each control volume face are stored in a vector of length n_f[4].

[4]We recall from §3.1 that n_n and n_f are the number of nodes and control volume faces in the dual mesh respectively.

Because we use unstructured meshes, the mappings between values in the node, edge and face vectors in the residual evaluation are specified explicitly using indirect indexing. In the flat model, index vectors are used to specify the mapping between values in the node, edge and face vectors. Each of the operations in the residual evaluation can then be expressed using simple indirect indexing with vectors, the implementation of which on both GPU and multi-core CPUs is discussed in §5.2. Furthermore, some global gather operations, such as interpolation, can be expressed as sparse matrix-vector multiplication (SPMV). Where possible, SPMV operations are used due to the availability of highly-optimised libraries for performing them on different hardware.

4.2.1 Time Step Preprocessing

The preprocessing step computes *state information* that is fixed throughout the time step. Examples of state information include time-dependent boundary conditions, and the flow direction information (upstream, downstream and 2up points) used in upstream weighting and flux limiting. The IDA library does not provide a mechanism for performing a preprocessing step, so the information is computed the first time that the residual function is called during a time step. In Algorithm 4.1, this is performed in step 1 by keeping track of the time t with which the residual is called, and recomputing the state information when the value changes between residual calls.

Transient boundary conditions

We consider two types of transient boundary condition. The first is a prescribed fluid flux boundary condition (type 2 in §2.3), for example infiltration due to rainfall at the surface. In the preprocessing stage the value of the flux is set for each face on the boundary using an analytic formula, or from a user-provided input file.

The second type of transient boundary condition we consider is the hydrostatic beach boundary condition (Type 3 in §2.3). At each boundary face, the penalty term R_b in equation (2.23) is set according to the height of the tide at the target time step t_{i+1}.

Seepage boundary conditions are not implemented in this thesis due to limitations imposed by the IDA library, which are discussed in Appendix D.

Setting the flow direction indicators: 1up and 2up points

Upwind weighting and flux limiting use information about the upstream points (1up and 2up) to compute the mobility term and advected quantities at control volume faces. It is necessary to hold the choice of 1up and 2up points fixed for the duration of a time step. For otherwise, if the choice of upstream points were allowed to change during the Newton iterations, the assumption that the residual function is smooth would be violated. In practice, it was found that failing to do this led to convergence difficulties for the Newton iterations.

In §3.2.3 the flow direction indicator (FDI) was introduced. The FDI is a scalar quantity computed for each edge in the domain, the sign of which indicates the edge's upstream and downstream points:

$$\begin{array}{lcl} \mathrm{FDI}_k > 0 & \rightarrow & \text{flow from back to front} \\ \mathrm{FDI}_k < 0 & \rightarrow & \text{flow from front to back} \end{array}, \tag{4.2}$$

where FDI_k is the FDI for the edge e_k. This is illustrated graphically for a two-dimensional mesh in Figure 4.2, however it is important to note that for three-dimensional meshes an edge is also a line segment, and the same approach is used.

The *scalar* $\mathrm{FDI}(\varphi)$ defined in equation (3.52) determines the direction of flow on an edge according to the difference of fluid flow potential φ at the front

and back nodes of the edge[5]

$$\mathrm{FDI}(\varphi)_k = \varphi^{\mathrm{back}} - \varphi^{\mathrm{front}}. \tag{4.3}$$

Algorithm 4.2 determines $\mathrm{FDI}(\varphi)$ for every edge in the domain with a single *for*-loop, using the indices in edge_back and edge_front to indicate the front and back nodes of each edge. The algorithm is relatively trivial, however it is shown here to illustrate the use of precomputed index vectors in the flat data model.

Algorithm 4.2: Algorithm for determining the scalar flow direction indicator $\mathrm{FDI}(\varphi)$.

Input : array potential of nodal φ values
index arrays edge_front and edge_back
Output: array FDI with FDI for each edge in mesh

```
for e ← 0 to n_e − 1 do
    FDI [e] = potential [edge_back [e]]-potential [edge_front [e]];
```

The *vector* FDIs, $\mathrm{FDI}(\boldsymbol{q})$ and $\mathrm{FDI}(\boldsymbol{\nabla}\phi)$ in (3.48) and (3.51), find the net contribution from each of the faces attached to an edge. For example, the net flux on edge e_k according to $\mathrm{FDI}(\boldsymbol{q})$ in equation (3.48) is

$$\mathrm{FDI}(\boldsymbol{q})_k = \sum_{j\in\mathcal{F}_{e_k}} A_j \left[q\right]_j, \tag{4.4}$$

where $[q]_j = [\boldsymbol{q}]_j \cdot \boldsymbol{n}_j$. The global operation of determining the vector of FDI values for all edges using (4.4) can be expressed in terms of a sparse matrix-vector multiplication:

$$\mathrm{FDI} = A_{\mathrm{gather}} \times \boldsymbol{q}_{\mathrm{faces}}, \tag{4.5}$$

where $\boldsymbol{q}_{\mathrm{faces}} \in \mathbb{R}^{n_f}$ is a global face array of the $[q]_j$ values, and the edge

[5]The fluid flow potential for Richards' equation, under the assumption that $\mathrm{d}\rho/\mathrm{d}\psi \approx 0$, is the *hydraulic head* H: $\varphi = H = \psi + z$.

gathering matrix $A_{\text{gather}} = (a_{ij}) \in \mathbb{R}^{n_e \times n_f}$ is a sparse matrix with entries defined:

$$a_{kj} = \begin{cases} A_j, & j \in \mathcal{F}_{e_k} \\ 0, & \text{otherwise} \end{cases} . \tag{4.6}$$

This approach allows us to exploit efficient implementations of sparse matrix-vector multiplication, which is discussed in §5.3.

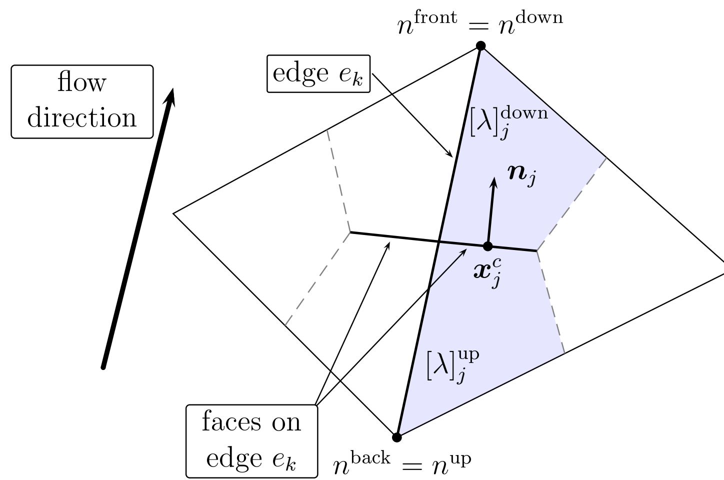


Figure 4.2: Diagram showing the two faces attached to an edge in a two-dimensional mesh. The value of the mobility term k_{rw}, and advected variables ρ and c are determined at the face as a weighted sum of the values at the front and back nodes of the edge.

Upstream weighting chooses the value of the variable at the upstream node according to equation (3.47). Because the upstream node is fixed throughout the time step, the spatial weights w_k^{front} and w_k^{back} are computed in the preprocessing phase:

$$\begin{aligned} \text{FDI}_k \geq 0 \quad &\rightarrow \quad \begin{cases} w_k^{\text{back}} = 0 \\ w_k^{\text{front}} = 1 \end{cases} \\ \text{FDI}_k < 0 \quad &\rightarrow \quad \begin{cases} w_k^{\text{back}} = 1 \\ w_k^{\text{front}} = 0 \end{cases} \end{aligned} . \tag{4.7}$$

Flux limiting requires additional information about the 2up point, which is determined using the method of maximum potential illustrated in Figure 4.3. The method of maximum potential selects as n^{2up} the neighbour of n^{up} that has the largest flow into n^{up}. It is not necessary to compute the 2up point for every edge in the mesh, because edges with the same 1up point also have the same 2up point. Instead, it is sufficient to find the 1up point for each edge, and the neighbour of maximum potential for each node. The upstream, downstream and 2up nodes for every edge in the mesh can then be determined in a *for*-loop, as implemented in Algorithm 4.3.

The algorithm first sets the maximum recorded flux into each point to zero, then loops over each edge in the mesh. The upstream point is determined for the edge, then the neighbour of maximum potential for the edge's downstream node is set to be the edge's upstream node if the flux along the edge exceeds the previously set maximum flow into the downstream point. In this manner, once every edge in the mesh has been visited, the neighbour of maximum potential will have been determined for each node in the mesh.

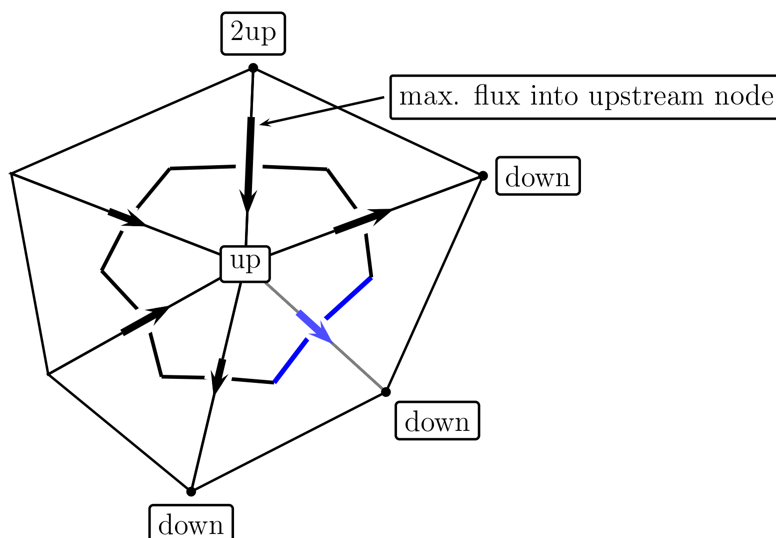


Figure 4.3: Each of the downstream nodes of the upstream node has the same 1up point. Furthermore, each of the downstream points have the same 2up point, which is the node with maximum flux into the control volume of the 1up point.

Algorithm 4.3: Method for determining the 1up nodes and corresponding flux for each control volume in the domain.

```
Input  : array FDI_edges of FDI values for each edge
         index arrays edge_down_node and edge_up_node
Output : array CV_1up_flux with maximum flux into each CV
         array CV_1up of 1up points for each CV
         index array edge_1up of 1up points for each edge

initialise CV_max_1up_flux(:) to 0;
for e ← 0 to n_e − 1 do
    if FDI_edges[e] < 0 then
        down_node ← edge_up_node[e];
        up_node ← edge_down_node[e];
    else
        down_node ← edge_up_node[e];
        up_node ← edge_down_node[e];
    flux ← abs(FDI_edges [e]);
    if flux > CV_1up_flux[down_node] then
        CV_1up_flux [down_node ] ← flux;
        CV_1up [down_node ] ← up_node;
    edge_1up [e] ← up_node;
```

4.2.2 Interpolation

Shape function interpolation reconstructs variables at a control volume face as a weighted sum of the variable at the vertices of the element in which the face lies. For example, consider the shape functions for a tetrahedral element described in §3.1.3. The value of a variable φ is reconstructed at the centroid of face f_j using a dot product

$$s(\boldsymbol{x}_j^c) = \boldsymbol{N}_j \cdot \boldsymbol{\varphi}_{\epsilon_\ell}, \tag{4.8}$$

where $\boldsymbol{N}_j \equiv \boldsymbol{N}(\boldsymbol{x}_j^c)$ is a vector of the shape function weights defined using (3.17a), and $\boldsymbol{\varphi}_{\epsilon_\ell}$ is a vector of the value of φ at the nodes of the element ϵ_ℓ. To express the per-element local operation in (4.8) as a global operation for all faces in the mesh, the weight vector $\boldsymbol{N}_j$ is first expanded as a sparse

vector of length n_n, and the inner product (4.8) is expressed:

$$s(\boldsymbol{x}_j^c) = \boldsymbol{N}_j^s \cdot \boldsymbol{\varphi}, \tag{4.9}$$

where $\boldsymbol{N}_j^s$ is the sparse vector, and $\boldsymbol{\varphi} \in \mathbb{R}^{n_n}$ is a vector of all nodal values for φ. The global operation of interpolating φ to all control volume face midpoints can now be expressed as a sparse matrix-vector multiplication (SPMV)

$$\boldsymbol{\varphi}_f = S\boldsymbol{\varphi}, \tag{4.10}$$

where row j of the sparse matrix $S \in \mathbb{R}^{n_f \times n_n}$ is the sparse weight vector $\boldsymbol{N}_j^s$, and $\boldsymbol{\varphi}_f$ is the vector of values at control volume faces. In the same way, the global operation of computing each component of the gradient at control volume face midpoints for a tetrahedron using the weights in equations (3.17b)–(3.17d) can be expressed using sparse matrix vector multiplication.

The shape function weights are dependent only on the mesh geometry. Hence, they can be stored in matrix form at startup, and applied using efficient SPMV routines discussed in §5.3.

4.2.3 Edge-Based Weighting

The edge weights for upstream weighting are set along with the flow direction information during the time step preprocessing stage in §4.2.1, and the weights for central averaging are constant throughout the simulation according to (3.54).

The flux limiter weights are functions of the solution at the downstream, upstream and 2upstream nodes. The location of the points is fixed for the time step, however the value of the solution changes with each residual evaluation, so the edge weights are set for each residual evaluation when flux limiting is used. The sensor r and limiter σ are determined, then used to set the edge weights according to (3.56). The exception to this rule is when the

flux-based flow direction indicator FDI($\boldsymbol{q}$) is used, in which case the sensor r in equation (3.57) and the edge weights are set in the preprocessing stage because they are impractical to compute at each residual evaluation[6].

Once the edge weights have been set, they can be used to find the advection terms $[\rho]_j$ and $[c]_j$. Algorithm 4.4 is used to compute the advected quantities at each face by looping over each edge, computing the weighting of the value at the front and back nodes, then setting the value at each face attached to the edge to the weighted value. The method used to compute the relative permeability at a face, $[k_{\mathrm{rw}}]_j$, is more complicated due to the dependence of the relative permeability on material properties, and is discussed in §4.2.4.

Algorithm 4.4: Algorithm for determining the density at each face using precomputed edge weights.

Input : array adv_node of nodal values
index arrays edge_front and edge_back
edge weights weight_back and weight_front
Output: array adv_face with face values

```
for k = 0 to n_e − 1 do
    adv_edge = weight_back [k]×adv_node [edge_back [k]]
             + weight_front [k]×adv_node [edge_front [k]];
    for j ∈ F_{e_k} do
        adv_face [j] = adv_edge;
```

4.2.4 Fluid Properties

The fluid properties that close the model are functions of the primary variables and of parameters associated with material properties. Some are used in volume averages (for example moisture content θ), others are used at con-

[6]The flux-based sensor in equation (3.57) is not practical to compute because it requires an approximation to the flux $\boldsymbol{q}$ at each face, which is computationally expensive to form. Truscott (2004) used the value of the flux computed during the previous Newton iteration. Although the value of the flux computed during the previous residual evaluation is available in IDA, it is not possible to determine the context in which the last call was made.

trol volume faces to determine fluxes (relative permeability k_{rw}), and some are used for both (density ρ). The way that the properties are computed depends on whether the property is a function of material properties, and whether it is used as a volume average or a face value. Here we consider the fluid properties for the MPR formulation of Richards' equation, which are summarised in Table 4.1.

variable	material dependent	at faces	volume average	defined
ρ	·	✓	✓	(2.11)
θ	✓	·	✓	(2.13)
n	✓	·	✓	(2.29)
k_{rw}	✓	✓	·	(2.10)
ϕ	✓	·	·	(2.12)
S_w	✓	·	·	(2.8)

Table 4.1: The fluid properties used in the PR and MPR formulations of Richards' equation. The density of the fluid is used as both a volume average and as a face value. Both saturation S_w and porosity ϕ are used as intermediate values for computing moisture content θ and the storage term n.

Take, for example, the density ρ, which is used as both a volume average and a face value. The volume average is computed at each node as a function of the nodal value for pressure head ψ (and concentration c for the coupled transport model), using the linear relationship in equation (2.11). The dimensionless density, $[\hat{\rho}]_j$, in the buoyancy term of the Darcy flux (3.43) is interpolated from the nodal values using shape functions. Conversely, the density in the advection term, $[\rho]_j$ in equation (3.68), is determined using edge-based weights.

The volume average of moisture content θ_i and the face value of relative permeability $[k_{\mathrm{rw}}]_j$, are more complicated to determine than density because they are functions of both material properties and pressure head. Given the definition of the volume average in (3.25), the volume average of moisture content in Ω_i is the weighted sum of the moisture content computed according to the pressure head ψ_i and the material properties $\mathcal{P}_j$ of each sub-control

volume:

$$\theta_i = \frac{1}{\Delta_i} \sum_{s_j \in \mathcal{S}_i} \delta_j \theta(\psi_i; \mathcal{P}_j). \tag{4.11}$$

The value of relative permeability at the face f_j according to Figure 4.2 is a weighted sum of relative permeability computed in the sub-control volumes either side of the face

$$[k_{\text{rw}}]_j = w_j^{\text{back}} [k_{\text{rw}}]_j^{\text{front}} + w_j^{\text{front}} [k_{\text{rw}}]_j^{\text{back}}. \tag{4.12}$$

A naïve approach to finding the values in (4.11) and (4.12) would be to loop over each sub-control volume in the dual mesh, compute the fluid properties according the pressure and material properties of the sub-control volume, and add their weighted contribution to the appropriate volume average (4.11) and face values (4.12). This would be computationally expensive, because the number of sub-control volumes far exceeds the number of nodes in the mesh, and computing individual moisture content and permeability values in each sub-control volume uses expensive power functions in the van Genuchten-Mualem model (see equations (2.9) and (2.10)).

The number of moisture content and permeability values that need to be computed can be reduced using the observation that there are typically relatively few distinct materials, such as clay, sand and rock types in a model. If this is the case, the formula for the volume average of moisture content (4.11) can be rewritten

$$\theta_i = \sum_{\mathcal{P}_k \in P_i} w_k \theta(\psi_i; \mathcal{P}_k), \tag{4.13}$$

where P_i is the set of unique material properties in Ω_i, and w_k are the weights for the relative volume of each material in Ω_i. This is illustrated graphically in Figure 4.4, which shows a control volume with sub-control volumes composed of two materials. The moisture content $\theta(\psi_i; \mathcal{P}_k)$ is identical in each sub-control volume with the same material properties, so only two moisture

content values are required to determine the volume average using (4.13), instead of five values using the naïve approach in (4.11). Similarly, the relative permeability $k_{\mathrm{rw}}(\psi_i; \mathcal{P}_k)$ is computed once for each material property $\mathcal{P}_k$, and its contribution added to each face with the material property $\mathcal{P}_k$ using (4.12).

Determining the volume averages and face values that are dependent on material properties is a performance-critical part of the code, because of the amount of indirect indexing required to map nodal values to material properties and faces. For each material property, the weights in (4.13) are precomputed along with indices of the nodes have contributions from the material property. The volume averages and face values are then computed efficiently by looping over the material properties, using the approach discussed in §5.6.4.

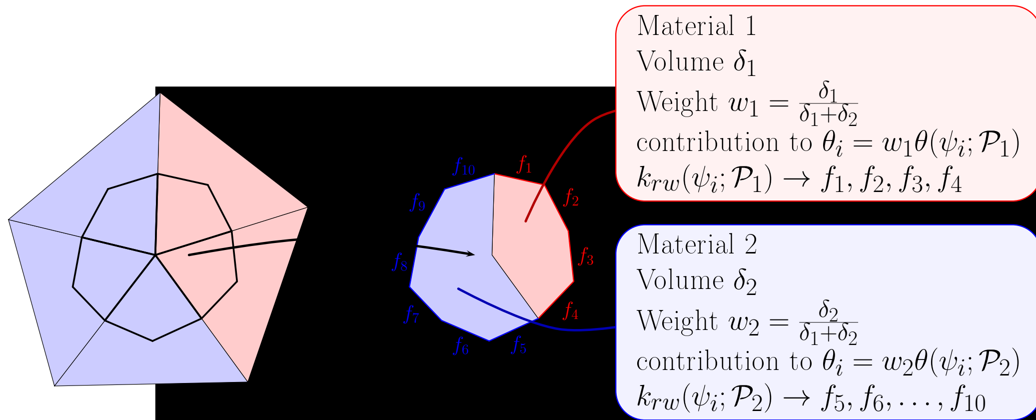


Figure 4.4: A control volume composed of sub-control volumes with two different material properties, denoted by colour. Moisture content and relative permeability are calculated for each material property, and mapped onto the sub-control volumes and control volume faces that share the material property.

4.2.5 Flux Assembly

The net flux of fluid mass over a control volume face f_j is a scalar value, that the CV-FE discretisation approximates using the midpoint rule (3.37)

$$[q^{\mathrm{w}}]_j = A_j \, [\boldsymbol{q}^{\mathrm{w}}]_j \cdot \boldsymbol{n}_j. \tag{4.14}$$

For interior control volume faces (faces that are not on the domain boundary) the volumetric flux vector at the centre of the control volume face is the approximation to the Darcy flux defined in (3.43)

$$[\boldsymbol{q}^{\mathrm{w}}]_j = -\underbrace{[\rho]_j \, [k_{\mathrm{rw}}]_j}_{\text{weighting}} \mathsf{K}_j \left(\underbrace{[\boldsymbol{\nabla}\psi]_j}_{\text{shape}} + \frac{1}{\rho_0} \underbrace{[\rho]_j}_{\text{shape}} \boldsymbol{\nabla} z \right), \tag{4.15}$$

where the subscripts *shape* and *weighting* indicate the method used to reconstruct the face values from nodal information. The gradient and density values from the shape function interpolation, and the mobility and advection terms computed using edge-based weighting are stored in vectors. Once the vectors of face values have been computed, the global vector of $[q^{\mathrm{w}}]_j$ values is formed using a series of basic vector operations discussed in §5.6.5.

At boundary faces with Neumann boundary conditions specified, the volumetric flux over the face is defined as a function of space and time:

$$[q]_j = q_b(t, \boldsymbol{x}). \tag{4.16}$$

The fluxes specified by (4.16) at each face do not change throughout a time step, and are computed and stored in place in the flux vector during the preprocessing phase in §4.2.1.

The flux is not computed over boundaries subject to Dirichlet boundary conditions for Richards' equation because the residual function based on the mass balance equation at nodes subject to Dirichlet boundaries is replaced with the algebraic equation (3.73). However, the flux is needed in the full

transport model to determine the mass-balance for the solute at the boundary, and when computing mass balances for Richards' equation.

The net flux Q_i^D over the Dirichlet faces of a control volume on a Dirichlet boundary was computed in equation (3.66), namely

$$Q_i^D = -\sum_{j \in \mathcal{F}_j^D} A_j \left[\boldsymbol{q}\right]_j \cdot \boldsymbol{n}_{ij}. \tag{4.17}$$

To determine the flux over each of the control volume's Dirichlet faces, the net flux can be expressed as a weighted sum of the flux over the Dirichlet faces

$$Q_i^D = \sum_{f_j \in \mathcal{F}_i^D} w_j q_j, \tag{4.18}$$

where the weight at each face is proportional to the face's area

$$w_j = \frac{A_j}{\sum_{k \in \mathcal{F}_i^D} A_k}. \tag{4.19}$$

To form a vector of fluxes over every Dirichlet face in the domain, we note that the net flux for each control volume in equation (4.17) can be written as the inner product of a weight vector and a vector of the volumetric flux over each face of the domain. Using this observation and with equation (4.18), one can find a sparse matrix[7] A_{d} that can be multiplied against the global vector of volumetric face fluxes to determine a vector of fluxes over each Dirichlet face

$$\boldsymbol{q}_{\mathrm{d}} = A_{\mathrm{d}} \times \boldsymbol{q}_{\mathrm{faces}}, \tag{4.20}$$

where $A_{\mathrm{d}} \in \mathbb{R}^{n_d \times n_f}$, $\boldsymbol{q}_{\mathrm{d}} \in \mathbb{R}^{n_d}$ is a vector of fluxes at the Dirichlet faces and n_d is the number of Dirichlet faces in the mesh.

[7] The derivation is relatively simple, though tedious, and is not given here.

4.2.6 Residual Assembly

The residual assembly gathers the net flux at each face, which it adds to the accumulation term and the source term to determine the residual for each control volume. Take for example the residual function for the PR formulation at each node, defined in (3.70) as

$$f_i(t, \boldsymbol{\psi}, \psi') = n_i \psi'_i + Q^{\mathrm{w}}_i(\boldsymbol{\psi}) - \rho_i S^{\mathrm{w}}_i. \tag{4.21}$$

At this point, all of the terms in the residual function (4.21) have been computed, except for the net surface flux Q^w_i. The net flux Q^w_i is defined using (3.67) as the weighted sum of the flux over each face (4.14)

$$Q^{\mathrm{w}}_i(\boldsymbol{\psi}) = \sum_{j \in \mathcal{F}_i} \pm \frac{[q^{\mathrm{w}}]_j}{\Delta_i}, \tag{4.22}$$

where the sign of the term $\pm 1/\Delta_i$ is positive if the face normal $\boldsymbol{n}_j$ is outward facing relative to Γ_i, and negative otherwise.

The global operation of determining the net flux over all control volume surfaces can be performed given a sparse matrix $A_{\mathrm{gather}} \in \mathbb{R}^{n_n \times n_f}$ with nonzero entries defined

$$a_{ij} = \begin{cases} \frac{1}{\Delta_i}, & j \in \mathcal{F}_i \\ 0, & \text{otherwise} \end{cases}. \tag{4.23}$$

Then a vector of the net flux over each control volume surface can be computed using a sparse matrix-vector product as follows

$$\boldsymbol{Q}_{\mathrm{nodes}} = A_{\mathrm{gather}} \times \boldsymbol{q}_{\mathrm{faces}}, \tag{4.24}$$

where $\boldsymbol{q}_{\mathrm{faces}}$ is a vector of the flux $[q]_j$ over the faces and $\boldsymbol{Q}_{\mathrm{nodes}}$ is the vector of net fluxes.

Once the face fluxes have been gathered to each control volume using the SPMV product implied by equation (4.22), the other steps in forming the residual are straightforward to implement using vector operations, as will be

discussed in §5.6.6.

4.3 Domain Decomposition

Domain decomposition is a coarse-grained approach for sub-dividing the computational work between computational processes[8] (Chan and Mathew, 1994, Quarteroni and Vialli, 1999). Domain decomposition divides the domain into smaller regions called sub-domains, by assigning each node in the mesh to a sub-domain, as illustrated in Figure 4.5(a). Each sub-domain is then assigned to a computational process, which performs computations such as residual evaluation and preconditioning for the part of the mesh represented by the sub-domain.

The message passing library MPI (Pacheco, 1997) is used to execute the program in parallel. Multiple instances of FVMPor (one for each sub-domain) are started by MPI, each of which performs computation for its sub-domain on a separate computational units. In this thesis, the FVMPor instances are referred to as MPI *processes* (see §5.4 for more detail). Each process sees only its sub-domain, and uses MPI for communication with other instances.

In order for a process to perform local operations, information from the neighbouring sub-domains may be required. For example, to evaluate the residual for a node at the interface between sub-domains, the value of pressure head and concentration at nodes in the neighbouring sub-domain are required. This leads us to identify two classifications for nodes within a sub-domain:

- *local nodes* are those that were assigned to the sub-domain during domain decomposition.
- *halo nodes* are nodes in neighbouring domains from which information

[8]The computational units in a computer are divided into discrete sets called processes. A process may be a single CPU core, or multiple CPU cores and a GPU: see §5.4 for a detailed discussion.

is required, as illustrated in Figure 4.5(b).

We distinguish between the "global" view of the domain, and each sub-domain's "local" view. Globally, the values of nodal variables over the entire domain exist in a single *distributed* vector. Locally, each process stores only a portion of this vector: the values associated with local and halo nodes on that process' sub-domain. Figure 4.6(a) illustrates this for a distributed vector associated with the two sub-domain decomposition in Figure 4.5(a). Asynchronous communication between neighbouring sub-domains is used to update the value of variables at halo nodes by performing separate send and receive operations, as illustrated in Figure 4.6(b). This allows computation and communication to be overlapped by performing computation between

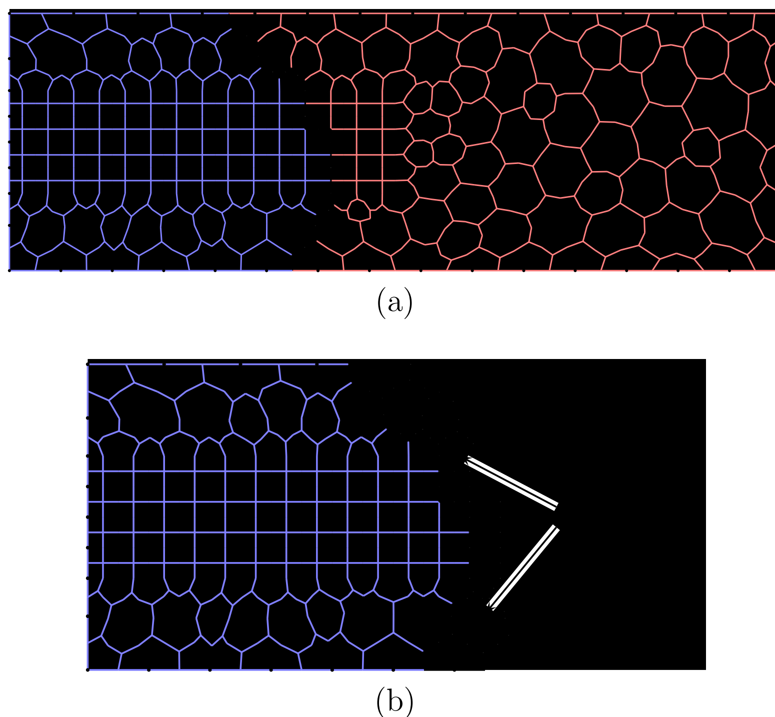


Figure 4.5: Domain decomposition of a mixed two-dimensional mesh into two sub-domains. (a) The global mesh, with the control volumes coloured according to the sub-domain, while control volume faces at the interface between the sub-domains are coloured black. (b) A sub-domain, with halo nodes assigned to the other sub-domain coloured black.

the send a receive phases of communication, as is discussed in detail in §5.4.

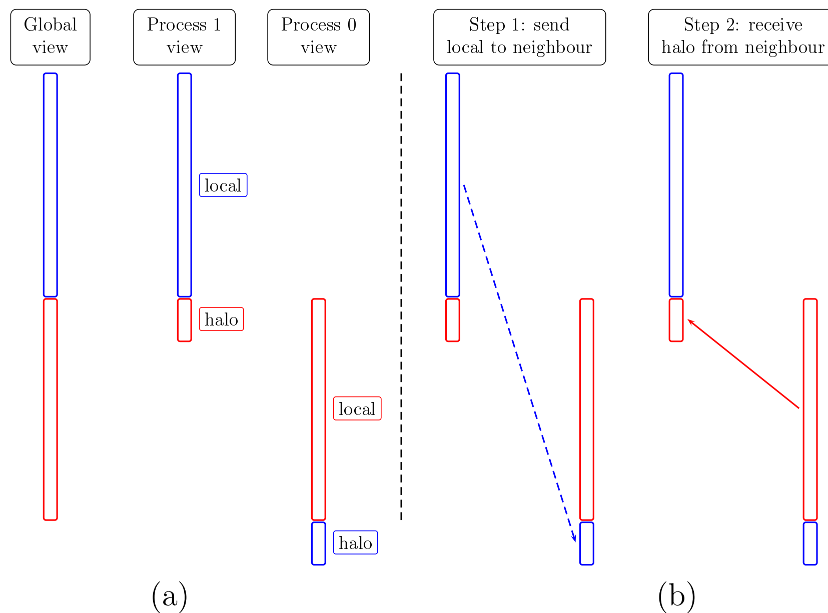


Figure 4.6: Storage of a distributed vector. The blue arrow in (b) is dashed because the communication is not completed until receive is finalised in Step 2.

4.4 Preconditioners

Preconditioning is required to ensure timely convergence of GMRES iterations. In this thesis, preconditioners based on sparse factorisations are used. The preconditioner is formed by computing and factorising an approximation to the iteration matrix G in (3.90). Then the preconditioner is applied at each inner iteration of GMRES, which requires the solution of two sparse triangular linear systems. In this section we discuss the matrix structure imposed on G by the domain decomposition, before looking at efficient methods for forming and applying the sparse factorisations.

4.4.1 The Global Matrix

The global iteration matrix is stored in a distributed manner, similarly to distributed vectors in §4.3. This introduces a block structure to the matrix, which is illustrated for a domain decomposition with four sub-domains in Figure 4.7. The global iteration matrix has one diagonal block for each sub-domain that represents connections between local nodes in each sub-domain, and off-diagonal blocks that represent the connections between local nodes and halo nodes.

Global preconditioning approaches such as Schur complement and additive Schwarz form a local preconditioner for each diagonal block, and use iterative approaches to introduce coupling between sub-domains (Saad, 2000, Chapter 13). In this thesis a block Jacobi approach is used, whereby a preconditioner is formed for each local block, and the connections between sub-domains are ignored. Any reference made here to the iteration matrix will imply the local diagonal block.

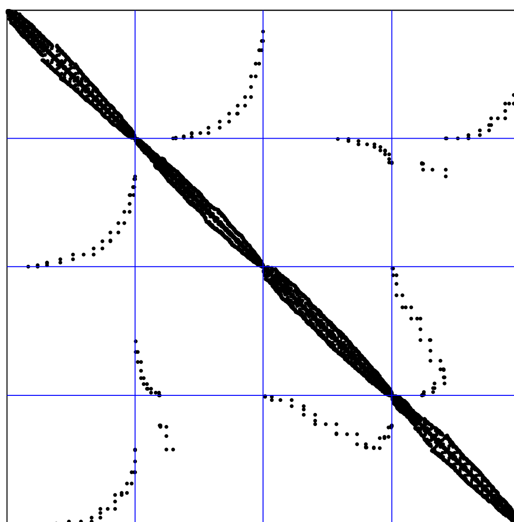


Figure 4.7: The global iteration matrix with four sub-domains.

4.4.2 Finding the Local Block

The first step in forming the local preconditioner, is to form an approximation to the local diagonal block of the iteration matrix. This can be performed efficiently using a small number of residual evaluations. To see how, we first revisit equation (4.25) for approximating the jth column of the iteration matrix G with a shifted residual evaluation:

$$G(:,j) = \frac{\boldsymbol{F}(\boldsymbol{y} + \epsilon \boldsymbol{e}_j, \boldsymbol{y}' + \epsilon\alpha \boldsymbol{e}_j) - \boldsymbol{F}(\boldsymbol{y}, \boldsymbol{y}')}{\epsilon} \tag{4.25}$$

where references to time t have been dropped for clarity, and the vectors $\boldsymbol{y}$ and $\boldsymbol{y}'$ contain variables associated with nodes local to the sub-domain. If the columns of G were to be computed separately, the cost of performing the shifted residual evaluations would be prohibitively expensive for large matrices.

However, it is possible compute multiple columns of the matrix using a single residual evaluation by analysing the connections between nodes. The residual function for each node is dependent only on the value of variables at the node and its immediate neighbours. This means that the shift in (4.25) can actually be performed for multiple variables at once, provided these variables correspond to nodes which have no common neighbours. That is, for an independent set of variables $\mathcal{J}$, we can evaluate (4.25) with the following shift vector

$$\boldsymbol{e}_s = \epsilon \sum_{j \in \mathcal{J}} \boldsymbol{e}_j, \tag{4.26}$$

to determine the set of columns $G(:,\mathcal{J})$.

A multi-colouring of the nodes in the mesh is used to identify sets of independent variables. The multi-colouring assigns a colour to each node of the mesh, such that no two nodes with the same colour have common neighbours. The simple greedy algorithm in Algorithm 4.5 is used (Saad, 2000), and only needs to be computed once at startup, because the sparsity pattern of the matrix is fixed throughout the course of a simulation.

Algorithm 4.5: Greedy multi-colouring algorithm used to find the minimal independent column set.

```
Input  : adjacency set for each node adj
         number of nodes in mesh N
Output: vector of colours for each node colour
mark = array(N, -1);
colour = array(N, N-1);
maxcol ← 0;
for i = 0 : N − 1 do
    for j ∈adj(i) do
        mark [colour [j]]← i;
    mincol ← 0;
    while mincol < maxcol and mark[mincol] = i do
        increment(mincol);
    if mincol =maxcol then
        increment(maxcol);
    colour[i] ← mincol;
```

We note that the multi-colouring in Algorithm 4.5 does not account for dependencies on values at 2up nodes introduced by flux limiting. Because of these 2up dependencies, the iteration matrix for flux limiting is more dense, and requires considerably more computational effort to form and use. Indeed, previous investigations have shown that the computational overhead of forming and using the more dense matrix outweigh any gains in faster convergence rates (Forsyth et al., 1996). This is especially true for matrix-free methods, where the iteration matrix is not used directly, and the matrix-vector products in the GMRES method are approximated using shifted residual evaluations that include the 2up information (Moroney and Turner, 2006). Hence, the local block is always formed using only nearest neighbour information in this thesis.

4.4.3 Preconditioning the Local Block

To form a preconditioner based on sparse factorisation of the iteration matrix G, we seek two triangular matrices, L and U, such that

$$G \approx LU. \tag{4.27}$$

To apply the preconditioner to a vector $\boldsymbol{v}$, the following linear system is solved

$$LU\boldsymbol{z} = \boldsymbol{v},$$

which is performed by solving two triangular linear systems as follows

$$\text{forward substitution} \qquad \boldsymbol{w} = L^{-1}\boldsymbol{v}, \tag{4.28a}$$

$$\text{backward substitution} \qquad \boldsymbol{z} = U^{-1}\boldsymbol{w}. \tag{4.28b}$$

Numerical experiments in Chapter 7 reveal that the dominant cost of preconditioning for the problems investigated in this thesis is in the application phase in equations (4.28a) and (4.28b). This is because the preconditioner is formed relatively infrequently[9], whereas it is applied in every inner iteration of the GMRES method. For this reason, a method for applying the preconditioner that is amenable to GPU implementation is the focus of the rest of this chapter.

The solution of the triangular linear systems using the forward and backward substitution in (4.28) is an inherently serial operation if the triangular factors are dense[10]. However, a multi-colouring of the matrix graph associated with the iteration matrix can be used to parallelise the solution of sparse triangle

[9] The preconditioner is reformed periodically by IDA (Figure 4.1 shows the criteria for recomputing the preconditioner during the IDA time step loop) when the information in the iteration matrix becomes out of date. When this occurs, the matrix is recomputed using forward differences in equation (4.25), then the matrix is factorised.

[10] Take for example the forward substitution phase in (4.28a): to find the value of w_j, the values of w_i, $\forall i < j$ must first be computed.

equations (Saad, 2000, Li and Saad, 2010, Naumov, 2011, Heuveline et al., 2011b). Given a colouring of the matrix graph with p colours, the iteration matrix G can be reordered into p blocks by colours. The reordered system matrix then has the following block structure

$$G(\mathsf{q},\mathsf{q}) = \begin{bmatrix} D_1 & & \boxed{U_p} & \\ \boxed{L_2} & D_2 & \boxed{U_{p-1}} & \\ & \ddots & & \\ \boxed{L_{p-1}} & & D_{p-1} & \boxed{U_2} \\ & \boxed{L_p} & & D_p \end{bmatrix}, \tag{4.29}$$

where the D_i blocks are diagonal, and the index vector $\mathsf{q}(1:N)$ is the permutation for the new row and column ordering.

Given the matrix stored in the permuted form (4.29), the rows in each block may be processed simultaneously during each of the forward and backward substitution phases (Saad, 2000, Chapter 11). For each phase, the solution for the variables in each block is determined using sparse matrix-vector multiplication and vector-vector operations in Algorithm 4.6. For example, the forward substitution phase uses one matrix-vector product and one axpy operation[11], both of which are operations that can be performed efficiently in parallel. We note that Algorithm 4.6 does not explicitly use the permutation vector q because the matrix is already stored in block form (4.29); instead it uses an index array $\mathsf{idx}(i), i = 1, 2, \ldots, p+1$ that points to the beginning of the ith colour in q.

Three sparse factorisations are investigated as preconditioners in this thesis, a full sparse LU factorisation, and two incomplete (ILU) factorisations[12]. The first incomplete factorisation is a dual threshold incomplete factorisation (ILUT) (Saad, 1994), and the second is an incomplete factorisation with no fill in (ILU(0)). The full LU and the ILUT factorisations introduce fill

[11]The axpy operation is a BLAS operation for finding a linear combination of two vectors, $\boldsymbol{y} \leftarrow a\boldsymbol{x} + \boldsymbol{y}$, where a is a scalar.

[12]All three of the sparse factorisations are in the Intel MKL. See §5.8 for details.

Algorithm 4.6: Forward and backward substitution for sparse triangular factors that have been reordered according to a multi-coloring.

Input : index idx that points to the start of each block
lower triangle blocks L_i
upper triangle blocks U_i
diagonal blocks D_i as vectors
number of colours p
right hand side vector b of length N
Output: preconditioned vector z

$\mathsf{z} \leftarrow \mathsf{b}$
for $i = 2 : p$ **do**
  $r_1 \leftarrow \mathsf{idx}[i]$;
  $r_2 \leftarrow \mathsf{idx}[i+1]$;
  $\mathsf{z}(r_1 : r_2 - 1) \leftarrow \mathsf{z}(r_1 : r_2 - 1) - \mathsf{L}_i \times \mathsf{z}(1, r_1 - 1)$;
for $i = p : -1 : 1$ **do**
  $r_1 \leftarrow \mathsf{idx}[i]$;
  $r_2 \leftarrow \mathsf{idx}[i+1]$;
  if $i < \mathsf{p}$ **then**
    $\mathsf{z}(r_1 : r_2 - 1) \leftarrow \mathsf{z}(r_1 : r_2 - 1) - \mathsf{U}_i \times \mathsf{z}(r_2, \mathsf{N})$;
  $\mathsf{z}(r_1 : r_2 - 1) \leftarrow {\mathsf{D}_i}^{-1} \times \mathsf{z}(r_1 : r_2 - 1)$;

in, or additional nonzeros, to the triangular factors L and U. The ILU(0) preconditioner does not introduce fill-in, so that the sparsity pattern is of the triangular factors is identical to that of the iteration matrix, which is fixed throughout the simulation.

To use the method for applying the preconditioner in Algorithm 4.6, an analysis phase must first be performed to compute the multi-colouring of the sparse factors, which imposes a considerable computational overhead if it is performed each time the preconditioner is formed. To avoid frequently recomputing the multi-colouring, we seek sparse factorisations for which the sparsity pattern is known *a priori* and is fixed throughout the simulation. This is possible for the ILU(0) preconditioner, which has the same (fixed) sparsity pattern as the iteration matrix. Furthermore, the multi-colouring computed in Algorithm 4.5 can also be used for the ILU(0) factors.

4.5 Conclusions

This chapter introduced the algorithms and data structures used in FVMPor to implement the CV-FE spatial discretisation from Chapter 3 for solution in IDA. The chapter started with an overview of the internal steps taken inside an IDA time step.

The flat data model, that is be amenable to implementation on data parallel compuation architectures was then introduced for the residual evaluation. This was followed by a detailed discussion of each the steps involved in performing the residual evaluation from §3.2 in §4.2. Care was taken to emphasise how each step in the residual evaluation can be expressed in terms of data parallel vector and sparse matrix-vector operations.

Then, the domain decomposition approach used to obtain coarse-grained parallelism was introduced, with a brief introduction to distributed vectors and asynchronous communication. Finally, the block Jacboi preconditioner for distributed matrices was introduced, along with a method for efficiently solving linear systems with sparse triangular matrices in parallel.

To summarise, this chapter has shown the specific steps that must be taken to solve, and shown how each step can be expressed using data-parallel operations that can be implemented efficiently on multi-core and many-core hardware. The following chapter will discuss the low-level details of implementing these operations using OpenMP for multi-core CPUs and the CUDA language for GPUs.

Chapter 5

Implementation on GPU Clusters using C++, MPI and CUDA

This chapter presents a description of the implementation of the algorithms developed in Chapter 4 to run efficiently in on multi-core and GPU computer sytems. We start with a discussion about general-purpose computing on GPUs, with a focus on the issues that arise when using unstructured meshes and an implicit time stepping scheme. Then the vectorlib library, which implements data parallel operations on both multi-core CPU and GPU hardware will be introduced. The rest of the chapter will discuss in detail the implementation of the CV-FE discretisation, the IDA timestepping library and preconditioners.

Of the work presented in this Chapter, there is considerable novelty in the methods employed to implement the solution process in a hardware-agnostic manner. This requires that we address two tasks that are challenging on the GPU: discretistion on unstructured meshes; and implicit time stepping methods. The simple syntax for vector and indexing operations provided by vectorlib library in §5.2 facilitates clear, concise and efficient code that runs on both CPU and GPU. Using vectorlib, the CV-FE discretisation on unstructured meshes is implemented using flat data structures and index

vectors in §5.6. The novel renumbering scheme for nodes, edges and faces in §5.7 improves performance of the implicit indexing that arises from the unstructured meshes. Efficient implicit time stepping is acheived on GPU clusters using a matrix-free Newton-Krylov method, performed using a modified version of IDA in §5.5, which combined with an efficient application of an ILU(0) preconditioner in §5.8, sees the computationally-intensive parts of the implicit time stepping process implemented entirely on the GPU.

5.1 Using GPUs as Computational Accelerators

Graphics processing units (GPUs) were, as their name suggests, originally designed for the specific task of performing the arithmetically intense, data parallel computations associated with graphics rendering. The data parallel operations for which GPUs are specialised are often encountered in scientific computing, and other fields such as image processing. This has lead to an explosion of interest in using GPUs for general purpose computing.

Dedicated GPU accelerators used in high performance computing have separate memory to the main system memory, and communicate with the CPU and main memory via PCI-Express bus. The GPU and its memory are labelled as the *device*, and the CPUs and the main system memory are referred to as the *host*. Figure 5.1 describes a high-performance GPU desktop computer similar to the one used in our testing, with two GPU devices and two sockets, each with four CPU cores[1].

An important consideration when designing computer processors is memory latency. Memory latency occurs because the time taken for a memory request to be served by is typically in the order of hundreds of processor clock cycles.

[1]The test machine used for this thesis had two four-core Intel Xeon E5620 CPUs rated at 2.4Ghz with 12 Gigabytes of DRAM, and two NVIDIA Tesla C2050 GPU cards each with 3 Gigabytes of DRAM

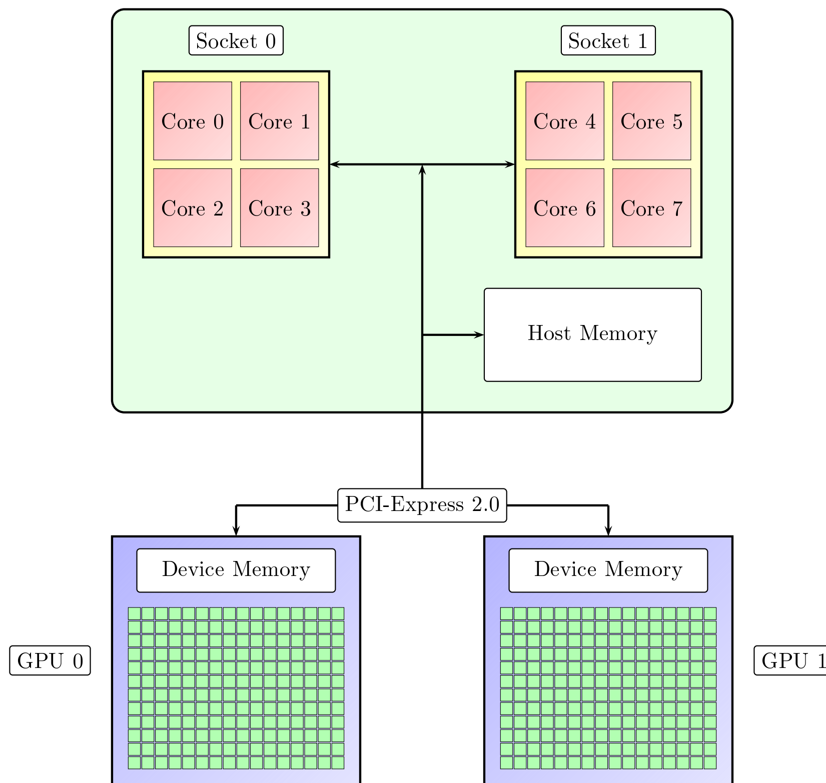


Figure 5.1: Schematic of GPU workstation with two GPUs and two sockets, each with 4 cores.

CPUs and GPUs differ in how they overcome memory latency according to the type of operations they are designed to perform.

CPUs are designed to general purpose computing, so that they must perform well for inherently serial computational tasks. In this context, memory latency is addressed by adding large on-chip caches to buffer memory transfers between the processor and off-chip memory, and performing sophisticated prediction and memory prefetching to hide latency. The downside of this approach is that a large proportion of the silicon, and power consumption, on a CPU is devoted to cache and memory control hardware.

The graphical operations for which GPUs were designed are data parallel, or single instruction-multiple data (SIMD), operations that repeat the same

operation on structured data sets. GPUs are designed under the assumption that memory access will follow regular patterns, common in graphics applications. This allows chip designers to remove much of the memory controlling hardware, and replace it with many cores for parallel processing. Thus, GPUs can outperform CPUs by orders of magnitudes for data-parallel algorithms, however the opposite is also true: algorithms that are not well-suited to GPU implementation perform very poorly. For this reason GPU codes are particularly sensitive to Amdahl's law, which states that any gains in parallel sections of an algorithm will be limited by any parts of the algorithm that can not be parallelised.

5.1.1 Fermi Architecture

The Fermi architecture is the third-generation GPU for general purpose computing manufactured by NVIDIA. Fermi devices have either 480 or 512 cores and global DRAM that can be read and written by all cores. For example, the Tesla C2050 cards used in this work have 480 cores and 3 GB of global memory.

The cores on the GPU are packaged into units called streaming multi processors (SMs) that each contain 32 cores. Each SM schedules threads in groups of 32, called warps, which are launched concurrently to hide memory latency. In addition to the cores, each SM has 64 kB of on chip memory that can be used as either shared memory or, unlike previous GPU generations, L1 cache. The L1 cache is a new feature of the Fermi architecture, that improves the performance of global memory access (NVIDIA, 2009).

One of the main motivations for the addition of L1 cache on the Fermi architecture was to improve the performance of indirect indexing in global memory. This is an important development for implementing the unstructured mesh code for FVMPor on the GPU, because indirect indexing is used extensively to map between the nodes, edges and control volume faces in the mesh. Specifically, the flat data model introduced in §4.2 uses index vectors

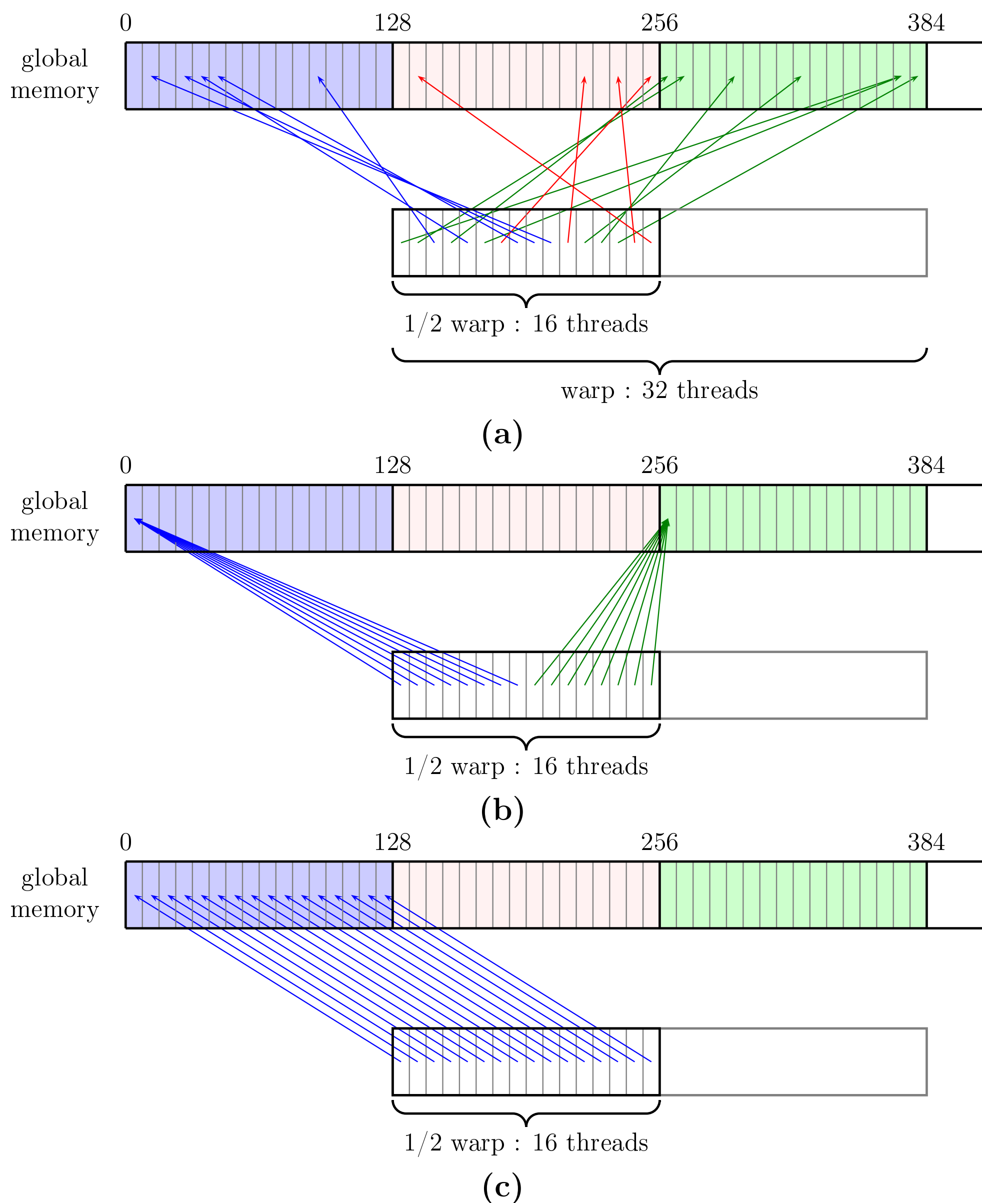


Figure 5.2: Caching of global memory access for indirect indexing, where different colours represent different cache lines. Memory requests are always performed for 128-byte blocks, which for doubles equates to 16 threads, or half a warp. In (a) three cache lines are fetched from global memory to satisfy the memory requests from the half warp; in (b) two cache lines are read because the two read values lie in different cache lines of global memory; and in (c) one cache line is read.

that index into vectors of node, edge and face values. An example of such an operation in the MATLAB programming language is:

```
u = v(p);

% which is equivalent to
for i=1:length(p)
  u(i) = v(p(i));
end
```

where the index vector `p` contains indices into `v`.

Despite the addition of L1 cache, the performance of indirect indexing on the Fermi architecture is sensitive to the memory access patterns implied by the indices. The remainder of this section will investigate the restrictions placed on the indices to obtain good performance, which will inform the derivation of the numbering scheme for nodes, edges and faces in the dual mesh that is presented in §5.7.

The role of L1 cache in indirect indexing on a Fermi GPU is illustrated in Figure 5.2. When indexing arrays of double precision floating point numbers (which we will refer to as *doubles*), as is the case in this work, memory requests are issued per half warp, or set of 16 threads. Each memory request fetches a cache line of length 128 bytes from global memory to L1 cache, where the threads in the half warp can simultaneously access the data. The SM issues as many memory requests for cache lines as needed to load all the values required by the threads in the half warp. Figure 5.2 illustrates that the order of the indices has a large impact on the number of cache lines that have to be read by each half warp: if the index values are scattered as in Figure 5.2(a), a large number of of cache lines have to be read; and if contiguous indices are closely clustered or contiguous as in Figure 5.2(c), the number of cache lines read from global memory is minimised.

Memory bandwidth is the main bottleneck on the GPU because fetching one cache line from global memory takes hundreds of clock cycles (NVIDIA, 2011a). And so, minimising the number of memory requests that have to be

issued by each half warp is the key to optimising indirect indexing[2]. To do this we must ensure that the index vectors reference locations that are close together, or clustered, in memory.

The indices used in the CV-FE method map between nodes, edges and faces in the unstructured mesh. The level of clustering in the index vectors can be changed by renumbering the nodes, edges and faces. In this thesis, this is achieved using a numbering scheme based on analysis of the adjacency graph of the finite element mesh, that is discussed in §5.7.

5.2 The vectorlib Library

The vectorlib library[3] is a minimal C++ template library for linear algebra with MATLAB-like syntax that was adapted as part of this research program to provide a hardware-agnostic interface to optimised vector and matrix operations used in FVMPor. A brief overview of the capabilities of the library is presented in this section, with an emphasis on features that facilitate writing high-level linear algebra code that runs efficiently on a range of architectures. A basic knowledge of C++, particularly templates and the standard library containers is assumed. Recommended references for background reading are the guide to the C++ language by Stroustrup (1993), and the thorough treatment of C++ templates by Vandevorde and Josuttis (2003).

A key template in vectorlib is the Vector class template:

```
template<typename T, typename Coord = DefaultCoordinator<T> > class Vector;
```

The template parameters T and Coord represent the scalar type (e.g. **double**) and the so-called coordinator respectively. The coordinator plays a similar role to that of the allocator in the C++ standard library. Indeed, it shares many of the same roles, including allocating and deallocating storage for the

[2]Closely clustered indices in also improve memory reuse on the SM level, because the same cache is shared by all of the concurrently running warps on an SM.

[3]The vectorlib library was originally developed by Timothy Moroney in 2009, at the Queensland University of Technology.

container, as well as "coordinating" access to that storage vis the provided pointer and reference types. However, in vectorlib, the coordinator plays an additional role in dispatching to appropriate hardware-optimised implementations of linear algebra and associated routines.

The default coordinator allocates memory on the host. Hence, a double-precision vector residing on the host can be defined simply as

```
Vector<double> v_host(/* constructor arguments */);
```

The class template Vector provide numerous constructors, for creating empty vectors, vectors filled with given values, vectors copied from other vectors, and so on. All of these constructors forward to the coordinator to perform the actual allocation and initialisation.

This separation of interface and implementation allows vectors residing on the device to be created and manipulated, by simply defining an appropriate coordinator type. In vectorlib a vector residing on the device is defined by using the device coordinator, gpu::Coordinator. The key differences between gpu::Coordinator, and the default coordinator are

1. gpu::Coordinator forwards calls such as memory allocation and fill to GPU (device) implementations, rather than standard library (host) implementations

2. gpu::Coordinator utilises auxiliary "smart-pointer" and "smart-reference" types

The forwarding of calls to device implementations is quite natural. Device manufacturers supply platforms and associated libraries with their hardware, which provide implementations of standard primitives such as allocating and filling. In vectorlib these are wrapped by functions with names such as allocate and fill which provide a common interface to this functionality. This permits different platforms, such as CUDA (NVIDIA, 2011a) and OpenCL (Khronos Group, 2010) to be used, simply by implementing these wrapper functions to forward to the appropriate hardware-specific library calls.

The auxiliary smart pointer and smart reference types abstract away the details of GPU memory access and traversal, thereby facilitating the familiar model of traversal and element access through pointer arithmetic and dereferencing. This abstraction ensures that expressions such as

```
v(i) = q;
```

where v is a vector type, i is an index and q is a value, remain valid even when v has been allocated on the device. Naturally, when working on the device it is usually preferable to process entire vectors at a time, rather than individual elements. However, individual element accesses such as the above do sometimes occur in code (see, for example, updating the source term in the residual function in §5.6.6). In these situations, the overhead of individual store and fetch operations is not significant, and hence it is beneficial to the programmer that such code need not be re-written when running on the device.

The greatest benefit however comes when vectors are processed as a whole unit, and vectorlib provides a rich set of such operations that benefit from acceleration whether implemented on the host or the device. As an example, consider the expression, in MATLAB notation

```
v = v + x .* y;
```

where v, x and y are all vector types. In C++ there is no equivalent to MATLAB's element-wise multiplication operator .*, but there is a compound assignment operator for addition, +=. The equivalent expression using vectorlib is

```
v += mul(x,y);
```

Once again, the coordinator is involved in dispatching to the correct implementation of operations such as these, depending on whether the operands reside on the host or the device.

The previous compound assignment statement can also be performed where all three vectors are *indirectly* indexed:

```
v(p) += mul(x(q), y(r));
```

That is, the expressions p, q and r are *vectors* of indices, representing non-contiguous subranges of the vectors v, x and y respectively.

Sub-ranges, with strided access, are also supported:

```
v(0,2,end) = x; // assign the values in x to every second entry in v
v(all) = 3; // set every value in v to equal 3
```

Another supported operation that occurs frequently is the copying of a vector from host to device, or from device to host. Such a copy would be triggered, for example, by the following code.

```
typedef Vector<double> HostVector;
typedef Vector< double, gpu::Coordinator<double> > DeviceVector;

HostVector host_v(n);
// Fill vector on host
for (HostVector::difference_type i = 0; i < n; ++i)
    host_v(i) = /*initialiser*/

// Copy to device
DeviceVector device_v = host_v;
```

This is the typical idiom for non-trivial initialisation of a vector on the device. It is assumed that the commented **/*initialiser*/** in fact represents a complex operation that is not straightforward to perform efficiently on the device. Instead, the initialisation takes place on the host, and the resulting vector is then copied from host to device.

5.3 Sparse Matrices

Sparse matrix-vector multiplication (SPMV) is used throughout the solver, for example for shape function interpolation in §4.2.2 and in the application of the multi-colour optimised sparse triangle solves in Algorithm 4.6. This approach is taken, instead of writing custom optimised CPU and GPU routines for such operations, because optimised SPMV codes are provided by hardware vendors. On the CPU, the Intel Math Kernel Library (MKL) (Intel, 2010) is used, and the GPU implementation uses the CUSPARSE library pro-

vided by NVIDIA (NVIDIA, 2011c). By writing a vectorlib wrapper around each of these implementations, future generations of CPU and GPU hardware will be supported as the vendors add support to the their libraries.

The sparsity patterns of the matrices in used for gather operations in the residual evaluation are determined by the structure of the mesh. Because the mesh is fixed, the structure of the matrices is also fixed, and the sparsity pattern and memory used by the matrices can be precomputed and allocated at startup. The coefficient matrices are initially generated on the host in compressed sparse row (CSR) format when the mesh is loaded. If a GPU is used, the matrices are then copied to device memory so that computations can be performed entirely on the GPU with no further copying of memory between host and device.

Storing the matrices on the GPU in this manner limits the size of meshes that can be processed, due to the storage requirements of the matrices, which increase in size with the number of nodes and elements in the mesh. It is possible to minimise the memory footprint of the matrices by noting that some of the matrices have the same sparsity pattern, for example the interpolation matrices for scalar and gradients. A mechanism that allows index information to be shared by multiple matrices with the identical sparsity patterns is provided, to minimise the memory costs associated with storing the matrices.

The GPU devices used in this work have 3 Gigabytes of memory, which is enough for single GPU to store index and sparse matrix information for a two-dimensional mesh of about 1,000,000 nodes, and three-dimensional meshes of 200,000 nodes. The size of meshes that can be processed can be increased by using multiple GPUs. Mesh sizes will also increase as the amount of memory available on devices increases (the latest C2090 model of the Fermi compute devices at the time of writing doubles the available memory to 6 Gigabytes), and if there is a move towards unification of host and device memory spaces.

5.4 Domain Decomposition

Domain decomposition implements course-grained parallelism by assigning the computational work to separate computational resources by means of sub-dividing the mesh. In §4.3 the method of domain decomposition was introduced, and in this section, three key aspects of the low level implementation of domain decomposition are considered.

- How the hardware resources are assigned to MPI processes. This is especially important in a heterogeneous computational environment, where different types of computational unit such as CPU cores and GPUs are available.
- The method used to perform the domain decomposition of the mesh.
- How communication between the sub-domains is performed. We use the widely-used standard message passing library MPI, for which we consider how to overlap communication and computation, and how to communicate data stored in device memory.

5.4.1 Sub-Dividing the Computer Into Processes

There are a variety of ways that the computational units can be assigned to processes on a heterogeneous computer such as that illustrated in Figure 5.1, with CPU cores and GPU co-processors as the basic computational units. In this thesis two different approaches will be investigated, with the message passing library MPI (Pacheco, 1997) used for communication between the sub-domains in each case:

- **Hybrid MPI-OpenMP**: **Hybrid MPI-OpenMP-CUDA**: To accommodate GPUs, the hybrid MPI-OpenMP model is extended by assigning a GPU to each process. This is illustrated in Figure 5.3, where

four CPU cores and one GPU device are assigned to each sub-domain in a two sub-domain decomposition.

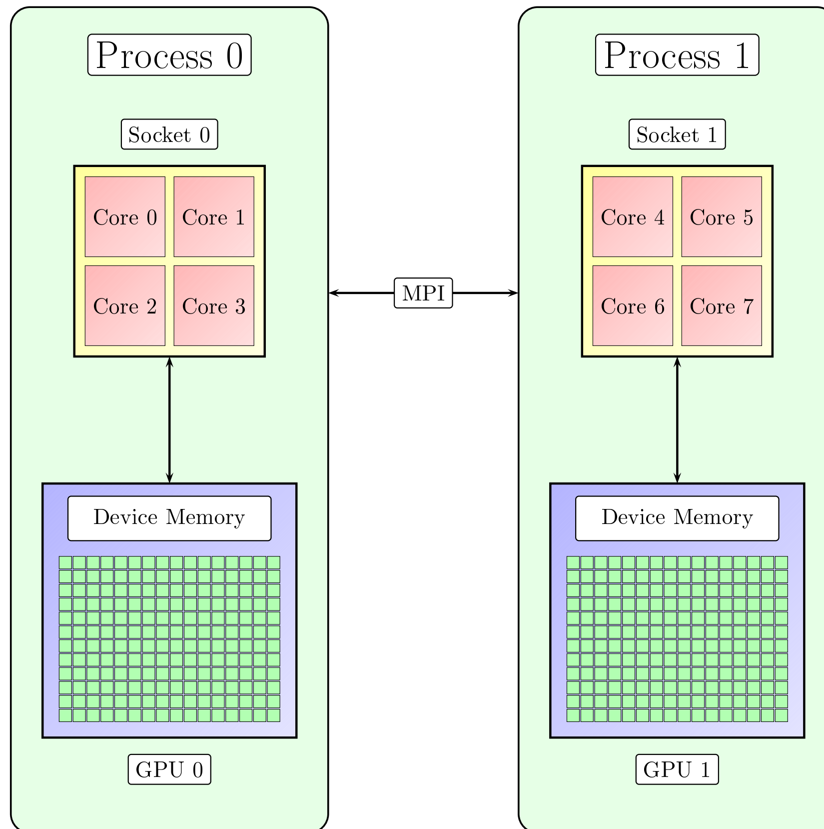


Figure 5.3: Allocation of computational resources in the workstation in Figure 5.1 for a domain decomposition of two sub-domains.

5.4.2 Mesh Generation and Domain Decomposition

The finite volume meshes are generated, and the domain decomposition performed, in a separate preprocessing stage. The open source mesh generation program Gmsh (Geuzaine and Remacle, 2009) is used to generate the finite element mesh. Domain decomposition is then performed in a stand-alone program that uses the ParMetis library (Karypis and Kumar, 1999) to perform the decomposition. The mesh for each sub-domain, and the connection

information between sub-domains, is written to file. When the simulation code starts, each process reads the part of the mesh corresponding to its sub-domain from disk, generates the dual mesh for the sub-domain (see §3.1.2), then builds data structures for MPI communication based on the connection information.

5.4.3 Communication Between Processes

In the inexact Newton-Krylov solver communication is performed on distributed vectors. A distributed vector is a vector of nodal values, for example concentration values, defined at every node in the domain. Distributed vectors can be defined in two ways: with and without overlap. Storing without overlap stores only values that correspond to local nodes on each process. A distributed vector with overlap also stores copies of values at halo nodes in addition to values at local nodes, as illustrated in Figure 5.4.

Two forms of communication are performed with distributed vectors in FVMPor and IDA. The first form consists of reduction operations such as dot products and norms. A mechanism is provided by MPI for performing reduction operations, whereby each process first performs the reduction operation for local values, then a global reduction is performed by all processes making an MPI reduction call.

The second type of communication is updating the halo values between neighbouring sub-domains, or *halo communication*. Two C++ classes are used to implement halo communication. The first is the Pattern class, which describes the connectivity of local and halo nodes between sub-domains. The Communicator class performs communication according to the pattern described by the Pattern class.

The definition of the Pattern class is shown in Listing 5.1. The class stores a list of neighbouring sub-domains, and for each neighbour a list of local variables to be sent to the neighbour, and a list of halo values received from

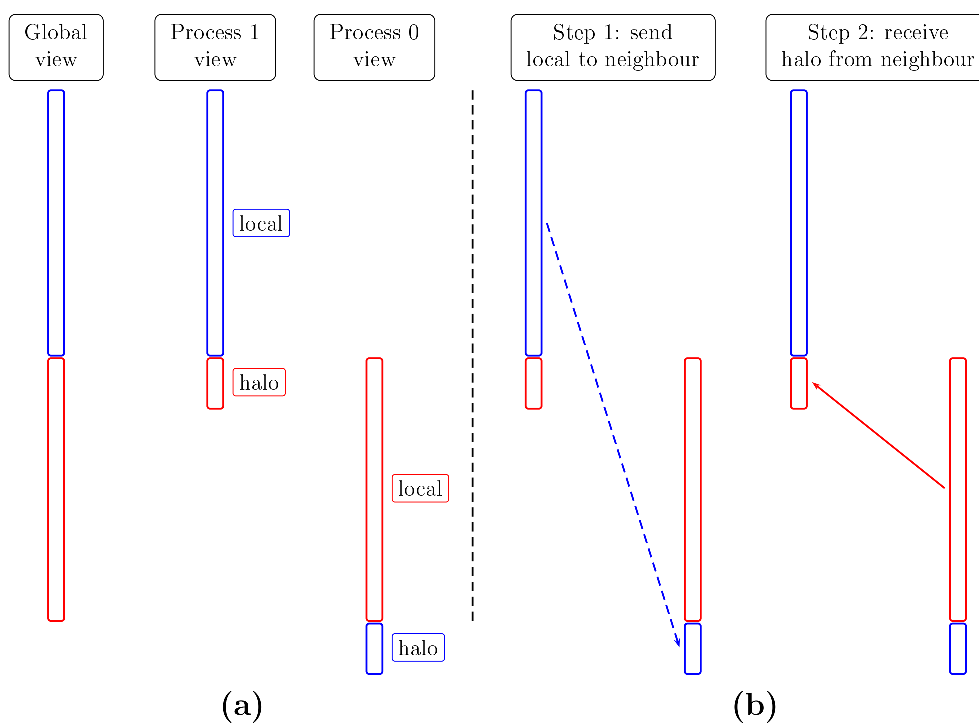


Figure 5.4: (a) Storage of a distributed vector with overlap for two-subdomains. Each process stores entries in the vector that correspond to its local nodes, as well as the value of halo nodes from neighbouring sub-domains, which are stored after the local values. (b) Asynchronous communication to update the halo values on neighbouring sub-domains. The blue arrow for the send in Step 1 is dashed because the communication is not completed until receive is finalised in Step 2.

the neighbours. The Pattern class also provides a public interface to this information.

The definition of the Communicator class is shown in Listing 5.2(a). The constructor for the class uses the communication pattern implied by the Pattern class. The Communicator class is templated on a vectorlib coordinator and a storage type. The storage type, for example double, specifies the storage class of the distributed vectors to be communicated. Listing 5.2(b) shows a simple example of how the Communicator is used to handle communication of two distributed vectors. Each vector, which has the same storage class as the Communicator, is first added to the Communicator so that the Communicator

```
class Pattern{
public:
  Pattern(mpi::MPICommPtr);

  // return a lits of neighbour sub-domains
  const std::vector<int>& neighbour_list() const;
  // number of neighbouring sub-domains
  int num_neighbours() const;
  // get the id of the nth neighbour
  int neighbour(int n) const;
  // get index of local nodes to send to sub-domain n
  const std::vector<int>& send_index(int n) const;
  // get index of halo nodes to receive from sub-domain n
  const std::vector<int>& recv_index(int n) const;

  // add a neighbour to the pattern
  void add_neighbour( int n,
                      std::vector<int>& send,
                      std::vector<int>& recv  );

private:
  mpi::MPICommPtr comm_;
  // list of neighbour sub-domains
  std::vector<int> neighbours_;
  // lists of local/halo nodes to send/receive for each neigbbour
  std::map<int, std::vector<int> > send_index_;
  std::map<int, std::vector<int> > recv_index_;
};
```

Listing 5.1: Definition of Pattern class.

can allocate communication buffers for the vector. Once a vector has been added, the Communicator can perform asynchronous communication of the vector's halo information. The communication is explicitly broken into asynchronous send and receive phases so that communication and computation can be overlapped.

The Communicator is also templated on a vectorlib coordinator because the location of the vector data affects how the communication is performed. For vectors allocated in host memory using a host coordinator, the send and receives are performed directly to and from the host memory using predefined MPI_Type_indexed data types (Pacheco, 1997), that do not require buffering.

To perform halo updates for vectors that are stored on the GPU, two small buffers are required: one in host memory and one in device memory. To perform a send, the local values that are to be sent to neighbours are gath-

```
template <typename Coord, typename Type>
class Communicator{
    public:
        Communicator( const mesh::Pattern& );

        // add and remove vectors to and from the communicator
        int vec_add(TVec&);
        int vec_remove(int);

        // communication
        int recv( int );
        int send( int );
        int recv_all();

        // ...
};
```

(a)

```
// create a communicator for vectors of double distributed by mesh nodes
mpi::Communicator<CoordDeviceDouble, double> comm(mesh.pattern());

// create device vectors:
// mesh.nodes() is the combined number of local+halo nodes
TVecDevice pressure(mesh.nodes());
TVecDevice concentration(mesh.nodes());

// add vectors to the communicator
int ptag = comm.vec_add(pressure);
int ctag = comm.vec_add(concentration);

// initialise vectors by copying in vectors of length mesh.local_nodes()
// into the local part of the distributed vectors
pressure(0,mesh.local_nodes()-1) = pressure_local;
concentration(0,mesh.local_nodes()-1) = concentration_local;

// send
comm.send(ptag);
comm.send(ctag);

// ... do work on local values

// finish all pending receives so both vectors have up-to-date halo values
comm.recv_all();

// ... do work on halo values
```

(b)

Listing 5.2: (a) The interface to the Communicator class. (b) An example of how communication is performed for distributed vectors of pressure and concentration values.

ered into the device buffer, which is copied to host memory before sending. The same host and device buffers are used to receive values from neighbours, copy them to the device, then insert them into the global vector. Typically, the overhead of performing the intermediate copies between host and device memory are very small, because only the local/halo values that are sent/received are copied, and these are relatively few compared to the total number of nodes in a sub-domain.

5.5 The IDA Library

The matrix-free Newton-Krylov method used by IDA has three significant computational operations, each of which operates on distributed vector and matrix data. Residual function evaluation and the preconditioner represent the most significant computational overheads, and both are implemented in user-supplied routines that are discussed later in §5.6 and §5.8. The remaining operations in the Newton-Krylov solver are the vector operations (equivalent to Level 1 BLAS) applied to distributed vectors in IDA.

The vector operations in IDA are implemented in a separate library called NVector, which provides an interface to the hardware implementation of memory allocation and operations for the vector data. The abstraction of the hardware implementation via NVector makes it possible to add support for GPUs to IDA simply by writing a version of NVector that implements all the vector-vector operations on the GPU.

NVector provides an interface that hides the low-level memory and hardware implementation from the calling code. The only data passed between the calling code and NVector are scalar values, such as the return values of reduction operations like dot products. This design allows IDA to be adapted to different hardware platforms by using a version of NVector that supports that hardware.

Two implementations of NVector are provided with the standard IDA library: a serial implementation; and a implementation for distributed memory machines that uses MPI for communication called NVector_Parallel. To add support for GPU clusters to IDA in FVMPor, the parallel implementation NVector_Parallel was altered to perform local vector operations on the GPU.

Reduction calls were the most challenging part of NVector to implement on the GPU, because of the challenges inherent in reducing an array of length N to a single scalar value over many threads (Harris, 2007). Once the local reduced value had been computed on the GPU, the scalar value is copied to the host and the global reduction is performed using the same MPI reduction as the original version of NVector_Parallel.

For the non-reduction vector operations, such axpy[4], the CUBLAS library[5] was used when possible, otherwise CUDA kernels were written and compiled separately using CUDA. For example, CUBLAS provides routines for allocating device memory that are a like-for-like replacement for the equivalent CPU code in Listing 5.3. An example of an operation in NVector for which there is no direct equivalent in BLAS is the inverse in Listing 5.4(a), which was replaced with the CUDA kernel in Listing 5.4(c).

The MPI-CUDA implementation of NVector developed in this thesis can be downloaded from GitHub at `github.com/bencumming/NVectorCUDA`.

```
data = (realtype *)
  malloc( local_length *
          sizeof(realtype));
```

Original

```
cublasStatus stat =
  cublasAlloc( local_length,
               sizeof(realtype),
               (void**)&data);
```

CUBLAS

Listing 5.3: Replacing memory allocation in NVector library with a CUDA call.

[4]The axpy operation is a BLAS operation for finding a linear combination of two vectors, $\boldsymbol{y} \leftarrow a\boldsymbol{x} + \boldsymbol{y}$, where a is a scalar.

[5]CUBLAS is an implementation of BLAS on the GPU provided by NVIDIA (2011b).

```
N  = NV_LOCLENGTH_P(x);
xd = NV_DATA_P(x);
zd = NV_DATA_P(z);

for (i = 0; i < N; i++)
  zd[i] = ONE/xd[i];
```

(a)

```
N  = NV_LOCLENGTH_P(x);
xd = NV_DATA_P(x);
zd = NV_DATA_P(z);

cuda_inv(N, zd, xd);
```

(b)

```
extern "C" void cuda_inv(int N, double *lhs, double *rhs){
  int blocksPerGrid = (N+threadsPerBlock-1) / threadsPerBlock;
  blocksPerGrid = blocksPerGrid > maxBlocksPerGrid ?
                  maxBlocksPerGrid : blocksPerGrid;
  cu_inv<<<blocksPerGrid, threadsPerBlock>>>
    (lhs, rhs, N);
}

__global__ void cu_inv(double* lhs, double *rhs, int N)
{
  int i = blockDim.x * blockIdx.x + threadIdx.x;
  while (i < N){
    lhs[i] = 1./rhs[i];
    i += blockDim.x * gridDim.x;
  }
}
```

(c)

Listing 5.4: (a) A for loop in NVector for determining the inverse operation $z_i \leftarrow 1/x_i$, $i = 1, 2, \ldots, N$. (b) Replacing the *for*-loop with a call to a CUDA kernel. (c) The CUDA implementation of the inverse operation.

5.6 The CV-FE Discretisation

The algorithms, data structures and steps taken in the residual evaluation were introduced in §4.2. In this section we discuss the implementation of the residual evaluation that supports both CPU and GPU implementation by virtue of using vectorlib to store and operate upon vector data.

5.6.1 Interface

All of the operations and data structures for the CV-FE discretisation are implemented in a Physics class. The Physics class is derived from a generic base class, and implements a generic residual callback to interface with IDA.

Listing 5.5 shows the definition of the Physics class for Richards' equation.

The first template parameter in the definition of the Physics class is value_type, which is a structure of the primary variables at each node in the mesh. In the MPR formulation value_type is the type hM defined at the top of Listing 5.6. The type is stored in a **struct** with static members that specify the number of algebraic and differential variables, which specifies whether the member function residual_evaluation returns the residual for the PR or for the MPR formulation, and is used by the preconditioner to determine if the optimal Schur complement method from §3.3.2 can be used.

The Physics class stores persistent information used to evaluate the residual. This includes the data structures, such as index vectors, sparse matrices and working vectors required to perform the residual evaluation, which are generated at startup. State information such as upwind points and boundary condition information is computed in the time-step preprocessing step (see §4.2.1) of the first callback of the residual function at each time step, and then used for subsequent calls.

The Physics class in Listing 5.5 is templated on two coordinators: a *host coordinator*, and a *device coordinator*. The host coordinator is used for data that resides on the host, such as that used for the initialisation phase and MPI communication[6]. The device coordinator can be thought of as the *computation coordinator*, because it specifies the coordinator for the data-parallel operations in the residual evaluation.

Listing 5.6 shows how to use either the CPU or GPU for the Physics class by means of setting the device coordinator. With this approach, the physics class does not contain any hardware-specific code, because the vectorlib coordinators determine where memory is allocated, and dispatch appropriate hardware-optimised kernels for each operation. This allows either the CPU or GPU implementation to be chosen at compile time by changing the tem-

[6]The host template parameter is explicitly stated to allow the user to specify their own specialised host coordinator, for example, different coordinators that allow the user to choose between OpenMP or pthreads for multi-core processing.

plate parameter for the device coordinator, without making any changes to the Physics code.

```
template <typename value_type, typename CoordHost, typename CoordDevice>
class VarSatPhysics :
  // derive from base class
  public fvm::PhysicsBase< VarSatPhysics<value_type, CoordHost, CoordDevice>,
                           value_type, CoordDevice>,
{
public:
  // expose types that describe data storage
  typedef typename lin::rebind<CoordHost, double>::type CoordHostDouble;
  typedef typename lin::rebind<CoordHost, int>::type CoordHostInt;
  typedef typename lin::rebind<CoordDevice, double>::type CoordDeviceDouble;
  typedef typename lin::rebind<CoordDevice, int>::type CoordDeviceInt;

  // host and compute vectors
  typedef lin::Vector<double, CoordHostDouble> TVec;
  typedef lin::Vector<int, CoordHostInt> TIndexVec;
  typedef lin::Vector<double, CoordDeviceDouble> TVecDevice;
  typedef lin::Vector<int, CoordDeviceInt> TIndexVecDevice;

  // interface for Integrator
  void postprocess_timestep( double t,
                             const mesh::Mesh& m,
                             const TVecDevice &sol,
                             const TVecDevice &deriv);
  void residual_evaluation(  double t,
                             const mesh::Mesh& m,
                             const TVecDevice &sol,
                             const TVecDevice &deriv,
                             TVecDevice &res);
  // ...
}
```

Listing 5.5: Definition of Physics class for the Richards' equation model. The class, VarSatPhysics, is derived from an physics base class. The first template parameter is the variable type, which for the PR model is a scalar for pressure head, and is a structure of pressure head and fluid mass for the MPR model (see Listing 5.6). The second and third template parameters are the host and device coordinators, which are used to define the vectorlib vectors used in the class.

```
// define variable structure for the mixed Richards' model
struct hM {
  double h; // pressure head
  double M; // fluid mass
  // the following information is used by the preconditioner
  // to determine matrix structure
  static const int variables = 2;
  static const int differential_variables = 1;
};

// define CPU and GPU coordinators
typedef lin::DefaultCoordinator<int> CPUCoord;
typedef lin::gpu::Coordinator<int> GPUCoord;

// define physics that run on GPU
typedef VarSatPhysics<hM, CPUCoord, GPUCoord> PhysicsGPU;
// define physics that run on CPU
typedef VarSatPhysics<hM, CPUCoord, CPUCoord> PhysicsCPU;
```

Listing 5.6: Definition of the block variable for the MPR model, and physics for CPU and GPU implementation.

5.6.2 Edge-Based Weighting

Edge-based weighting is used to determine the value of mobility terms and advection terms at control volume faces. It is performed by the use of edge weight vectors, that specify a weighting of variables at the front and back nodes of each edge, which we recall from (3.46)

$$[\lambda]_j = w_k^{\text{front}} [\lambda]_j^{\text{front}} + w_k^{\text{back}} [\lambda]_j^{\text{back}} . \tag{5.1}$$

Two vectors, edge_weights_front and edge_weights_back, are used to store the edge weights, and the indices for the front and back nodes for each edge are stored in two index vectors, edge_nodes_front and edge_nodes_back respectively.

The type of spatial weighting (upstream, central averaging or flux limiting) dictates how the weight vectors for each edge are determined. The simplest case is for central weighting in equation (3.54), that takes the average of the front and back nodes by setting the front and back weights for each edge to 0.5:

```
edge_weights_front(all) = 0.5;
edge_weights_back(all) = 0.5;
```

Because the edge weights are constant for central averaging, the weights vector is fixed throughout the simulation.

The upstream and flux limiting methods first compute the flow direction indicator for each edge during time step preprocessing. FDI(φ) is computed using Algorithm 4.2, which is performed using a single vector subtraction and indirect indexing in vectorlib as follows

```
potential(all) = head + z_nodes;
FDIvec(all) = potential(edge_back) - potential(edge_front);
```

In the first step, the potential $\varphi = \psi + z$ is computed for every node using vector addition, then the FDI for each edge is determined using indirect indexing and vector subtraction.

The vector FDIs, FDI($\boldsymbol{q}$) and FDI($\boldsymbol{\nabla}\phi$), are found using sparse matrix-vector product described in equations (4.5) and (4.6).

The edge weights for upstream weighting are set according to the sign of the FDI on each edge using equation (4.7). For each edge, the upwind node is selected, its weight set to 1, then the weight at the downstream node is set to 0. Operations such as this are specific to the finite volume implementation, and as such are not implemented in vectorlib. Instead, such operations are implemented in an FVMPor routine that dispatches to OpenMP and CUDA kernels according to the device coordinator.

Algorithm 4.3 is used to find the upstream and downstream nodes for each edge, and the neighbour of maximum potential for each node that is used to determine the flux limiter weights. To implement Algorithm 4.3 on the GPU, care has to be taken to avoid a race condition that occurs if two threads simultaneously attempt to update the neighbour of maximum potential for the same node[7]. However, because Algorithm 4.3 is only performed once per

[7]A race condition occurs when two threads attempt to write to the same location on memory at the same time, in which case the result is not well-defined.

time step, in the preprocessing step, it was implemented in serial on the host. This requires that data is copied from the device to the host for processing, then the weights copied back to the device. To ameliorate the overhead of copying data between the device and host, page-locked memory buffers in host memory were used[8].

The edge-based weighting operations discussed above are all performed in the time step preprocessing in §4.2.1. We now consider the operations that are performed every time the residual is evaluated, namely finding edge weights for flux limiting and using the edge weights to approximate the value of mobility and advection terms at control volume faces.

Flux limiting requires inter-process communication to determine 2up information for *interface edges*[9]. This is because, to determine 2up information on interface edges, the neighbour of maximum potential must be known for halo nodes, which requires information about each of a halo node's neighbours, some of which are not stored only on a neighbouring process. Hence, when the residual is evaluated, the upstream information from the neighbour of maximum potential is first determined for each local node, then communicated to neighbouring sub-domains using a halo update. This updates the information pertaining to the neighbour of maximum potential at halo nodes and 2up information for interface edges. Then the weights can the be determined efficiently in parallel.

Once the edge weights have been set, they are used to compute the value of advected quantities $[\rho]_j$ and $[c]_j$ using Algorithm 4.4, which was implemented using OpenMP and CUDA in FVMPor. The same edge weights are used to find the relative permeability $[k_{\mathrm{rw}}]_j$, however a different approach is used in §5.6.4 due to the dependence of permeability on material properties.

[8] Page locked host memory is guaranteed not to be swapped into virtual memory, so that copies between host and device can be performed without buffering, which significantly reduces the copying time (NVIDIA, 2011a). A specialised vectorlib coordinator that uses an allocator for page-locked memory was used for this purpose.

[9] An interface edge has both a local node and a halo node as end points.

5.6.3 Interpolation

Interpolation from nodal values to control volume faces is performed using a sparse matrix vector multiplication in equation (4.10). For example, to interpolate the concentration values to control volume faces the following matrix vector product is used

$$\boldsymbol{\psi}_f = S\boldsymbol{\psi}, \tag{5.2}$$

where $\boldsymbol{\psi}_f$ is a vector of pressure head values at control volume faces. This operation is performed using the following code:

```
// create vector for storing the pressure head at faces
TVecDevice head_faces(mesh.cv_faces());
// use SPMV to to interpolate to faces by multiplying the interpolation
// matrix by the vector of nodal head values
head_faces(all) = S * head;
```

5.6.4 Fluid Properties

Volume averages of fluid properties that are independent of material properties, such as density, are computed directly using the nodal pressure head and concentration values according to (3.23), and face values for these variables are found using using shape function interpolation or edge-based weighting of the nodal values. Volume averages and face values of fluid properties that depend on material properties are more complicated and computationally expensive to form. In §4.2.4 it was proposed that the number of fluid property computations could be minimised by grouping nodes according to material properties. We now describe how this is implemented.

The first step, performed at startup, is to generate the required index and weight vectors. Nodes with the same material property are grouped into sets, called *zones*, where a node is deemed to have a material property if any of its the sub-control volumes has the property. Next, the weight w_k in equation (4.13) is found for the nodes in each zone, along with indices of the

control volume faces that have the material property. The index and weight information is used in Listing 5.7 to compute the volume average moisture content θ_i, and the face value of relative permeability $[k_{\mathrm{rw}}]_j$.

```
// zero the vector containing CV average of moisture content theta
theta.zero();

// loop over material properties/zones
for(k=0; k<num_zones; k++){
    // load head values of CVs in zone k into vector h_zone
    head_zone = head(p_zone[k]);

    // call kernel that computes psk values (saturation and krw) using
    // parameters for zone k
    psk(Sw_zone, krw_zone, head_zone, parameters[k]);
    porosity(phi_zone, head_zone, parameters[k], constants());

    // find moisture content as product of saturation and porosity
    theta_zone = mul(Sw_zone, phi_zone);

    // add contribution to CV average of moisture content
    theta(p_zone[k]) += mul(theta_zone, weight_zone[k]);

    // add contribution to permeability at faces
    // performed in two steps to avoid aliasing on LHS
    krw_faces(q_front[k]) =
            mul( krw_zone(n_front[k])CPU
                 edge_weights_front(p_front[k]) );
    krw_faces(q_back[k]) +=
            mul( krw_zone(n_back[k]),
                 edge_weights_back(p_back[k]) );
}

// density is not dependent on material properties, and can be
// determined outside the zone loop
density = density(h, constants())
```

Listing 5.7: Code for computing fluid properties for Richards' equation. The method loops over each zone, evaluating the moisture content and relative permeability theta_zone and krw_zone for each node that lies in the zone, then adding each value's contribution to the relevant volume average and face respectively.

The outer loop is over the material zones, so that the volume averages and control faces values in a zone are computed together. A vector of pressure head at nodes in zone k is gathered using the index vector p_zone[k], then the functions psk and porosity[10] compute saturation S_{w}, permeability k_{rw}, and

[10] psk and porosity dispatch to optimised OpenMP or GPU kernels according to the compute coordinator of Physics.

porosity ϕ using the pressure head values and material properties of zone k. The moisture content is then found as the product of saturation and porosity.

The contribution of the moisture content from each node in zone k is added to the volume averages according to (4.13), using the index vector p_zone and the weights vector for the w_k. Then, the contribution of the relative permeabilities is added to the control volume faces in zone k. This is performed in two steps, first adding contributions from front nodes of each face, followed by the contribution from back nodes, to avoid a race condition when a face value is simultaneously updated with front and back values.

The mapping of relative permeability values onto control volume faces uses indirect indexing of nodal values, faces and edge weights. The computational efficiency of this operation is sensitive to the order of indices in the index vectors. Further investigation of how this operation was optimised by renumbering of nodes, edges and faces is discussed in detail in §5.7.

5.6.5 Flux Assembly

Once the values at faces have been reconstructed using shape function interpolation and edge weighting, we can form the net fluid mass flux over each face

$$[q^{\mathrm{w}}]_j = -A_j\,[\rho]_j\,[k_{\mathrm{rw}}]_j\,\mathsf{K}_j\left([\boldsymbol{\nabla}\psi]_j + \frac{1}{\rho_0}\,[\rho]_j\,\boldsymbol{\nabla} z\right)\cdot \boldsymbol{n}_j, \tag{5.3}$$

which is a scalar value. The face values[11] used to compute $[q^{\mathrm{w}}]_j$ in (5.3) are stored in vectors of length n_f, with vector-valued quantities, such as pressure head gradient $[\boldsymbol{\nabla}\psi]_j$, stored with one vector for each component. The vector of net flux values, qw_faces, is computed using the sequence of vectorlib operations in Listing 5.8.

[11]The face values are: face area A_j; density $[\rho]_j$, which is found using edge weights in the advection term, and using shape functions in the buoyancy term; relative permeability $[k_{\mathrm{rw}}]_j$; hydraulic conductivity K_j which is a diagonal tensor; the gradient of pressure head $[\boldsymbol{\nabla}\psi]_j$; and the control volume face normal $\boldsymbol{n}_j$.

```
// x component of grad potential
pot_faces.x = mul(K_faces.x, grad_h.x);
pot_faces.y = mul(K_faces.y, grad_h.y);

// y component of grad potential: include buoyancy term
pot_faces.z()(all) = grad_h_faces.z;
pot_faces.z() += (1./rho_0)*rho_faces_lim;
pot_faces.z() *= K_faces.z;

// take dot product with face normals
q_faces  = mul(pot_faces.x, face_norms.x);
q_faces += mul(pot_faces.y, face_norms.y);
q_faces += mul(pot_faces.z, face_norms.z);

// scale saturated flux by relative permeability to get volumetric flux
q_faces *= krw_faces_lim;
// mass flux is product of volumetric flux and fluid density
qw_faces = mul(rho_faces_lim, q_faces);
// scale mass flux by face area
qw_faces *= face_areas;
```

Listing 5.8: Flux assembly for the fluid mass $[q^{\mathrm{w}}]_j$ in three dimensions.

We note that the sequence of operations in Listing 5.8 could be performed more efficiently if all or some of the operations were fused, which would reduce the number of global memory accesses. The approach taken here is to first write the application in terms of basic operations, and implement fused operations only in parts of code that are identified as bottlenecks during profiling and tuning.

If the volumetric flux over the boundary is prescribed by the boundary condition, as per (2.21), the flux at each face is computed on the host[12], then copied to the device during the time step preprocessing. They are then substituted into the q_faces each time the residual is evaluated in Listing 5.8.

[12]The specified flux is a function of space and time, and may have an arbitrary functional form or be interpolated from an input file, so it is easier to compute on the host. The overhead of performing the copy is negligible, because the boundary faces are relatively few compared to internal faces, and the copy is performed only once each time step.

Computing the flux at the Dirichlet faces using the matrix-vector multiplication in (4.20) is performed as follows

```
// create temporary vector to store fluxes
TVecDevice flux_tmp(dirichlet_faces.size());
// use SPMV to gather fluxes at Dirichlet faces into the temporary vector
flux_tmp = dirichlet_matrix*q_faces;
// insert fluxes into global face flux vector
q_faces(dirichlet_faces) = flux_tmp;
```

The flux at Dirichlet faces are first gathered in the temporary vector using a matrix-vector product, then substituted back into the global flux vector using the index vector `dirichlet_faces` .

5.6.6 Residual Assembly

Once the fluxes over each face and the control volume averages are computed, the residual is assembled. Here we consider residual assembly for the PR and MPR formulations, which have one and two equations per node respectively.

The residual function for the PR formulation is a differential equation, defined in (3.70) as

$$f_i(t, \boldsymbol{\psi}, \psi') = n_i \psi_i' + Q_i^{\mathrm{w}}(\boldsymbol{\psi}) - \rho_i S_i^{\mathrm{w}}. \tag{5.4}$$

The accumulation term, $n_i \psi_i$, and the net flux for each control volume, $Q_i^w(\boldsymbol{\psi})$, are computed in two steps in Listing 5.9, where `flux_matrix` is the sparse matrix defined in (4.23), `qw_faces` is the vector of face values of $[q^{\mathrm{w}}]_j$ computed in Listing 5.8, `h_p` is a vector with the derivatives of the pressure head ψ_i', and `storage` is a vector of the volume average of the storage term n_i.

```
res  = flux_matrix * qw_faces; // collect fluxes for each control volume
res += mul(storage, h_p);      // add accumulation term
```

Listing 5.9: Source code for residual assembly of the PR formulation of Richards' equation.

The contribution of the source term, $\rho_i S_i^{\mathrm{w}}$, is added one-by-one for each source term in the domain

```
for(int i=0; i<source_S.size(); i++){
    // Note: source_S, sourec_idx and source_rho are std::vector
    //       containers on the host,
    //       while res and density are minlin device vectors
    double S = source_S[i];
    int idx = source_idx[i];
    if(S<0.)
        res(idx) += density(idx)*S;
    else
        res(idx) += source_rho[i]*S;
}
```

The above code manipulates individual values stored in the device vectors `res` and `density` from the host, which requires copying scalar values between the host and device. The overhead for this operation is negligible, so long as there are only a few source terms. In this situation, the simplicity of using host code outweighs the computational benefits of implementing the source term update entirely on the GPU.

The MPR formulation of Richards' equation has two residual functions per node, defined in (3.74) and (3.75). The first is the differential equation

$$f_i = M_i' + Q_i^{\mathrm{w}}(\boldsymbol{\psi}) - \rho_i S_i^{\mathrm{w}}, \tag{5.5}$$

and the second is the algebraic expression

$$g_i = M_i - \rho_i \theta_i. \tag{5.6}$$

Assembling the residual is very similar to the PR formulation, with an extra step to compute the algebraic expression g_i in (5.6), and with the fluid mass

M_i as primary variable

```
int NL = mesh.local_nodes();

// store f values in first half of residual vector
res(0,NL-1)  = flux_matrix*qw_faces; // collect fluxes
res(0,NL-1) += M_p;                  // add accumulation term

// store g values in second half of residual vector
res(NL,end)  = M;
res(NL,end) -= mul(theta, density);
```

The residual function for the mass balance, (5.4) and (5.5) in the PR and MPR formulations respectively, is replaced with an algebraic expression (3.73) for nodes on Dirichlet boundaries

$$\psi_i - \psi_b(t, \boldsymbol{x}), \tag{5.7}$$

where $\psi_b(t, \boldsymbol{x})$ is the pressure head prescribed by the boundary condition. This is performed using indirect referencing to restrict the operation to the Dirichlet nodes

```
res(dirichlet_nodes)  = head(dirichlet_nodes) - head_b;
```

The index vector `dirichlet_nodes` lists the nodes on Dirichlet boundaries, and `head_b` has the corresponding head values prescribed by the Dirichlet condition, which are set during the time step preprocessing.

5.7 A Mesh Renumbering To Optimise Indirect Indexing On GPUs

Indirect indexing is used extensively throughout the residual evaluation via flat index vectors and sparse matrix-vector multiplication. The computational performance of these operations is very sensitive to the order of the indices, for reasons discussed in §5.1.1. Here a novel method for renum-

bering the *components*[13] in the dual mesh is presented, which improves the performance of operations that use indirect indexing.

The indirect indexing operations that are computational bottlenecks in FVM-Por can be categorised as either *gather* or *scatter* operations.

- Examples of gather operations include: gathering values at the vertices of an element to compute a face value using shape functions as described in (4.9); gathering the flux over control volume faces to form the net control volume flux in (4.22); gathering fluxes at the faces on an edge to form a vector FDI (3.48).
- An example of a scatter operation is adding the contribution of a nodal k_{rw} value to each face of the node's control volume in Listing 5.10.

In each of these gather/scatter operations, values are mapped from/to mesh components that are directly adjacent in the mesh. This reflects the local nature of the finite volume operator, whereby the face and edge values that contribute to the residual function at each node are adjacent (local) to the node in the mesh.

Cache reuse is optimised when threads in a warp access memory locations that are close in memory, preferably in the same cache line. For this to occur, the indices into the node, face and edge arrays used for the spatially-local gather and scatter operations must refer to entries that are close together in memory. This suggests numbering the nodes, edges and faces such that consecutively-numbered components are close together spatially.

To ensure that nodes that are near-by in space are also near-by in memory, we analyse the matrix of the adjacency graph of the nodes in the finite element mesh. The upper bandwidth of the matrix represents the maximum distance between any two adjacent nodes in memory. This distance is reduced

[13] We refer to nodes, edges and faces collectively as components of the dual mesh.

by renumbering nodes according to the bandwidth-reducing reverse Cuthill-McKee (RCM) permutation (Cuthill and McKee, 1969).

Following the renumbering of nodes, the edges and faces are numbered according to their nearest-neighbour node in the mesh. This is done by first sorting the edges according to the minimum index of the nodes at their end points. Then, when the dual mesh is constructed, control volume faces are formed by traversing the edges in order, and numbering the faces attached to each edge in ascending order.

With this renumbering, not only are the faces and edges associated with a single node close together in memory – edges and faces associated with a set of nodes with contiguous indices are also clustered locally in memory. This approach has the benefit of being straight-forward to implement when the mesh is loaded, without requiring any geometric or spatial analysis (Corrigan et al., 2010), which makes it well-suited to irregularly-shaped domains.

5.7.1 Indirect Indexing in Computing Relative Permeability

To illustrate the renumbering, we now look at a specific indirect indexing operation in the residual evaluation: finding the relative permeability at control volume faces. For simplicity, it is assumed that the porous medium is homogeneous in Listing 5.10 [14]. This operation is the most computationally expensive indirect indexing operation in the residual evaluation, because it uses indirect indices into node, edge and face vectors.

The mapping of front values in Listing 5.10 uses indexing as follows: assign to each face (indexed with `q_front`) the permeability at the face's front node (indexed with `n_front`) multiplied by the edge's front weight (indexed with

[14] We note that the homogeneous case in Listing 5.10 is simpler than the heterogeneous case in Listing 5.7. Indeed, it is equivalent to using indirect indexing on flat vectors to perform Algorithm 4.4.

```
// compute krw for each node from nodal head values and parameters
psk(krw, head, parameters());

// add contribution to permeability at faces
// performed in two steps to avoid aliasing on LHS
krw_faces(q_front) =  mul( krw(n_front),            // front mapping
                           edge_weights_front(p_front) );
krw_faces(q_back) +=  mul( krw(n_back),             // back mapping
                           edge_weights_back(p_back) );
```

Listing 5.10: Code for computing relative permeability at control volume faces in a homogeneous medium.

p_front). Figure 5.5 shows both the physical location and memory indices of nodes, edges and faces accessed by a half warp on an unstructured two-dimensional mesh for the front mapping [15]. Each node is referenced multiple times in index n_front, once for each face onto which it is mapped. By sorting it in ascending order, at most two cache lines are read from krw by each half warp [16]. The indices of the faces onto which the nodes are mapped also show a high degree of clustering. Furthermore, The spatial clustering of the nodes, edges and faces referenced by the indices is apparent in Figure 5.5(b).

5.8 Implementation of Preconditioners

The block-Jacobi preconditioner for distributed matrices was introduced in §4.4, along with a method for applying block preconditioners efficiently on the GPU. Here we give more detail about the different techniques used to factorise and apply the preconditioners on the local block of each sub-domain. First, the steps involved in forming the preconditioner, along with three different sparse factorisations are presented in §5.8.1. Then, in §5.8.2, the different methods used to apply the preconditioner are outlined.

[15] A two-dimensional triangle mesh was loaded and sorted, then the nodes, edges and faces referenced by a random warp were recorded.

[16] Most often only one cache line will have to be read, however there will be times when the clustered node indices fall either side of the boundary between cache lines.

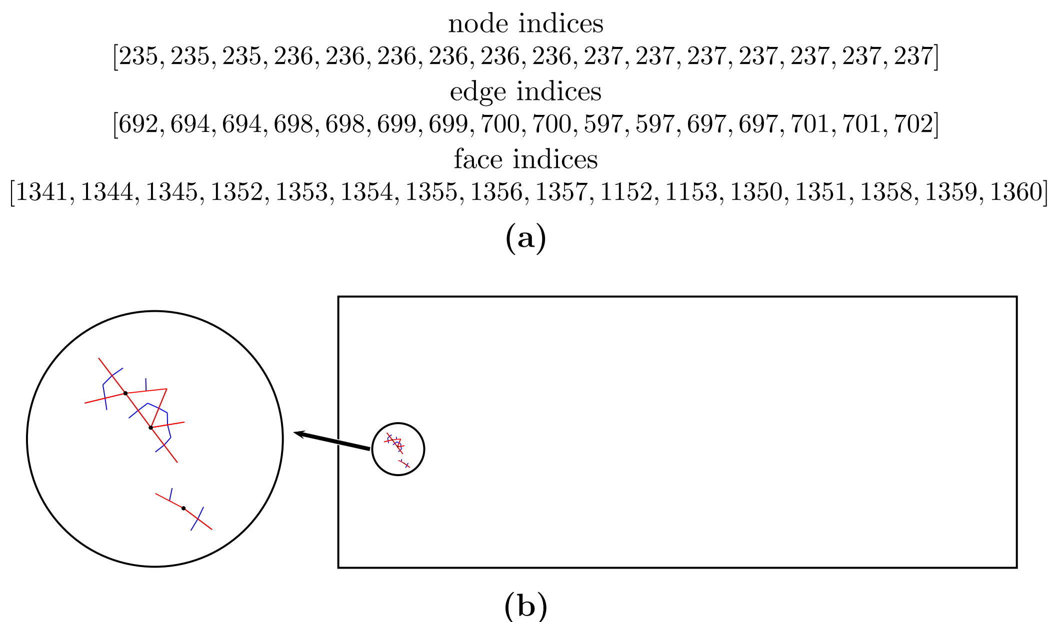


Figure 5.5: (a) The indices of nodes, edges and faces accessed by a half warp when mapping the front faces in Listing 5.10. (b) The physical location of the nodes, edges (red) and faces (blue). There are 16 faces (one for each thread in the half warp), nine edges and three nodes accessed.

5.8.1 Forming the Preconditioner

An analysis step is performed the first time that a preconditioner is formed. The analysis step first determines a multi-colouring of the columns using Algorithm 4.5 so that the columns of the iteration matrix can be computed efficiently. Additional analysis may also need to be performed at this point, depending on the specific preconditioner being used.

Each time the preconditioner is formed, the local block of the iteration matrix is approximated efficiently using the multi-colouring introduced in §4.4.2. The factorisation is always performed on the host, using sparse factorisations from the Intel MKL library (Intel, 2010). Hence, if the GPU is being used, the matrix is formed on the device, then copied to the host for factorisation.

Once the local block of the iteration matrix G is stored on the host, it is

factorised using one of the following methods:

- *LU*: A full sparse LU factorisation is performed using the PARDISO[17] library (Schenk and Gärtner, 2004), which is included in the Intel MKL. Both the factorisation and application phases are performed on the host, and are optimised to use OpenMP. This improves the preconditioner's performance using both MPI-OpenMP and MPI-OpenMP-CUDA.
- *ILUT*: An incomplete factorisation with thresholding (ILUT) preconditioner (Saad, 1994). The factorisation and triangle solution routines in MKL are both single threaded.
- *ILU(0)*: An incomplete factorisation without fill in from the Intel MKL library. The factorisation and application routines in the MKL are single threaded, however the fixed sparsity pattern makes it possible to use the multi-colouring that will be discussed in the following section to parallelise the application phase on the GPU.

If the application phase is to be conducted on the device, the factorised matrix is copied to the device.

5.8.2 Applying the Preconditioner

There are six different methods used to apply the preconditioner computed in the previous section: four of which apply the preconditioner on the host; and two of which apply the preconditioner on the device.

Applying the preconditioner on the host requires communication between host and device memory every time the preconditioner is applied. Analysis in Chapter 7 shows that the time spent copying is a significant proportion

[17] *http://www.pardiso-project.org*

of all time spent in the preconditioner, so to ameliorate the cost of copying, the vectorlib host vectors used for the copying use page-locked host memory.

The remainder of this section will discuss each of the methods for applying the preconditioners in turn.

- **LU (host):** The LU preconditioner applies the full sparse LU decomposition to the local block on the host. The full factorisation is very effective[18], however the computational overhead associated with applying this preconditioner are considerably higher than the other preconditioners.
- **ILU(0) (host):** This preconditioner applies the ILU(0) preconditioner on the host.
- **ILUT (host):** This preconditioner applies the ILUT preconditioner on the host.
- **B-ILU(0) (host):** This preconditioner applies a block Jacobi preconditioner to the local block, where the number of blocks is equal to the number of OpenMP threads on each process, and the ILU(0) preconditioner is used on each block. OpenMP is used to factorise and apply the preconditioner for each block on separate host CPU cores[19].
- **MC-ILU(0) (device):**

  The MC-ILU(0) preconditioner applies the ILU(0) preconditioner on the device using Algorithm 4.6. The factorised matrix is stored in the permuted block form (4.29) on the device, with a data structure that stores the L_i and U_i blocks as sparse matrices in CSR format, and the diagonal blocks D_i as vectors.

[18]The full LU factorisation is exact when used the first time on the matrix for which it is calculated, however because it is reused over multiple time steps during which the matrix G changes, it is no longer exact.

[19]To ensure optimal performance, thread affinity is fixed, so that the memory allocation, factorisation and application are performed on the same core for each block.

- **CUSP-ILU(0) (device):** The CUSP-ILU(0) preconditioner uses the sparse triangle solve implemented in the CUSPARSE library (Naumov, 2011) to apply an ILU(0) preconditioner on the device. CUSPARSE uses the same multi-colouring method to determine independent sets of rows for processing as the MC-ILU(0) preconditioner. However, where the MC-ILU(0) method stores the matrix in permuted form of (4.29), the CUSPARSE library does not permute the matrix. Instead, the permutation matrix q and idx are used to process the rows of the matrix in place.

5.9 Conclusions

With the emergence of new many-core processor architectures, such as GPUs, designed for fine-grained parallelism, one of the key challenges of in scientific computing is writing software that will perform well across a range of different computational units. Another related issue is ensuring that software will be able to support future hardware platforms with as little rewriting as possible. The method chosen to meet these aims in this thesis is to implement the low-level hardware implementation of memory management and data-parallel operations in the vectorlib library. The library supports multiple hardware platforms, and provides a flexible, hardware-agnostic syntax. This allows the FVMPor code written with vectorlib to be compiled to run for different hardware through changing a template parameter.

A significant challenge to implementing unstructured mesh codes on the GPU is the indirect indexing operations they necessitate, which can lead to inefficient global memory access patterns. This was addressed in §5.7, where a renumbering of nodes, edges and faces in the dual mesh was proposed to increase cache reuse on the Fermi architecture when performing gather and scatter operations in the residual evaluation.

In §5.5 the NVector library used by IDA was rewritten to store and process

local vectors used in the Newton and GMRES iterations on the GPU, and use MPI for communication between sub-domains. Finally, six methods for applying sparse factorisations as preconditioners were described in §5.8.2. Of these, four were applied on the host. The remaining two methods apply the preconditioner on the device using the multi-colouring method described in §4.4.3.

In this chapter each of the significant computational overheads of the Newton-Krylov method used by the implicit time stepping scheme has been implemented on the GPU, namely the residual evaluation, vector operations in NVector and preconditioner application. In Chapter 7 the computational efficiency of the CPU and GPU implementations of FVMPor presented in this chapter will be presented. The next chapter will verify the CV-FE discretisation and the different formulations on several challenging test problems.

Chapter 6

Model Verification

In this chapter the methods introduced in this thesis are verified and analysed by applying them to several test problems. The first section introduces test problems for both Richards' equation and the coupled transport model. The following two sections present results from these test problems, focussing on the accuracy and computational efficiency of the different spatial schemes and formulations. The final section of the chapter investigates the effect of options relating to the time stepping scheme, namely the order of integration and the choice of preconditioner, on the efficiency and accuracy of the scheme.

Throughout this chapter, results are computed using both the CPU and GPU implementation. In each case, it is noted which implementation was used to obtain then results. More detailed analysis of the computationaly performance of the GPU and CPU codes will follow in Chapter 7.

6.1 Test Cases

Each test case is chosen to analyse specific aspects of the numerical scheme. All of the test cases, except for the test cases presented in §6.1.3 and §**??**, have

been investigated previously in the literature, so that the solutions obtained here may be verified against numerical and experimental results.

There are two test problems for Richards' equation. The first was proposed by Forsyth et al. (1995) to test the efficacy of numerical methods for accurately modelling sharp infiltrating fronts in very dry soil, and has subsequently been investigated by Diersch and Perrochet (1999), Therrien et al. (2010) and Carr et al. (2011). The exact mass balance is also available in this test case, which makes it possible to verify conservation of mass for the different formulations.

The second test problem for Richards' equation is based on laboratory experiments performed by Vauclin et al. (1979), who investigated the transient location of the water table. The experimental results were reproduced numerically by Fahs et al. (2009), so that results can be validated against both experimental and numerical results.

Another three test cases are used to verify the full coupled flow and transport model. The first models the injection of a contaminated fluid into an unsaturated heterogeneous box. It was formulated to test both the efficacy of the flux limiters and conservation of mass for simulating transport in unsaturated heterogeneous media.

The second test problem is based on the experimental results of Zhang et al. (Zhang, 2000, Volker et al., 2002, Zhang et al., 2004), who investigated the density-dependent transport of contaminant plumes in unconfined coastal aquifers. The experiments were performed in a laboratory model of an unconfined coastal aquifer with time-varying boundary conditions on the seaward boundary and the evolution of contaminant plumes of different densities was investigated.

The final test case is formulated to verify the formulation for variably-saturated flow and transport on an unstructured mesh three-dimensional mesh. The test case simulates a contaminant plume, that is leached from a

contaminated heap into a shallow aquifer.

6.1.1 Richards' Equation: Infiltration Into Dry Heterogeneous Soil – The *dry_infiltration* Test Case

The *dry_infiltration* test case was proposed by Forsyth et al. (1995, Problem 2) to assess the efficacy and efficiency of numerical methods for simulating infiltrating fronts under very dry initial conditions. The domain has four regions, illustrated in Figure 6.1, each with different material properties that are summarised in Table 6.1. Water infiltrates through the 2.25 metre-wide gap in the top left hand boundary at a constant rate of $q_b = -0.02$m/d. The domain is initially dry, with a uniform pressure head.

In a separate numerical analysis of the problem, both Forsyth et al. (1995) and Diersch and Perrochet (1999) investigated two different uniform values for initial pressure head: the first was $\psi^0 = -7.34$m; and the second had very dry soil at $\psi^0 = -100\,\text{m}$. In both papers, a Galerkin finite element discretisation on a uniform, 1891-node quadrilateral mesh was used.

It is possible to verify the accuracy of the conservation of mass of the numerical solutions to this test case because both the exact initial mass of fluid in the domain, and the fixed mass flux of fluid over the gap at the top left boundary, are known.

Zone	K [m/s]	ϕ [1]	S_r [1]	α_v [1/m]	n_v [1]
1	$9.153 \cdot 10^{-5}$	0.3680	0.2771	3.34	1.982
2	$5.445 \cdot 10^{-5}$	0.3510	0.2806	3.63	1.632
3	$4.805 \cdot 10^{-5}$	0.3250	0.2643	3.45	1.573
4	$4.805 \cdot 10^{-4}$	0.3250	0.2643	3.45	1.573

Table 6.1: Material properties for the *dry_infiltration* test case.

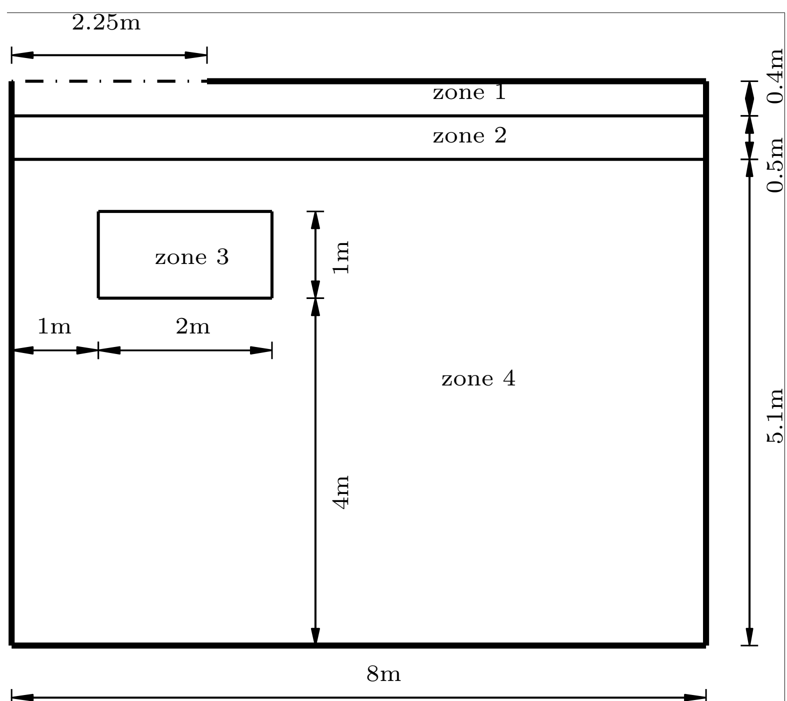


Figure 6.1: The domain for the *dry_infiltration* test case due to Forsyth et al. (1995).

6.1.2 Richards' Equation: Transient Water Table Experiment – The *water_table* Test Case

The *water_table* test case is based on the laboratory experiments investigated by Vauclin et al. (1979) for determining the transient position of a water table. The experiment was performed in a soil box of dimensions 6 m×2 m. Water infiltrated through a 1 m-wide strip at the top of the centre of the domain, and a hydrostatic condition was imposed for the lower 65 cm at the sides of the tank. Due to the symmetric nature of the problem, only the right hand half of the domain is modelled, with a no flow condition imposed along the axis of symmetry, as illustrated in Figure 6.2. The system is assumed to initially be at hydrostatic equilibrium with the water table at a height of 65 cm specified by a hydrostatic Dirichlet condition on the right hand side. The parameters for the homogeneous material properties are listed in Table 6.2.

Previous numerical investigations of this problem (Vauclin et al., 1979, Fahs et al., 2009) treated the problem as two-dimensional. To test the efficacy of our discretisation in three-dimensions, we extend the tank to a width of 0.5 m.

parameter	value
K	$9.72 \cdot 10^{-5}$ m·s^{-1}
ϕ	0.3
S_r	0.033
α_v	3.3 m^{-1}
n_v	4.1
S_s	$1 \cdot 10^{-8}$ m^{-1}

Table 6.2: Material properties for the *water_table* test case.

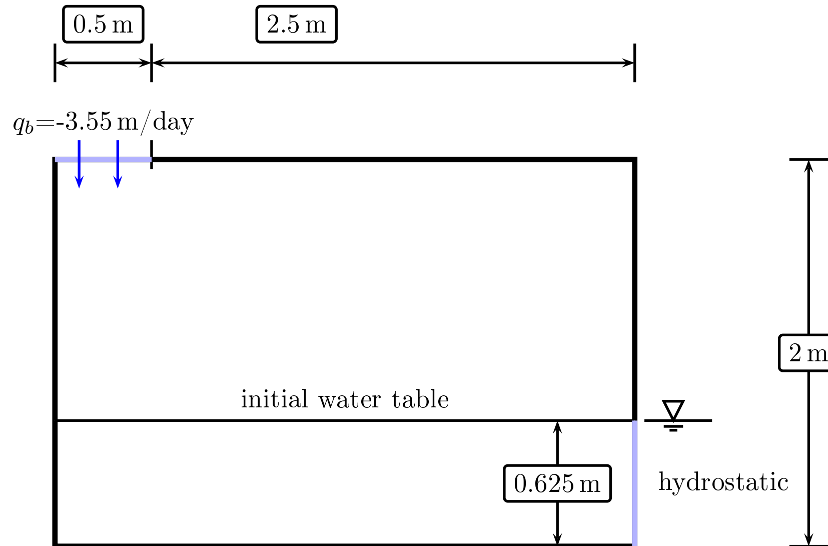


Figure 6.2: The domain for the *water_table* test case due to Vauclin et al. (1979). Only the right hand side of the full problem is illustrated here.

6.1.3 Transport Model: Flow and Transport in Unsaturated Soil – The *unsaturated_transport* Test Case

The *unsaturated_transport* test case is designed to test the upstream weighting and flux limiters, and the conservation of mass, for the coupled transport model. The domain is the two-dimensional box exhibited in Figure 6.3, that is initially unsaturated with a uniform pressure head of $\psi^0 = -2\,\mathrm{m}$ and a salt concentration of zero. All of the boundaries have zero flux, so that the only source of fluid and salt is a point source of fluid with a salt concentration of $c = 1$ at the point ($x = 1\,\mathrm{m}$, $y = 0.5\,\mathrm{m}$) at a rate of $1 \cdot 10^{-6}\ \mathrm{m}^3/\mathrm{s}$. The density of the water is computed with equation (2.11), given a reference fresh water density of $\rho_0 = 1000\,\mathrm{g/L}$, density coupling coefficient $\eta = 0.025$ and by assuming it is an incompressible fluid ($\beta = 0$).

As for the *dry_infiltration* test case, it is possible to measure mass balance

errors because the only source of fluid and salt is the constant source term.

high-permeability zone

parameter	value
K	$5 \cdot 10^{-5}$ m·s^{-1}
ϕ	0.368
S_r	0.3261
α_v	3.55 m^{-1}
n_v	2
α_L	$1 \cdot 10^{-3}$ m
α_T	$1 \cdot 10^{-3}$ m
D_m	0 m^2·s^{-1}
η	0.025 m^{3}kg^{-1}

(a)

low-permeability zone

parameter	value
K	$5 \cdot 10^{-7}$ m·s^{-1}
ϕ	0.2
S_r	0.1261
α_v	3.55 m^{-1}
n_v	1.8
α_L	$1 \cdot 10^{-3}$ m
α_T	$1 \cdot 10^{-3}$ m
D_m	0 m^2·s^{-1}
η	0.025 m^{3}kg^{-1}

(b)

Table 6.3: Parameters for *unsaturated_transport* test case.

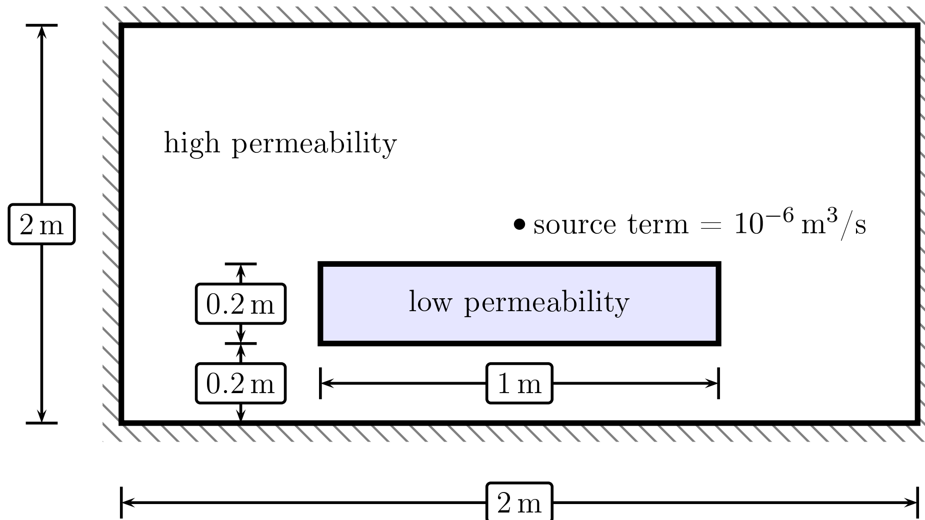


Figure 6.3: The domain for the *unsaturated_transport* test case. The domain has no-flux conditions imposed over each boundary, so that the only source of fluid and solute is at the injection point at $x = 1$, $y = 0.5$.

6.1.4 Transport Model: Flow Tank Experiments – The ***tank_steady***, ***tank_plume*** and ***tank_tidal_plume*** Test Cases of Zhang

These test cases are based on a series of experiments performed in the PhD work of Zhang (2000), some of which was subsequently published (Zhang et al., 2002, 2004). The experiments were performed in a laboratory flow tank filled with a homogeneous porous medium, with a sea water interface on a sloping beach, and constant fresh water head on the inland side of the domain, illustrated in Figure 6.5(a).

The flow tank had dimensions 1.65 m long, 0.6 m high, and 0.1 mm wide, with a homogeneous porous medium composed of uniform glass beads, for which it was possible to accurately determine the material properties listed in Table 6.4(a). A series of experiments were performed using the flow tank by Zhang (2000), three of which are investigated in this work.

tank_steady: This test was performed to determine the steady state location of the sea water interface in the absence of both tidal variation of sea levels and contaminant plumes. On the inland side of the domain a constant fresh water level of 0.463 m was maintained, and the sea level was kept constant at 0.439 m, as illustrated in Figure 6.5(b). The sea water interface was allowed to form, until it obtained a steady state.

tank_plume: The fresh water head and sea level were fixed at the same levels as for the *tank_steady* test case and the steady state sea water interface was allowed to form. Then, a contaminant plume of density 1015.7 g/L (corresponding to concentration $c = 0.689$, where $c = 1$ is the normalised concentration of salt in the sea water), was injected at a constant rate of 0.14 mm/s along the .18 m-long injection boundary above the beach.

tank_tidal_plume: These experiments investigated the evolution of contaminant plumes of different densities with periodic tidal forcing. The sea level

had an average height 0.439 m, and the tide had an amplitude of 40 mm with a period of 40 minutes. Initially, the conditions for this test were identical to those for the *tank_plume* test case, however tidal variations were initiated along with the injection of the contaminant when the steady state solution had formed. Contaminant plumes with four different densities were investigated, with densities that ranged from that of fresh water to the same density as sea water. The density of the plumes was varied by altering the concentration of salt in the injected fluid. The effect of tidal fluctuations was minimal for the lightest plume, while the most dense plume exhibited large unsteady fingering patterns due to the density differences between the plume and the fresh water, which makes modelling it a challenge (Brovelli et al., 2007). In this chapter we investigate the evolution of the second-most dense contaminant plume, which was also investigated by Brovelli et al. (2007), with details in Table 6.4(b).

Zhang (2000) also performed numerical simulations to reproduce the experimental results, for which a two-dimensional model was used, with computational domain in Figure 6.5(b). Limits imposed by computational resources when the original study was published restricted the numerical analysis to a relatively coarse quadrilateral mesh of 3,840 nodes, and did not consider tidal variation, which adds considerably to the computational cost of the simulations.

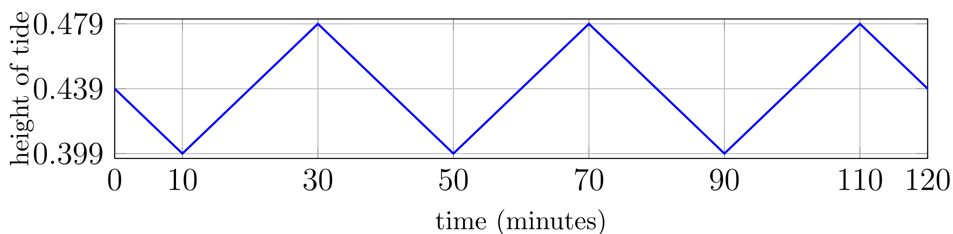


Figure 6.4: Height of the tide for the *tank_tidal_plume* experiment.

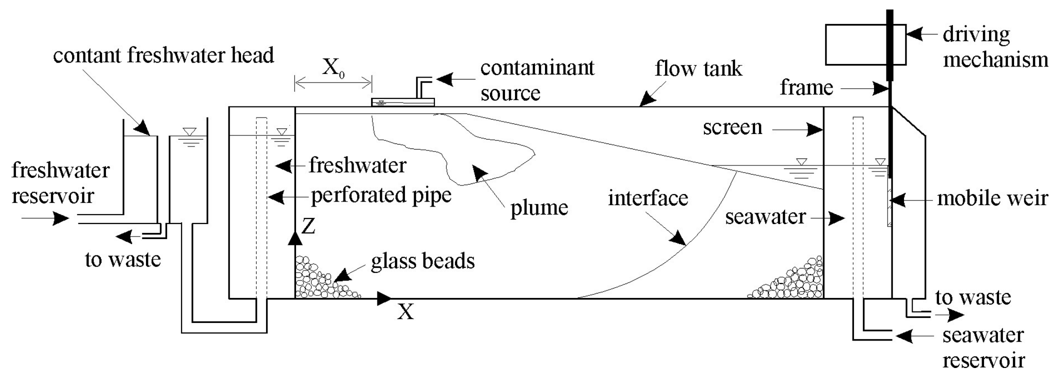
parameter	value
η	0.03
ρ_0	995.1 g/L
α_L	0.65 mm
α_T	0.1 mm
η	$0.025\,\mathrm{m^3kg^{-1}}$
D_m	$0\,\mathrm{mm^2/s}$
K	4 mm/s
ϕ	0.37
α_{vg}	$8.6\,\mathrm{m^{-1}}$
n_{vg}	9.5
S_r	0.3261

(a) Parameters

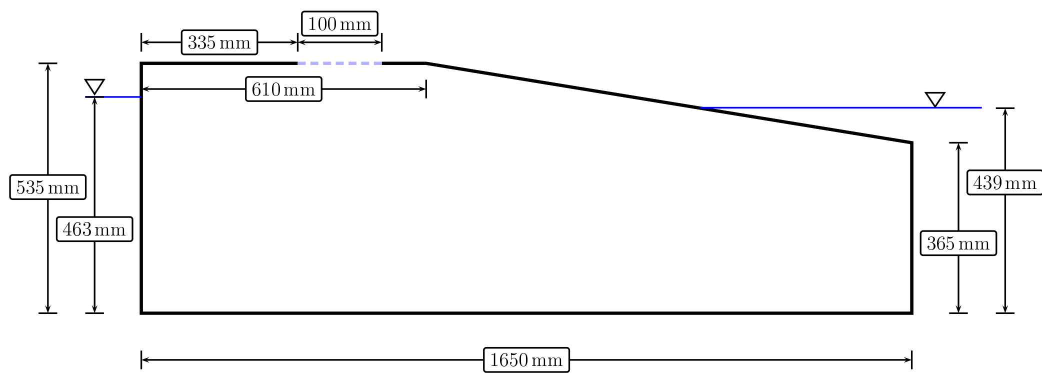
	concentration of solute	density of solute	relative density
fresh water	0	995.1	1
contaminant plume	0.689	1015.7	1.021
sea water	1	1025.0	1.03

(b) Properties of the contaminant plume.

Table 6.4: Experimentally determined parameters for experiments by Zhang (2000). The density of fresh water is 995.1 g/L, and the sea water has salt concentration of $c = 1$.



(a) Experimental flow tank



(b) Computational domain

Figure 6.5: The laboratory flow tank, and corresponding computational domain for the laboratory flow tank experiments by Zhang (2000).

6.1.5 Transport Model: Leaching of a Contaminant Plume in a Shallow Aquifer – The *heap_leaching* Test Case

This test case was formulated to investigate the spatial weighting methods for variably-saturated flow and contaminant transport in three dimensions. The domain, as illustrated in Figure 6.6, is a homogeneous shallow aquifer with a heap of contaminated porous media situated above the aquifer. The material properties of the aquifer and heap are summarised in Table 6.5.

Hydrostatic initial conditions are imposed everywhere in the domain, with the water table 20 cm below the surface. No flow boundary conditions are imposed along the sides and bottom of the domain, except for the left hand boundary ($x = 0$), where a hydrostatic boundary condition consistent with the initial conditions is imposed. A fluid source term is located at $[x, y, z] = [1, 0, -1.5]$, with a constant injection rate of $0.1728\,\mathrm{m}^3/\mathrm{day}$.

The initial concentration of the contaminant in the heap satisfies the condition $C = \phi$, or $c = 1/S_\mathrm{w}$. This concentration is imposed to ensure that the total volumetric average of contaminant is initially constant everywhere in the heap. The surface of the heap is subject to a constant inflow of 5 mm/day, with a no flow boundary condition imposed elsewhere on the top surface. The simulation period is 81 days, during which a contaminant plume is formed in the aquifer due to leaching of the contaminant from the heap.

aquifer

parameter	value
K	$5 \cdot 10^{-5}$ m·s^{-1}
ϕ	0.351
S_r	0.2806
α_v	3.63 m^{-1}
n_v	1.632
α_L	$1 \cdot 10^{-3}$ m
α_T	$1 \cdot 10^{-3}$ m
D_m	$5 \cdot 10^{-8}$ m^2·s^{-1}
η	0 m^{3}kg^{-1}

contaminated heap

parameter	value
K	$2 \cdot 10^{-4}$ m·s^{-1}
ϕ	0.37
S_r	0.3261
α_v	3.5 m^{-1}
n_v	1.5
α_L	$1 \cdot 10^{-3}$ m
α_T	$1 \cdot 10^{-3}$ m
D_m	$5 \cdot 10^{-8}$ m^2·s^{-1}
η	0 m^{3}kg^{-1}

Table 6.5: Parameters for *heap_leaching* test case.

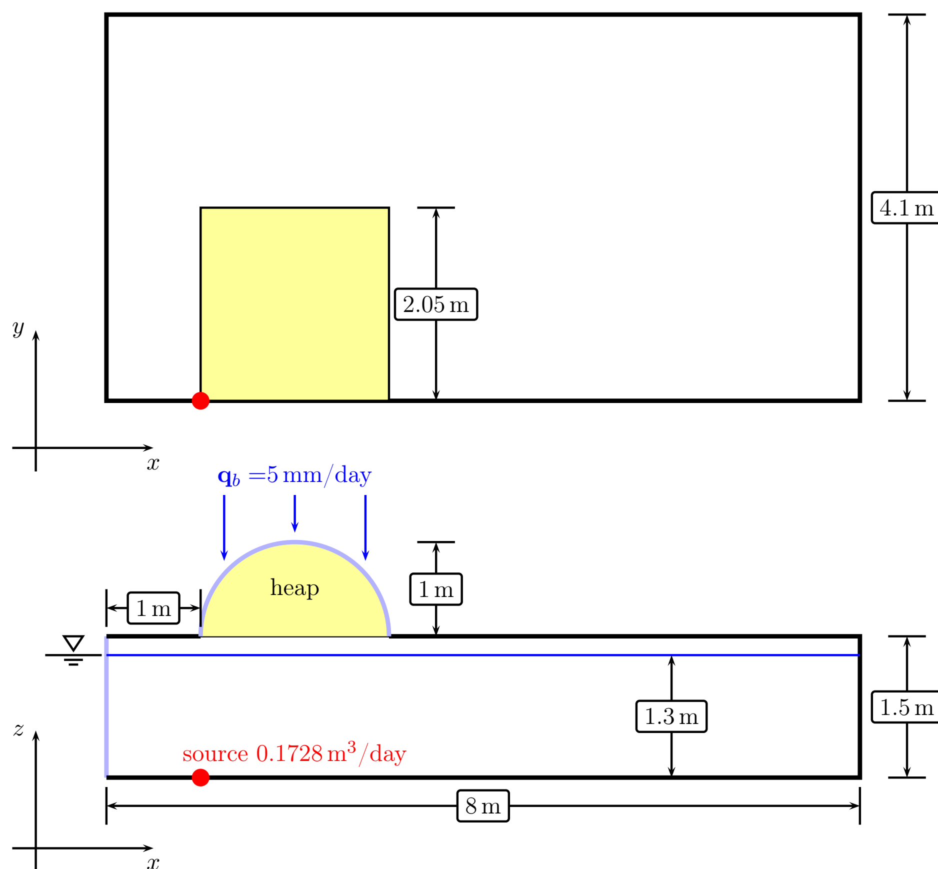


Figure 6.6: The domain for the *heap_leaching* test case.

6.2 Richards' equation: the *dry_infiltration* test case

Mesh convergence tests were performed, and a fine 23,555 node triangular mesh was chosen for the *reference solution*, which was was determined for both the *dry* initial conditions ($\psi^0 = -7.34\,\mathrm{m}$) and the *very dry* initial conditions ($\psi^0 = -100\,\mathrm{m}$) using central weighting and a very tight relative and absolute tight tolerances for the convergence of the Newton method (see equation (3.92)) of $\tau_r = \tau_a = 1 \cdot 10^{-8}$. The resultant saturation contours for the *very dry* case after 30 days are in good agreement with the fine-mesh solution for the same problem by Diersch and Perrochet (1999) in Figure 6.7.

To compare the computational efficiency and accuracy of the edge weighting methods (upstream weighting the flux limiting), and of the different formulations for Richards' equation (the PR and MPR formulations), the solution was computed on meshes with different resolutions using FVMPor compiled for the CPU. Table 6.6 summarises the properties of the three meshes used. Each successive mesh was formed by halving the typical edge length such that the medium resolution mesh has 1406 nodes, which is similar to the 1890-node quadrilateral meshes used in previous analysis of this problem (Forsyth et al., 1995, Diersch and Perrochet, 1999, Therrien et al., 2010).

The computational performance of each combination of limiter and formulation, measured in terms of residual evaluations, wall time and mass balance error, for each mesh is illustrated in Table 6.7 and Table 6.8 for the PR and MPR formulations respectively with the *dry* initial conditions. The same information is tabulated for the more computationally-challenging *very dry* initial conditions in Tables 6.9 and 6.10.

For this test case, it is possible to determine the mass balance error of the computed solutions. The initial mass of fluid in the system, $M_\Omega(t^{(0)})$, can be computed exactly given that the pressure head is constant everywhere in the domain. Furthermore, the only source of mass is the constant flux over the

top left boundary, so that the mass of water in the system at $t^{(m)}$ can then be determined exactly

$$M_\Omega(t^{(m)}) = M_\Omega(t^{(0)}) + \text{net total flux over boundary} \tag{6.1}$$

The total mass of fluid in the computed solution at $t = t^n$ is computed using equation (3.29)

$$\overline{M}_\Omega(t^{(m)}) = \sum_{i=1}^{N_n} \Delta_i M_i^{(m)}. \tag{6.2}$$

The relative mass balance error for the fluid, water phase, e^w_{mb}, is then

$$e^{\text{w}}_{\text{mb}}(t^{(m)}) = \frac{\left|\overline{M}_\Omega(t^{(m)}) - M_\Omega(t^{(m)})\right|}{M_\Omega(t^{(m)})}. \tag{6.3}$$

The same approach is used to determine mass balance error for both the fluid mass and solute mass, e^{w}_{mb} and e^{s}_{mb} respectively, for the coupled flow and transport model.

Comparison of upstream weighting and flux limiters

We now compare the mesh convergence and efficiency of solutions computed using upstream weighting and flux limiting. First, it is important to note that for a given weighting the solutions determined using the PR and MPR formulations are found to be visually indistinguishable. Furthermore, for this test case, the difference between solutions computed using the different limiters are very small. Hence, we take PR solutions computed using the parabolic limiter with the flow direction indicator FDI($\boldsymbol{q}$) (defined in (3.48)) as being representative of flux limiting, and compare them to contours for upstream weighting with FDI($\boldsymbol{q}$) in Figures 6.9 to 6.10.

The upstream solutions exhibit significant numerical diffusion, whereby contours of higher-saturation values are under-estimated, and the lower saturation contours are over-estimated. The numerical diffusion decreases as the

mesh is refined, however the fine mesh solution is a poor match to the reference solution on the left hand side of the domain where the infiltrating front moves fastest, for both the *dry* and the *very dry* initial conditions.

The solutions computed using flux limiting on the medium mesh are considerably more accurate than the most accurate fine mesh solution. The fine mesh solution using flux limiting captures the reference solution very well for both the *dry* and the *very dry* initial conditions, and the flux limited solutions on the medium mesh are more accurate than the fine mesh solutions for upstream weighting.

There is a small computational overhead, both in terms of residual evaluations and wall time, associated with using flux limiting compared to upstream weighting. However, upstream weighting requires much finer meshes to compute solutions with equivalent accuracy to flux limited solutions on coarse meshes. Indeed, the flux limited solutions on the medium mesh were more accurate than those computed using upstream weighting on the fine mesh, while requiring about 5 times and 7 times less computational effort to compute for the *dry* and *very dry* initial conditions respectively (in terms of wall time).

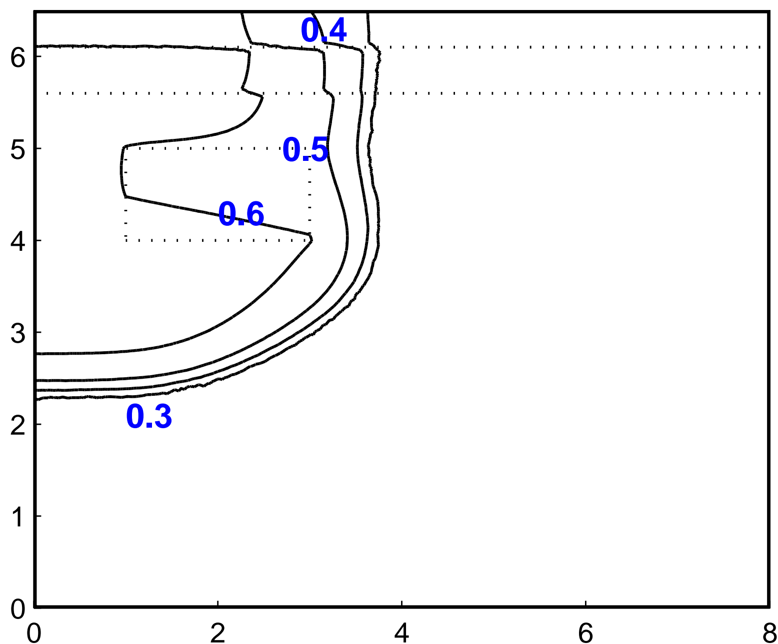
Comparison of the flux limiters

As already mentioned, the solutions for each of the van Leer and parabolic limiters are very similar, capturing the location of the infiltrating front better than the upstream solution. There is no clear trend to favour choosing one flow direction indicator over another from the convergence and error measurements in Tables 6.7–6.10, save for the observation that the van Leer limiter is less efficient for FDI($\boldsymbol{q}$) in some test runs.

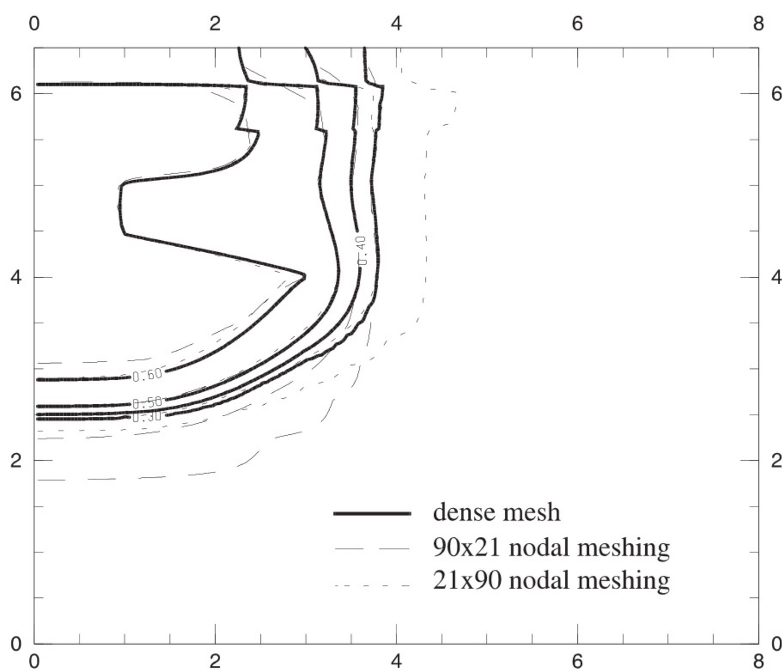
Comparison of the PR and MPR formulations

There is a computational overhead of between 10-20% in terms of wall time imposed by the MPR formulation over the PR formulation. Both formulations require roughly the same number of residual evaluations, which implies that the additional wall time is due to the expense imposed by solving the additional equation at each node for the MPR formulation. The overhead is small when we consider that the system of equations for the MPR formulation has twice as many equations and variables as the PR formulation. The relatively small overhead illustrates the effectiveness of the Schur complement preconditioner (3.103), and the relatively low cost of evaluating the additional equation (3.75).

Both the PR and MPR formulations are mass conservative, as illustrated in Figure 6.8. After a small initial increase, the magnitude of the mass balance error does not grow for either formulation, which reflects the mass-conservative nature of the CV-FE spatial discretisation. We note that while both formulations are mass conservative, the mass balance error for the MPR formulation is between one to two orders of magnitude lower than that for the PR formulation for the different meshes and edge weighting schemes in Tables 6.7–6.10. A more detailed analysis of the conservation of mass for each formulation is presented in §6.7.2.



(a) Fine mesh results from this work.



(b) Fine mesh contours from Diersch and Perrochet (1999).

Figure 6.7: Fine mesh solution for the *dry_infiltration* test case with *very dry* initial conditions of $\psi^0 = -100\,\text{m}$ after 30 days. The contours are for saturation: $S_w = 0.6$, $S_w = 0.5$, $S_w = 0.4$ and $S_w = 0.3$.

	edge length	nodes	edges	faces
coarse	0.4 m	367	1,030	2,128
medium	0.2 m	1406	4,076	8,291
fine	0.1 m	5,850	17,265	34,812

Table 6.6: Details of the meshes used to verify the *dry_infiltration* test case.

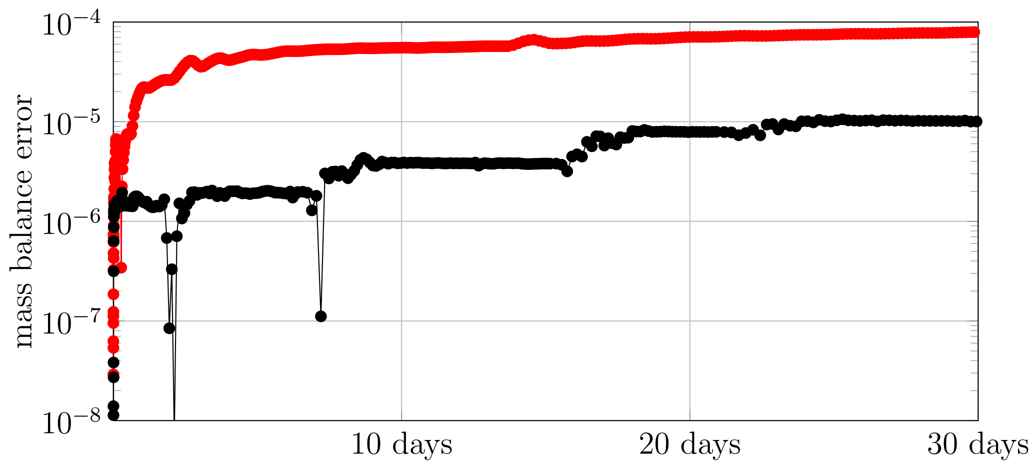


Figure 6.8: Time evolution of the mass balance error for the PR (red) and MPR (black) formulations solving the *dry_infiltration* test case with *very dry* initial conditions ($\psi^0 = -100\,\text{m}$), with upstream weighting on the medium resolution mesh.

	Upwind Weighting			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	0.47	0.44	0.77
	$\boldsymbol{F}$ eval	1260	1275	2300
	mass balance	$2.0 \cdot 10^{-5}$	$2.7 \cdot 10^{-5}$	$2.5 \cdot 10^{-5}$
medium	Wall Time (s)	4.08	3.76	3.98
	$\boldsymbol{F}$ eval	3247	3095	3297
	mass balance	$9.5 \cdot 10^{-5}$	$9.1 \cdot 10^{-5}$	$1.0 \cdot 10^{-4}$
fine	Wall Time (s)	31.90	31.49	30.85
	$\boldsymbol{F}$ eval	6062	6142	6028
	mass balance	$1.0 \cdot 10^{-4}$	$9.9 \cdot 10^{-5}$	$1.0 \cdot 10^{-4}$

	van Leer Limiter			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	0.77	1.12	0.99
	$\boldsymbol{F}$ eval	2124	2684	2591
	mass balance	$9.6 \cdot 10^{-5}$	$5.3 \cdot 10^{-5}$	$3.0 \cdot 10^{-5}$
medium	Wall Time (s)	7.21	5.84	5.52
	$\boldsymbol{F}$ eval	5723	3822	3932
	mass balance	$3.0 \cdot 10^{-4}$	$1.1 \cdot 10^{-4}$	$7.2 \cdot 10^{-5}$
fine	Wall Time (s)	45.87	39.57	42.75
	$\boldsymbol{F}$ eval	8554	6138	7258
	mass balance	$2.5 \cdot 10^{-4}$	$2.3 \cdot 10^{-4}$	$1.2 \cdot 10^{-4}$

	Parabolic Limiter			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	0.65	0.72	0.93
	$\boldsymbol{F}$ eval	1800	1691	2419
	mass balance	$3.3 \cdot 10^{-5}$	$3.2 \cdot 10^{-5}$	$2.1 \cdot 10^{-5}$
medium	Wall Time (s)	4.42	5.52	5.03
	$\boldsymbol{F}$ eval	3479	3610	3594
	mass balance	$1.3 \cdot 10^{-4}$	$8.6 \cdot 10^{-5}$	$8.4 \cdot 10^{-5}$
fine	Wall Time (s)	30.96	38.36	37.29
	$\boldsymbol{F}$ eval	5808	5950	6307
	mass balance	$2.7 \cdot 10^{-4}$	$2.4 \cdot 10^{-4}$	$2.1 \cdot 10^{-4}$

Table 6.7: Statistics for the solution of the *dry_infiltration* test case with *dry* initial conditions ($\psi^0 = -7.34$) using each combination of spatial averaging and mesh resolution with the PR formulation.

	Upwind Weighting			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	0.49	0.48	0.98
	$\boldsymbol{F}$ eval	1212	1238	2637
	mass balance	$5.3 \cdot 10^{-6}$	$5.3 \cdot 10^{-6}$	$5.3 \cdot 10^{-6}$
medium	Wall Time (s)	4.68	4.45	4.48
	$\boldsymbol{F}$ eval	3337	3252	3292
	mass balance	$5.3 \cdot 10^{-6}$	$5.6 \cdot 10^{-6}$	$5.3 \cdot 10^{-6}$
fine	Wall Time (s)	33.77	36.21	32.98
	$\boldsymbol{F}$ eval	5702	6253	5700
	mass balance	$3.9 \cdot 10^{-6}$	$3.9 \cdot 10^{-6}$	$3.1 \cdot 10^{-6}$

	van Leer Limiter			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	1.01	0.91	1.38
	$\boldsymbol{F}$ eval	2562	1968	3247
	mass balance	$7.7 \cdot 10^{-6}$	$8.9 \cdot 10^{-6}$	$8.0 \cdot 10^{-6}$
medium	Wall Time (s)	7.00	5.66	5.49
	$\boldsymbol{F}$ eval	4941	3358	3507
	mass balance	$1.1 \cdot 10^{-5}$	$4.1 \cdot 10^{-6}$	$2.9 \cdot 10^{-6}$
fine	Wall Time (s)	30.31	44.79	39.19
	$\boldsymbol{F}$ eval	4907	6267	5891
	mass balance	$2.3 \cdot 10^{-5}$	$5.3 \cdot 10^{-6}$	$3.9 \cdot 10^{-6}$

	Parabolic Limiter			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	0.68	0.90	0.62
	$\boldsymbol{F}$ eval	1720	1941	1422
	mass balance	$4.6 \cdot 10^{-6}$	$4.9 \cdot 10^{-6}$	$1.1 \cdot 10^{-5}$
medium	Wall Time (s)	5.99	6.81	5.51
	$\boldsymbol{F}$ eval	4273	4038	3508
	mass balance	$5.8 \cdot 10^{-6}$	$8.9 \cdot 10^{-6}$	$4.5 \cdot 10^{-6}$
fine	Wall Time (s)	39.40	44.16	48.08
	$\boldsymbol{F}$ eval	6603	6161	7216
	mass balance	$6.8 \cdot 10^{-6}$	$5.9 \cdot 10^{-6}$	$3.4 \cdot 10^{-6}$

Table 6.8: Statistics for the solution of the *dry_infiltration* test case with *dry* initial conditions ($\psi^0 = -7.34$) using each combination of spatial averaging and mesh resolution with the MPR formulation.

	Upwind Weighting			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	1.16	1.17	1.08
	$\boldsymbol{F}$ eval	3323	3505	3242
	mass balance	$1.6 \cdot 10^{-4}$	$1.2 \cdot 10^{-4}$	$1.5 \cdot 10^{-4}$
medium	Wall Time (s)	7.52	7.23	6.92
	$\boldsymbol{F}$ eval	6041	5990	5776
	mass balance	$7.9 \cdot 10^{-5}$	$7.2 \cdot 10^{-5}$	$7.9 \cdot 10^{-5}$
fine	Wall Time (s)	52.26	51.01	50.49
	$\boldsymbol{F}$ eval	10017	10137	10059
	mass balance	$2.6 \cdot 10^{-5}$	$2.5 \cdot 10^{-5}$	$2.6 \cdot 10^{-5}$

	van Leer Limiter			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	1.42	1.12	1.10
	$\boldsymbol{F}$ eval	4001	3311	3237
	mass balance	$7.5 \cdot 10^{-5}$	$1.4 \cdot 10^{-4}$	$1.1 \cdot 10^{-4}$
medium	Wall Time (s)	10.30	6.71	6.44
	$\boldsymbol{F}$ eval	8122	5518	5297
	mass balance	$3.9 \cdot 10^{-4}$	$6.8 \cdot 10^{-5}$	$6.7 \cdot 10^{-5}$
fine	Wall Time (s)	94.89	57.70	58.01
	$\boldsymbol{F}$ eval	18175	11359	11455
	mass balance	$6.0 \cdot 10^{-4}$	$3.7 \cdot 10^{-4}$	$2.7 \cdot 10^{-4}$

	Parabolic Limiter			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	1.23	1.05	1.11
	$\boldsymbol{F}$ eval	3453	3101	3302
	mass balance	$1.7 \cdot 10^{-4}$	$1.6 \cdot 10^{-4}$	$1.4 \cdot 10^{-4}$
medium	Wall Time (s)	7.68	6.19	6.28
	$\boldsymbol{F}$ eval	6091	5074	5186
	mass balance	$1.0 \cdot 10^{-4}$	$7.3 \cdot 10^{-5}$	$8.4 \cdot 10^{-5}$
fine	Wall Time (s)	55.24	56.47	56.06
	$\boldsymbol{F}$ eval	10549	11070	11073
	mass balance	$3.4 \cdot 10^{-5}$	$2.7 \cdot 10^{-4}$	$3.6 \cdot 10^{-4}$

Table 6.9: Statistics for the solution of the *dry_infiltration* test case with *very dry* initial conditions ($\psi^0 = -100m$) using each combination of spatial averaging and mesh resolution with the PR formulation.

	Upwind Weighting			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	1.28	1.28	1.13
	$\boldsymbol{F}$ eval	3296	3383	3026
	mass balance	$3.0 \cdot 10^{-6}$	$2.8 \cdot 10^{-6}$	$2.8 \cdot 10^{-6}$
medium	Wall Time (s)	7.15	6.94	6.70
	$\boldsymbol{F}$ eval	5063	5088	4914
	mass balance	$1.1 \cdot 10^{-5}$	$4.4 \cdot 10^{-6}$	$7.8 \cdot 10^{-6}$
fine	Wall Time (s)	60.54	56.94	58.46
	$\boldsymbol{F}$ eval	10323	9904	10231
	mass balance	$3.8 \cdot 10^{-6}$	$4.6 \cdot 10^{-6}$	$3.7 \cdot 10^{-6}$

	van Leer Limiter			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	1.71	1.29	1.45
	$\boldsymbol{F}$ eval	4348	3408	3830
	mass balance	$4.4 \cdot 10^{-6}$	$3.4 \cdot 10^{-6}$	$3.5 \cdot 10^{-6}$
medium	Wall Time (s)	13.36	8.22	9.13
	$\boldsymbol{F}$ eval	9492	5970	6685
	mass balance	$1.0 \cdot 10^{-5}$	$3.7 \cdot 10^{-6}$	$3.0 \cdot 10^{-6}$
fine	Wall Time (s)	85.43	70.91	74.13
	$\boldsymbol{F}$ eval	14273	12311	12893
	mass balance	$9.9 \cdot 10^{-6}$	$5.4 \cdot 10^{-6}$	$6.9 \cdot 10^{-6}$

	Parabolic Limiter			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	1.16	1.30	1.14
	$\boldsymbol{F}$ eval	2938	3404	3029
	mass balance	$5.1 \cdot 10^{-6}$	$3.1 \cdot 10^{-6}$	$6.1 \cdot 10^{-6}$
medium	Wall Time (s)	7.58	7.86	8.18
	$\boldsymbol{F}$ eval	5365	5707	5974
	mass balance	$6.4 \cdot 10^{-6}$	$5.6 \cdot 10^{-6}$	$2.9 \cdot 10^{-6}$
fine	Wall Time (s)	66.52	75.44	70.54
	$\boldsymbol{F}$ eval	11211	13046	12274
	mass balance	$6.2 \cdot 10^{-6}$	$1.3 \cdot 10^{-5}$	$5.5 \cdot 10^{-6}$

Table 6.10: Statistics for the solution of the *dry_infiltration* test case with *very dry* initial conditions ($\psi^0 = -100m$) using each combination of spatial averaging and mesh resolution with the MPR formulation.

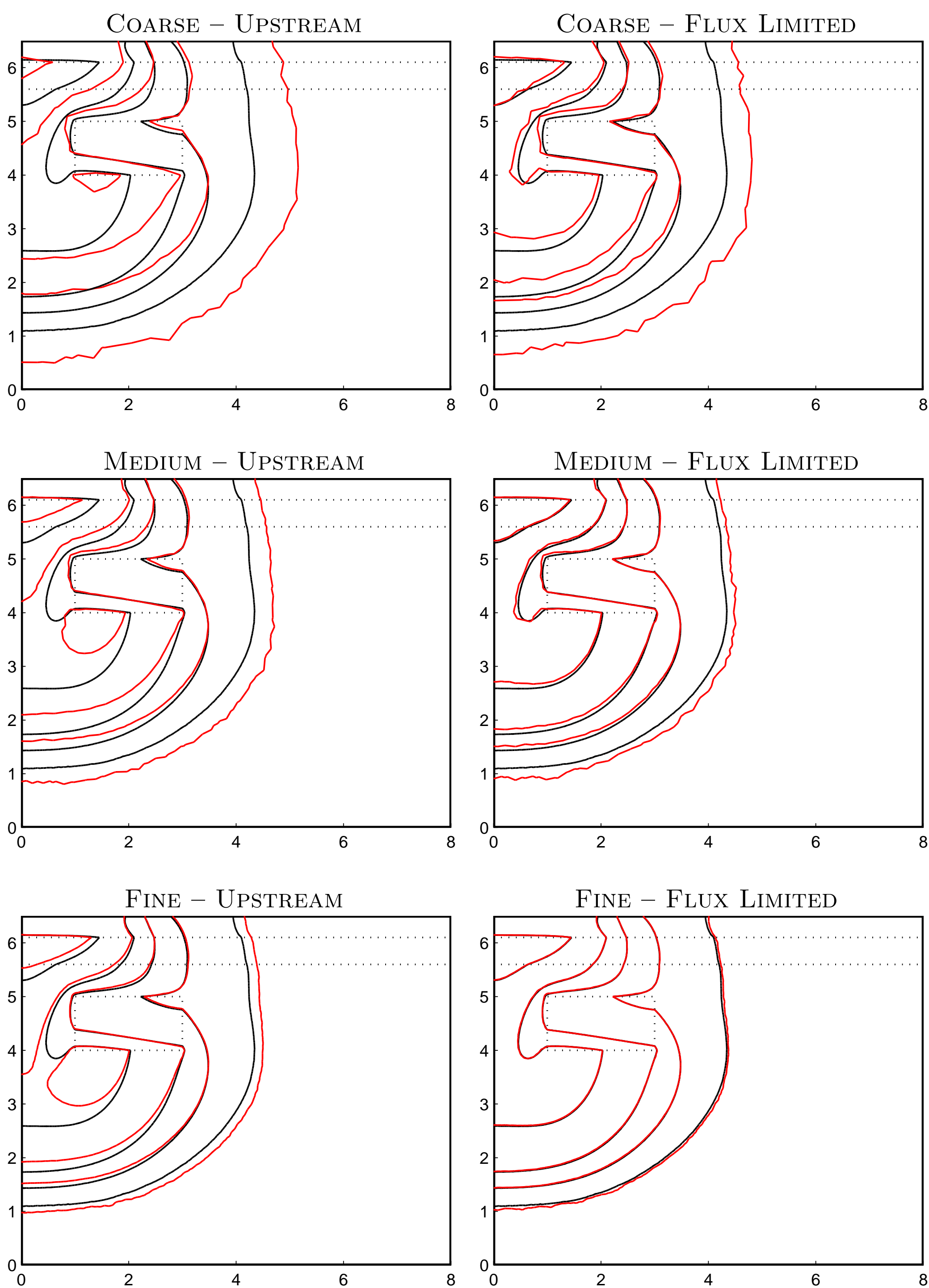


Figure 6.9: Pressure head contours for the *dry_infiltration* test case at 30 days with the *dry* initial conditions ($\psi^0 = -7.34m$). Pressure head contours are -7 m, -1.5 m, -1 m, -0.8 m and -0.7 m.

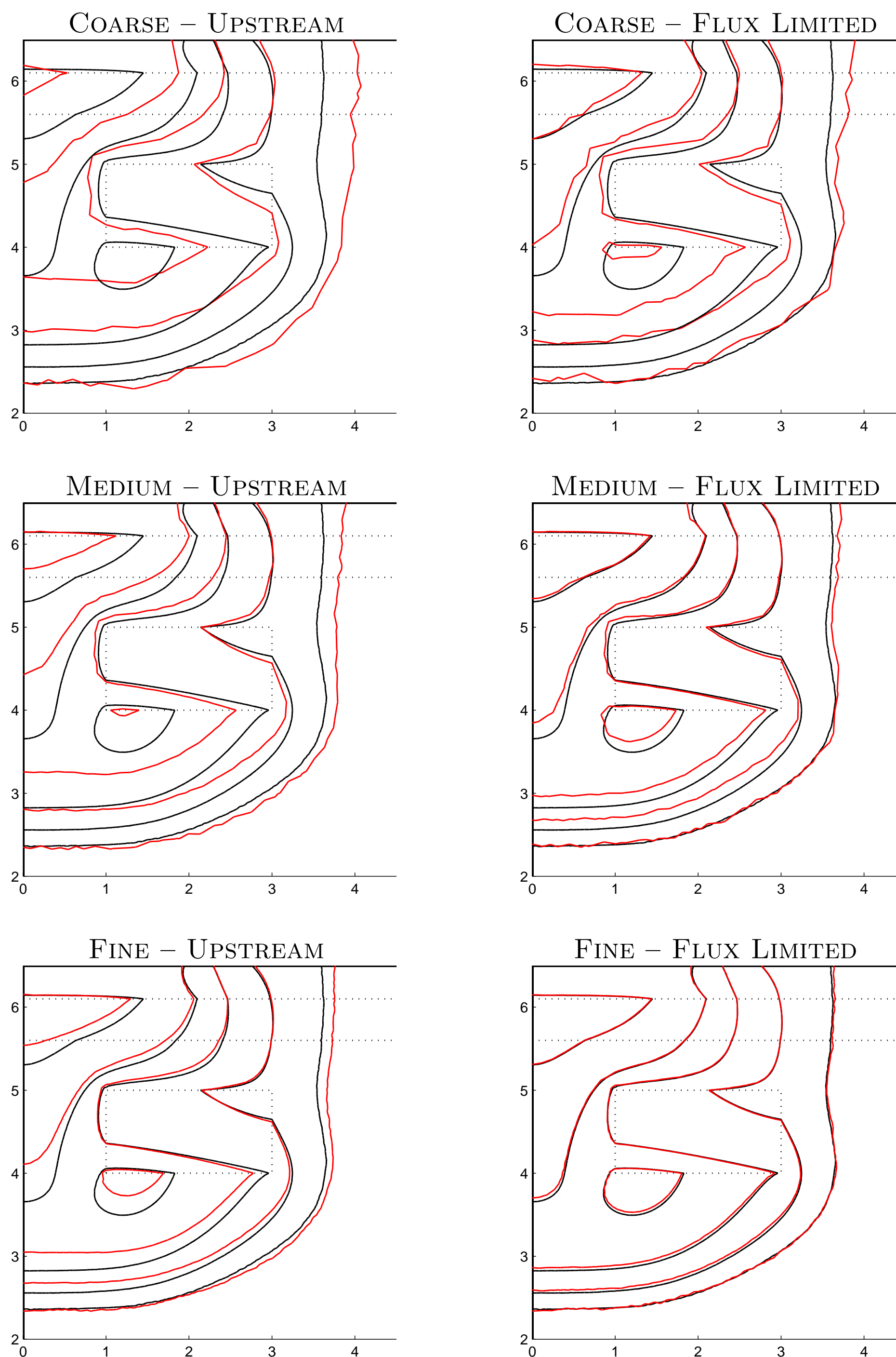


Figure 6.10: Pressure head contours for the *dry_infiltration* test case at 30 days with the *very dry* initial conditions ($\psi^0 = -100m$). Pressure head contours are -7 m, -1.5 m, -1 m, -0.8 m and -0.7 m.

6.3 Richards' equation: the *water_table* test case

This test case differs from the previous *dry_infiltration* test case because it has both saturated and unsaturated conditions, and because it has experimental results against which the numerical results can be verified. The original experiment was performed in a three-dimensional flow tank, however subsequent computational models of the experiment have been performed with two-dimensional meshes (Vauclin et al., 1979, Fahs et al., 2009). To test the efficacy of the different edge-based weighting schemes and formulations in three dimensions, the numerical investigation presented here compare results computed using a three-dimensional mesh of the flow tank against a reference solution found on a fine two-dimensional mesh.

The reference solution was determined using a fine two-dimensional triangular mesh with 25,318 nodes and central weighting for the mobility term. The height of the water table in the reference solution is shown in Figure 6.11. The reference solution agrees very well with the observed water table heights from the experiment, which are also illustrated in Figure 6.11.

To form the three-dimensional meshes used in the numerical experiments, the domain was extruded in the third dimension (along the y-axis) by 0.5 m, and meshed with unstructured tetrahedra. Meshes of varying resolutions are described in Table 6.11, with the finest mesh offering equivalent resolution to the triangular mesh in the analysis of Fahs et al. (2009). Relative and absolute tolerances of $\tau_a = 1 \cdot 10^{-3}$ and $\tau_r = 1 \cdot 10^{-7}$ respectively were used, along with an ILUT preconditioner[1]. Computation was performed on the GPU due to the computational cost of three-dimensional simulations.

The number of residual evaluations and the wall time required to solve the problem for each mesh and spatial weighting scheme are illustrated in Table 6.12 and Table 6.13 for the PR and MPR formulations respectively. Mass

[1] See §6.7.1 for more information about the choice of preconditioner for this test case.

balance errors are not given because an exact analytic expression is not available for flux over the hydrostatic boundary.

Comparison of upstream weighting and flux limiters

The height of the water table for coarse, medium and fine mesh solutions using upstream weighting and the parabolic flux limiter with FDI(φ) are compared to the reference mesh solution in Figure 6.12[2]. We note that, as for the *dry_infiltration* test case, solutions obtained using either the PR or MPR formulation are visually indistinguishable, as are solutions for the van Leer and parabolic flux limiters.

At $t = 8$ hours, the location of the water table in Figure 6.11 is very close to the steady state solution for this test case. Solutions computed using both upstream weighting and flux limiting capture the solution well at this point, with the exception of upstream weighting on the coarse mesh. The location of the water table at earlier times is more challenging to reproduce, with the upstream solutions all under-predicting its height at t =2, 3 and 4 hours.

Compared to upstream weighting, flux limiting reproduces the location of the water table far better: the flux limited solution on the coarse mesh is more accurate than the fine mesh solution computed using upstream weighting, and the flux limited solution on the medium mesh reproduces the water table location very well at all times. Hence, substantial gains in efficiency are made by using flux limiting for this test case, by virtue of being able to use coarser meshes: the coarse and medium mesh solutions take less than 2 seconds and approximately 11 seconds to compute using flux limiting, compared to the upstream solutions that take approximately 40 seconds.

[2] The two-dimensional contours were taken from the pressure head values at the front of the tank.

Comparison of different flux limiters

The computational performance of the edge weighting schemes is more sensitive to the choice of the flow direction indicator for this test case. Both upstream weighting and the van Leer limiter failed to converge when using FDI(φ) for some of the test runs, and the van Leer limiter was significantly more expensive than other methods on fine meshes for some runs, requiring 40%–50% more computational effort (c.f. the fine mesh solutions using FDI($\boldsymbol{q}$) and FDI($\boldsymbol{\nabla}\phi$) in Table 6.12). There were no such problems for the parabolic limiter, which was efficient for all combinations of FDI and mesh resolution. Finally, we note that the flux limited results are very accurate on the medium mesh, for which there is little or no additional computational cost associated with using either the van Leer or parabolic limiters in place of upstream weighting.

The parabolic limiter is selected as the best edge-based weighting for this test case. It computed the solution accurately and efficiently for each flow direction indicator on all of the meshes. The van Leer limiter was also accurate, however it was less reliable: it failed to converge using FDI(φ) in some cases, and was inefficient for some of the fine mesh solutions.

Comparison of the formulations

As was observed earlier, the solutions for the PR and MPR formulations are visually indistinguishable. In terms of computational effort required to find the solution, the PR formulation requires less effort than the MPR formulation when using upstream weighting by 15% and 25% respectively for the medium and fine mesh resolutions. However, the difference between the two approaches is less significant when using either of the flux limiters on the medium mesh. Given that the mass balance error is not available for this problem, it is not possible to draw any conclusions about the relative merits of either formulation in terms of conservation of balance.

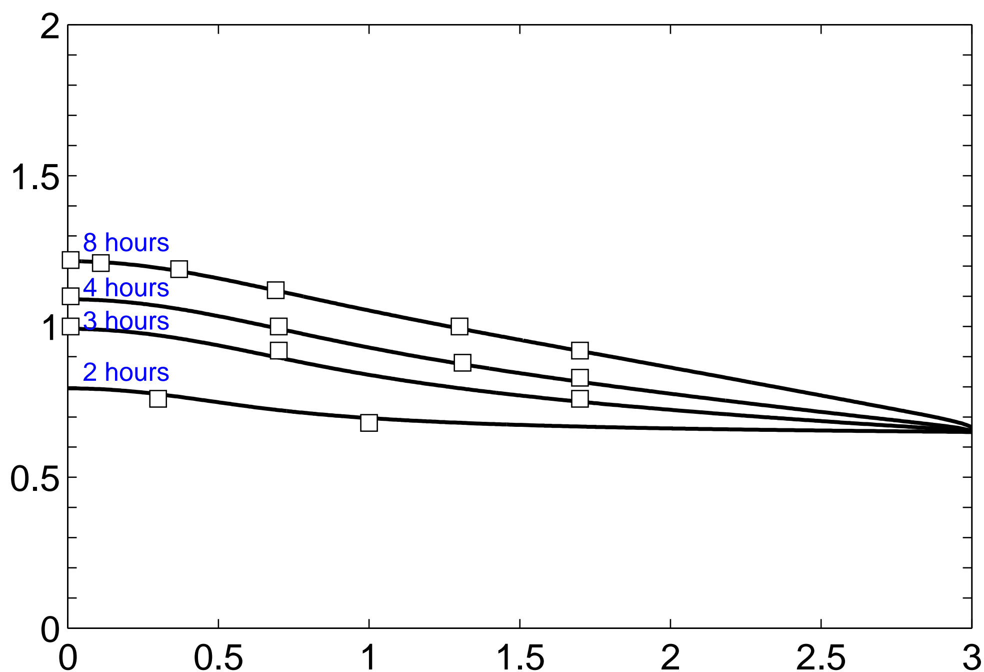


Figure 6.11: Reference solution for *water_table* test case. The computed water table height (the black lines), along with the measured water table heights from the experiment (the black boxes) are shown for t =2, 3, 4, and 8 hours.

	edge length	nodes	edges	faces
coarse	0.2 m	869	4,843	23,850
medium	0.1 m	5,515	34,142	171,768
fine	0.067 m	15,943	102,603	519,966

Table 6.11: Details of the three-dimensional tetrahedral meshes used to test the *water_table* test case.

	Upwind Weighting			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	1.12	1.09	-
	$\boldsymbol{F}$ eval	1305	1305	-
medium	Wall Time (s)	10.98	10.41	15.64
	$\boldsymbol{F}$ eval	3578	3479	5410
fine	Wall Time (s)	39.95	47.37	41.91
	$\boldsymbol{F}$ eval	5054	5982	5698

	van Leer Limiter			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	1.90	1.35	-
	$\boldsymbol{F}$ eval	2170	1575	-
medium	Wall Time (s)	13.70	11.11	14.84
	$\boldsymbol{F}$ eval	4554	3692	4884
fine	Wall Time (s)	100.88	100.31	42.03
	$\boldsymbol{F}$ eval	12152	12034	5623

	Parabolic Limiter			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	1.18	1.12	1.10
	$\boldsymbol{F}$ eval	1345	1299	1317
medium	Wall Time (s)	10.65	13.49	11.03
	$\boldsymbol{F}$ eval	3350	4403	3655
fine	Wall Time (s)	58.50	59.12	51.04
	$\boldsymbol{F}$ eval	7122	7103	6462

Table 6.12: Statistics for the solution of the *water_table* case study using each combination of spatial averaging and mesh resolution with the PR formulation.

	Upwind Weighting			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	1.33	1.30	-
	$\boldsymbol{F}$ eval	1524	1524	-
medium	Wall Time (s)	12.87	12.26	16.15
	$\boldsymbol{F}$ eval	4271	4146	5543
fine	Wall Time (s)	53.36	61.36	-
	$\boldsymbol{F}$ eval	6613	7688	-

	van Leer Limiter			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	1.99	1.80	-
	$\boldsymbol{F}$ eval	2194	2121	-
medium	Wall Time (s)	14.97	11.91	17.81
	$\boldsymbol{F}$ eval	4933	3913	5997
fine	Wall Time (s)	98.22	60.26	52.39
	$\boldsymbol{F}$ eval	12054	7749	7068

	Parabolic Limiter			
	Metric	FDI($\boldsymbol{q}$)	FDI($\boldsymbol{\nabla}\phi$)	FDI(φ)
coarse	Wall Time (s)	2.19	1.48	2.13
	$\boldsymbol{F}$ eval	2562	1672	2636
medium	Wall Time (s)	13.00	12.99	11.49
	$\boldsymbol{F}$ eval	4207	4242	3776
fine	Wall Time (s)	58.20	54.78	58.91
	$\boldsymbol{F}$ eval	7178	6929	7542

Table 6.13: Statistics for the solution of the *water_table* test case using each combination of spatial averaging and mesh resolution with the MPR formulation.

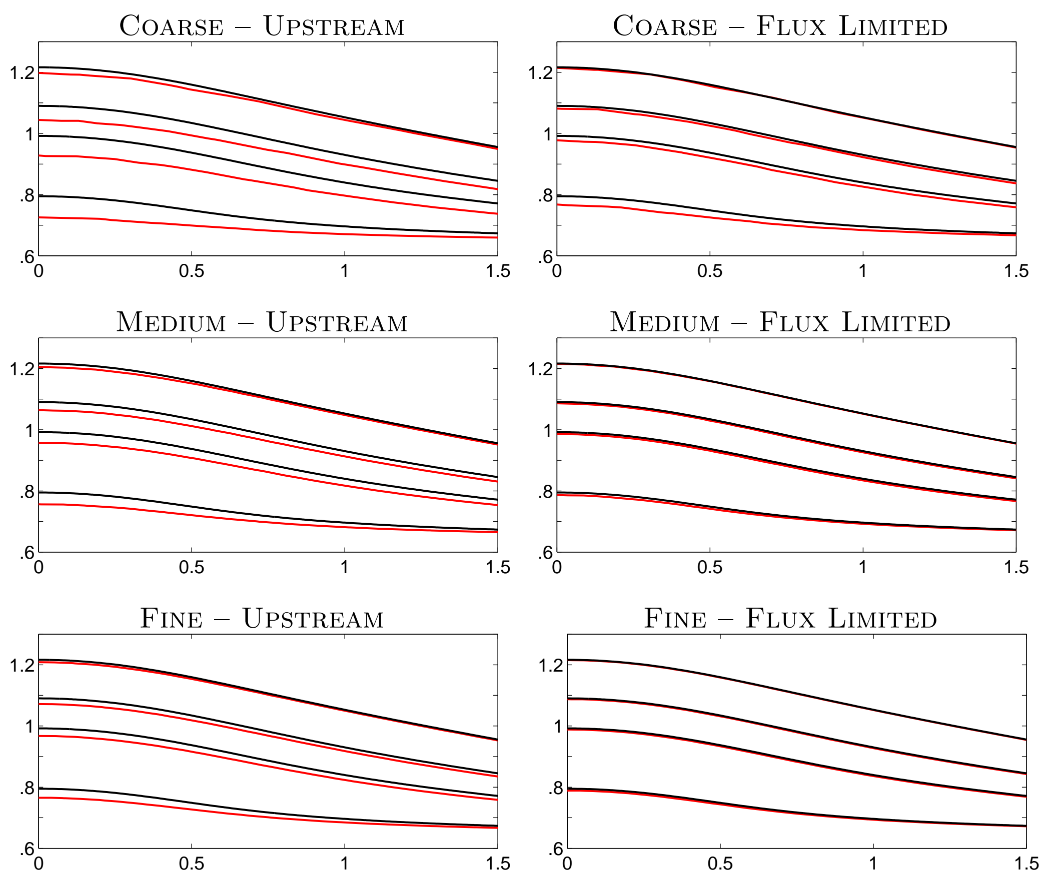


Figure 6.12: Comparison of the water table height in the reference solution (black lines) for the *water_table* test case to upstream and flux limited solutions (red lines). The flux limited solutions were computed using the parabolic limiter with FDI(φ). Only the region of interest, where the greatest change in the water table height takes place, is illustrated.

6.4 Transport Model: Unsaturated flow and transport

Mesh convergence was reached for this test case on a triangular mesh with 94,101 nodes, which was selected for computing the reference solution. The pressure head and concentration contours of the reference solution after 48 hours are illustrated in Figure 6.15. After 48 hours, the contaminated fluid from the point source has flowed downwards where it pools on top of, then flows around, the low-permeability region. Due to the very small dispersion values of $\alpha_L = \alpha_T = 1\,\text{mm}$ this test case is advection-dominated, which makes the solution for the solute concentration particularly prone to numerical diffusion.

To analyse the different edge weighting schemes and formulations, test were performed on three meshes described in Table 6.16. Computation was performed on the GPU with the ILU(0) preconditioner, and error tolerances of τ_r=1×10^{-3} and τ_a=1×10^{-6}. The wall time, number of residual evaluations and the mass balance error for the water and solute (e^{w}_{mb} and e^{s}_{mb} respectively, computed using equation (6.3)) are tabulated for the PC and MMPC formulations in Table 6.14 and Table 6.15.

Comparison of upstream weighting and flux limiters

Comparisons of the solutions for pressure head and concentration using different edge weighting schemes are illustrated in Figure 6.13 and Figure 6.14 respectively. Because the solution is symmetric about the vertical axis, only solutions in the half plane are shown, although the solution was computed for the entire domain in each case. Solutions are illustrated for both the parabolic and van Leer limiters applied to the mobility term, because they exhibit significant qualitative differences. On the other hand, the choice of limiter for the advection terms has very little qualitative effect on the solution, so only results for the van Leer limiter are shown.

First, we investigate the edge weighting method applied to the mobility term. The contours of pressure head, illustrated in Figure 6.13, are most difficult to compute accurately inside and below the low-permeability region. Of the upstream and flux limiter variants, the parabolic limiter performs the best, matching the reference solution well at all points on the medium mesh except for the $\psi = -0.6\,\mathrm{m}$ contour directly below the low-permeability region. All of the methods fail to reproduce this contour accurately at any mesh resolution, with the van Leer limiter performing best on the fine mesh. However, the van Leer limiter produces results that are less accurate than the parabolic limiter elsewhere in and below the low-permeability zone.

Furthermore, inspection of the wall times in Tables 6.14 and 6.15 shows that the van Leer limiter also requires up to two times the computational effort of the parabolic limiter. The overhead of using the parabolic limiter on the mobility term instead of upstream weighting is between 15% and 25% in wall time. However the medium mesh solution for the parabolic limiter is more accurate than the fine mesh solution using upstream weighting, and takes 3.8/5.3 times less computational work for the PC/MMPC formulations respectively.

The choice of edge weighting method for the advection terms is important for this test case, due to numeric diffusion in solutions of the advection-dominated solute transport. This is apparent in the concentration solutions computed using upstream weighting in Figure 6.14, which exhibit excessive numeric diffusion, even on the fine mesh. Flux limiting captures the concentration solution reasonably well on the medium resolution mesh (considerably better than the fine mesh upstream solution), however the fine mesh is required to match the reference solution closely. From Tables 6.14 and 6.15, both the parabolic and van Leer limiters require almost identical computational work[3], and produce visually-indistinguishable solutions. Hence, there is no preference for choosing one limiter over the other for this test case. We also note that the choice of limiter on the mobility term appears to have

[3]Here we assume that the parabolic limiter is used for the mobility term.

negligible effect on the solution for the concentration, even though it clearly affects the pressure head solution in Figure 6.13.

Comparison of PC and MMPC formulations

The mass balance errors given in Tables 6.14 and 6.15 show that both the PC and MMPC formulations offer excellent conservation of mass due to the mass conservative CV-FE discretisation. The MMPC formulation has better mass balance on average than the PC formulation, however the difference between the two formulations is less pronounced than was the case with the PR and MPR formulations for Richards' equation.

For this test case, the MMPC formulation has a computational overhead of approximately 15% relative to the PC formulation, with a large discrepancy of 51% for the fine mesh solution with upstream weighting. Take for example the medium mesh solutions with the parabolic limiter applied to the mobility term in Tables 6.14 and 6.15, where the MMPC formulation has wall times approximately 14% larger than the PC formulation.

In §6.7.2 it will be shown that the PR formulation of Richards' equation requires higher-order temporal integration to produce good conservation of mass relative to the modified mixed MPR formulation. This is also true for the PC and MMPC formulations, however the difference between the formulations for higher-order integration is less pronounced, as is evident in these results, which were computed using third-order temporal integration. Hence, for this test case there is little benefit in terms of mass balance to justify using the MMPC formulation, with the additional computational overhead that it requires, over the PC formulation.

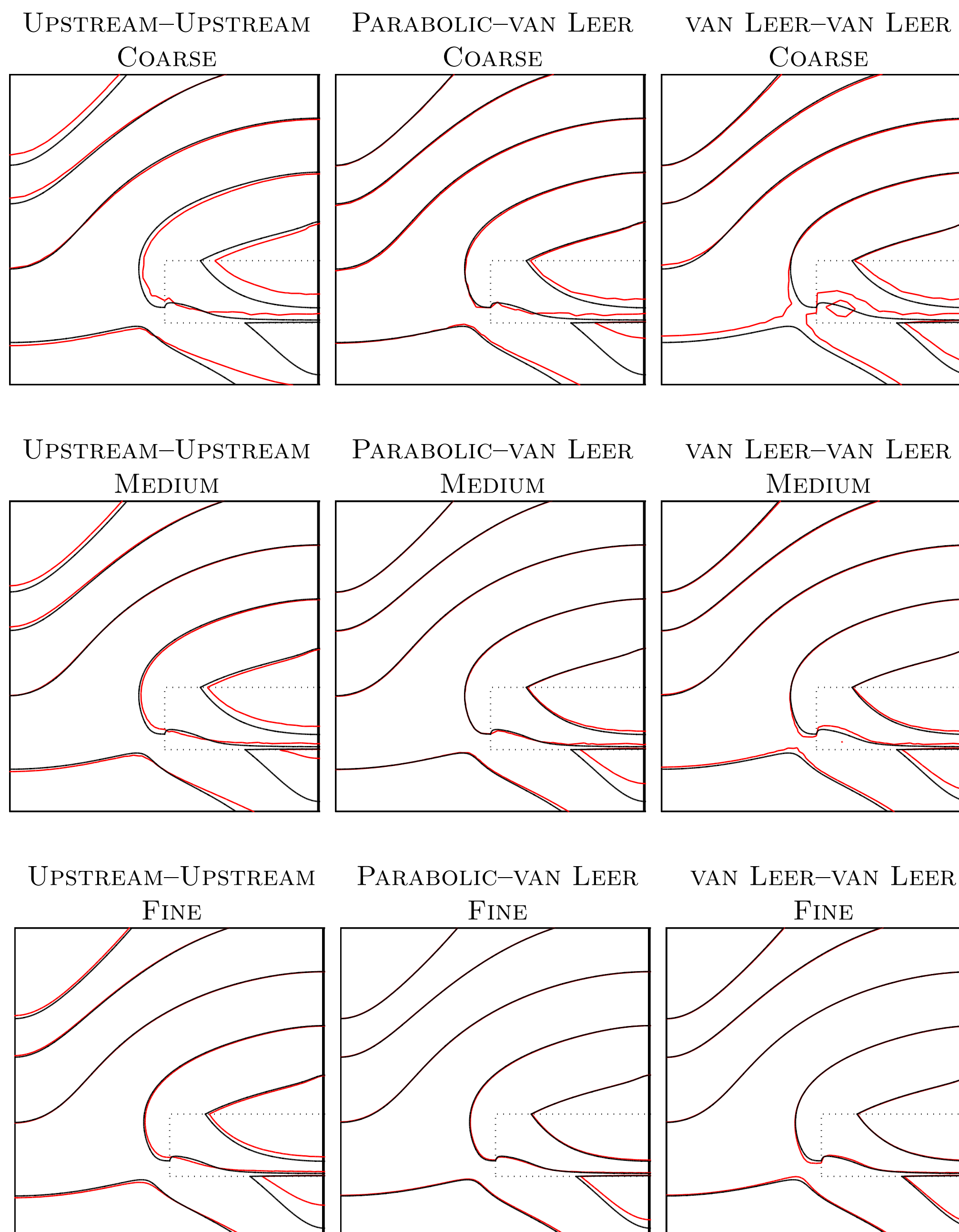


Figure 6.13: Comparison of pressure head contours obtained using different spatial weighting schemes at $t = 48$ hours for the *unsaturated_transport* test case. The limiter used on the mobility and advection terms are written Mobility–Advection above each plot. The contours are the same as those in the reference solution Figure 6.15.

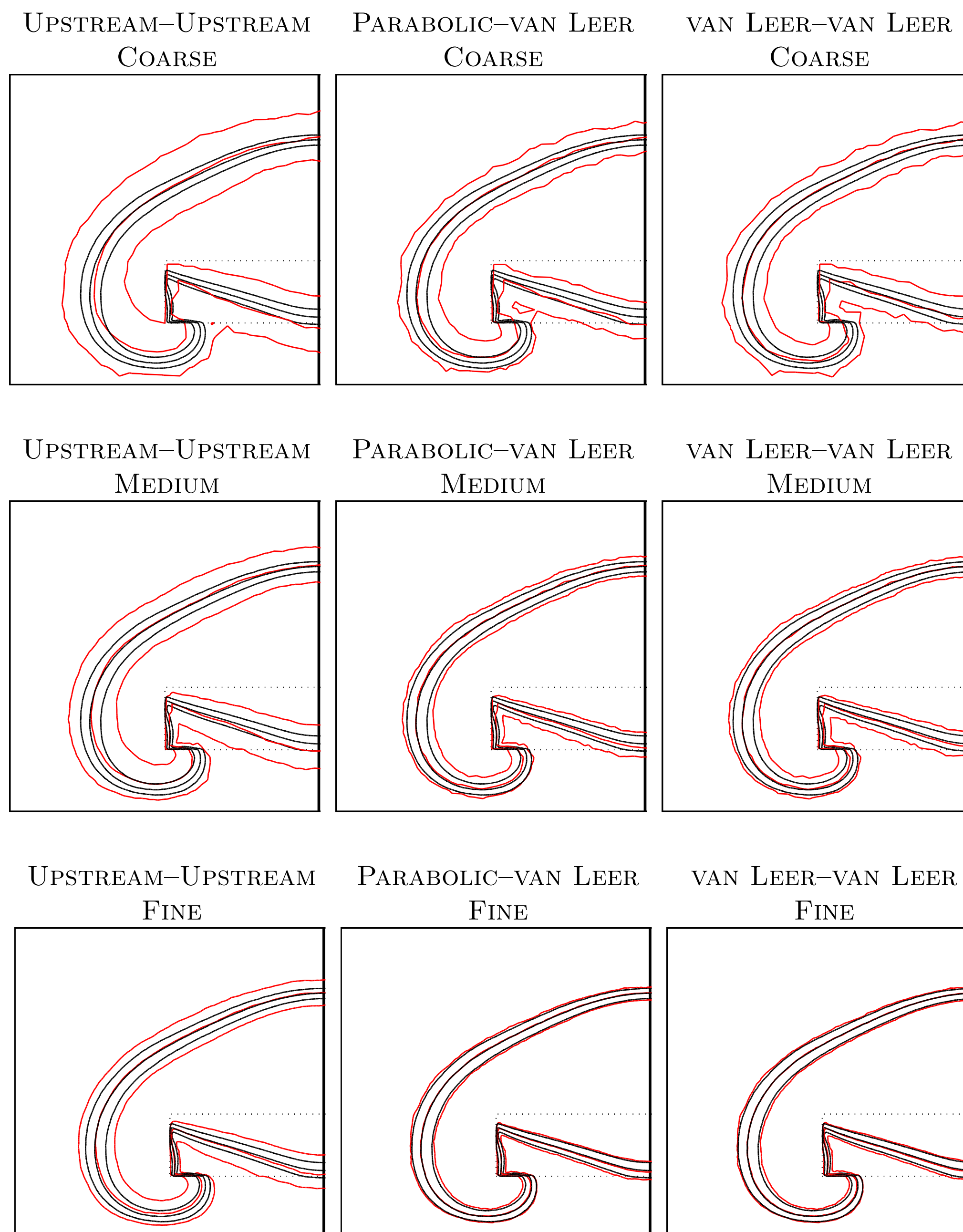


Figure 6.14: Comparison of concentration contours obtained using different spatial weighting schemes at $t = 48$ hours for the *unsaturated_transport* test case. The limiter used on the mobility and advection terms are written Mobility–Advection above each plot. The contours are the same as those in the reference solution Figure 6.15.

	mobility	advection	time steps	$\boldsymbol{F}$ evaluations	wall time (s)	$e_{\mathrm{mb}}^{\mathrm{w}}$	$e_{\mathrm{mb}}^{\mathrm{s}}$
coarse	upwind	upwind	171	1536	1.9	$1.3 \cdot 10^{-5}$	$2.1 \cdot 10^{-4}$
	van Leer	van Leer	312	4672	5.7	$1.2 \cdot 10^{-4}$	$1.3 \cdot 10^{-4}$
	van Leer	parabolic	305	5224	6.4	$1.2 \cdot 10^{-4}$	$5.2 \cdot 10^{-5}$
	parabolic	van Leer	190	2320	2.8	$5.4 \cdot 10^{-6}$	$2.5 \cdot 10^{-4}$
	parabolic	parabolic	191	1978	2.5	$9.6 \cdot 10^{-6}$	$2.1 \cdot 10^{-4}$
medium	upwind	upwind	273	3297	6.7	$6.4 \cdot 10^{-6}$	$7.4 \cdot 10^{-5}$
	van Leer	van Leer	523	7010	14.2	$3.1 \cdot 10^{-5}$	$9.1 \cdot 10^{-5}$
	van Leer	parabolic	517	8360	16.8	$3.0 \cdot 10^{-5}$	$3.7 \cdot 10^{-5}$
	parabolic	van Leer	317	3996	8.1	$4.1 \cdot 10^{-6}$	$7.4 \cdot 10^{-5}$
	parabolic	parabolic	325	4260	8.6	$5.8 \cdot 10^{-6}$	$6.2 \cdot 10^{-5}$
fine	upwind	upwind	408	6160	31.2	$1.9 \cdot 10^{-5}$	$8.4 \cdot 10^{-5}$
	van Leer	van Leer	685	11489	58.7	$3.5 \cdot 10^{-5}$	$7.7 \cdot 10^{-5}$
	van Leer	parabolic	716	11757	59.8	$4.7 \cdot 10^{-5}$	$1.2 \cdot 10^{-4}$
	parabolic	van Leer	526	8641	44.3	$2.3 \cdot 10^{-5}$	$7.8 \cdot 10^{-5}$
	parabolic	parabolic	525	8231	42.1	$3.2 \cdot 10^{-5}$	$1.3 \cdot 10^{-4}$

Table 6.14: Computational performance and mass balance results for the *unsaturated_transport* test case using different combinations of spatial weighting schemes for the PC formulation.

	mobility	advection	time steps	$\boldsymbol{F}$ evaluations	wall time (s)	$e_{\mathrm{mb}}^{\mathrm{w}}$	$e_{\mathrm{mb}}^{\mathrm{s}}$
coarse	upwind	upwind	157	1618	2.4	$1.2 \cdot 10^{-5}$	$3.6 \cdot 10^{-5}$
	van Leer	van Leer	289	4698	6.1	$2.3 \cdot 10^{-6}$	$1.2 \cdot 10^{-4}$
	van Leer	parabolic	303	4392	5.8	$2.4 \cdot 10^{-6}$	$9.9 \cdot 10^{-6}$
	parabolic	van Leer	182	2214	3.1	$2.2 \cdot 10^{-6}$	$1.4 \cdot 10^{-4}$
	parabolic	parabolic	183	1996	2.8	$5.1 \cdot 10^{-6}$	$3.9 \cdot 10^{-5}$
medium	upwind	upwind	258	3066	7.7	$5.1 \cdot 10^{-6}$	$1.9 \cdot 10^{-5}$
	van Leer	van Leer	498	5177	12.4	$2.9 \cdot 10^{-6}$	$3.9 \cdot 10^{-5}$
	van Leer	parabolic	618	11372	24.3	$2.4 \cdot 10^{-6}$	$8.0 \cdot 10^{-6}$
	parabolic	van Leer	306	4024	9.7	$2.0 \cdot 10^{-6}$	$1.1 \cdot 10^{-4}$
	parabolic	parabolic	307	3885	9.3	$2.4 \cdot 10^{-6}$	$1.6 \cdot 10^{-5}$
fine	upwind	upwind	450	7410	47.3	$3.5 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$
	van Leer	van Leer	802	11785	73.5	$5.0 \cdot 10^{-6}$	$3.2 \cdot 10^{-5}$
	van Leer	parabolic	739	10158	64.5	$8.8 \cdot 10^{-6}$	$2.2 \cdot 10^{-5}$
	parabolic	van Leer	463	7750	50.2	$8.5 \cdot 10^{-6}$	$3.7 \cdot 10^{-5}$
	parabolic	parabolic	474	7934	50.2	$8.2 \cdot 10^{-6}$	$3.2 \cdot 10^{-5}$

Table 6.15: Computational performance and mass balance results for the *unsaturated_transport* test case using different combinations of spatial weighting schemes for the MMPC formulation.

	edge length	nodes	edges	faces
coarse	0.05 m	1,054	2,820	5,760
medium	0.025 m	3,792	11,133	22,506
fine	0.0125 m	14,870	44,127	88,734

Table 6.16: Details of the triangular meshes used for testing the unsaturated flow and contaminant transport test case.

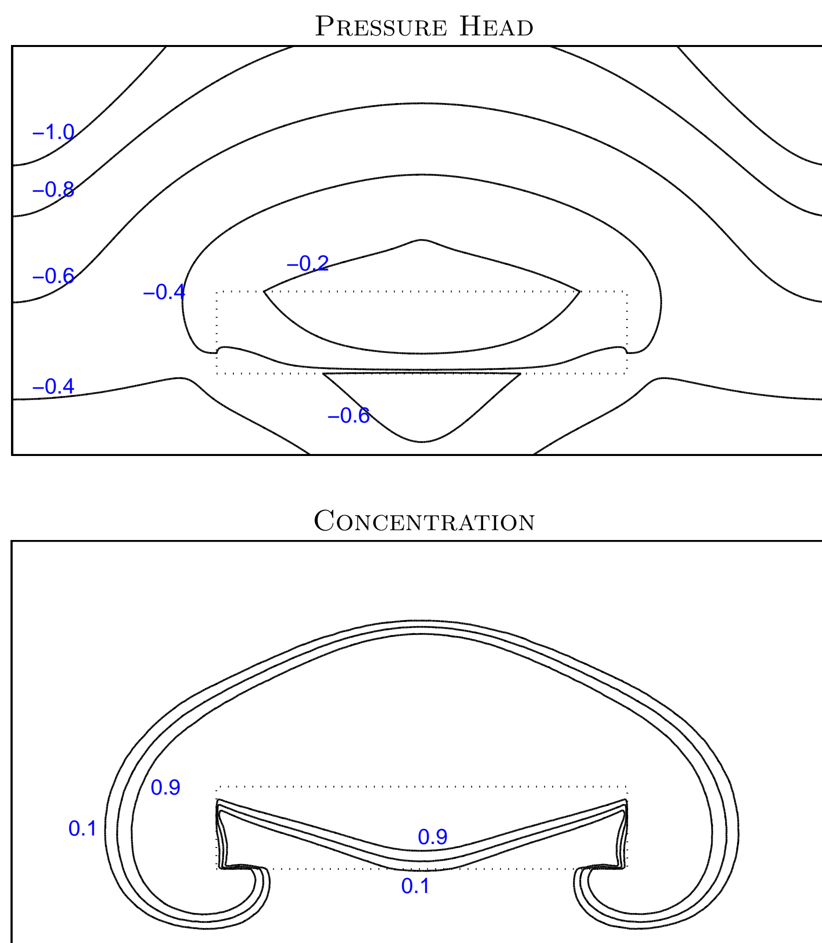


Figure 6.15: Reference solutions for the *water_table* test case after 48 hours. The solutions were found with averaging for the mobility term, and a van Leer limiter for the solute.

6.5 Transport Model: Zhang's Flow Tank Experiments

The flow tank experiments performed in the PhD work of Zhang (2000) included variably-saturated flow and transport, density-driven flow and time-dependent tidal boundary conditions, which makes them challenging to model numerically. The numerical simulations performed by Zhang (2000) were limited by the computational expense of modelling this complicated problem. A coarse 3,840-node quadrilateral mesh was used, and while the effect of tidal fluctuation of sea level was investigated experimentally, it was not included in the corresponding numerical models due to the computational expense it entails. As a result, while the simulations reproduced some gross features of the experiment, such as the extent and location of contaminant plumes, it was unable to capture more detailed phenomena such as the effect that tidal fluctuation has on the location of the sea water interface and the shape of the plume.

More detailed numerical investigation of the *tank_steady* and *tank_tidal_plume* test cases that incorporated tidal fluctuations was performed by Brovelli et al. (2007). They were able to more accurately model the evolution of the contaminant plume in the *tank_tidal_plume* test case using a mesh with 50,000 nodes. However, flow and transport in the unsaturated zone was not considered in their model. The analysis performed here is the first computational model of Zhang's experiments that includes both flow in the unsaturated zone and tidal variation.

Unstructured triangular meshes of varying resolutions, ranging from 3,883 nodes to 214,012 nodes (see Table 6.17), are used here to represent the domain in Figure 6.5(b). We note that the coarsest triangular mesh has a similar resolution to the quadrilateral mesh in Zhang's work.

Due to the large computational costs involved in performing these simulations, particularly those that involve tidal fluctuations, the GPU implemen-

mesh	1	2	3	4	5	6
nodes	3,883	11,277	23,149	53,026	92,356	214,012

Table 6.17: Details of the triangular meshes used to for the flow tank experiments.

tation is used for all tests. The PC formulation is used, given that the PC and MMPC formulations produce qualitatively consistent solutions. The van Leer limiter applied to the mobility term required considerably more computational effort relative to the parabolic limiter, as was observed for the *unsaturated_transport* test case in §6.4. However, when applied to the advection term, the van Leer limiter was slightly more efficient than the parabolic limiter. Hence, the flux limiter results presented here were computed using the parabolic limiter for the mobility term, and the van Leer limiter for the advection terms.

The ***tank_steady*** experiment: steady state location of the sea water interface

This experiment fixed the height of the water table at the inland boundary and the sea level at 463 mm and 439 mm respectively, then allowed the steady state water table level and sea water interface to develop. The steady state results from the experiment and the numerical solution by (Zhang, 2000) are illustrated in Figure 6.16(a). The corresponding reference solution, computed on mesh 5 with flux limiting, is illustrated in Figure 6.16(b).

The location of the water table in the numerical results of Zhang and the reference solution are in very good agreement with the experimental results. However, the difference between the fixed water table level and the inland and beach interfaces is very small, which makes it difficult to draw conclusions about the accuracy of the estimated water table. More interesting is the difference between the location of the $c = 0.5$ isochlor in the numerical and experimental results. The numerical results of Zhang (2000) match the foot of the sea water wedge quite well, slightly over-estimating the height of

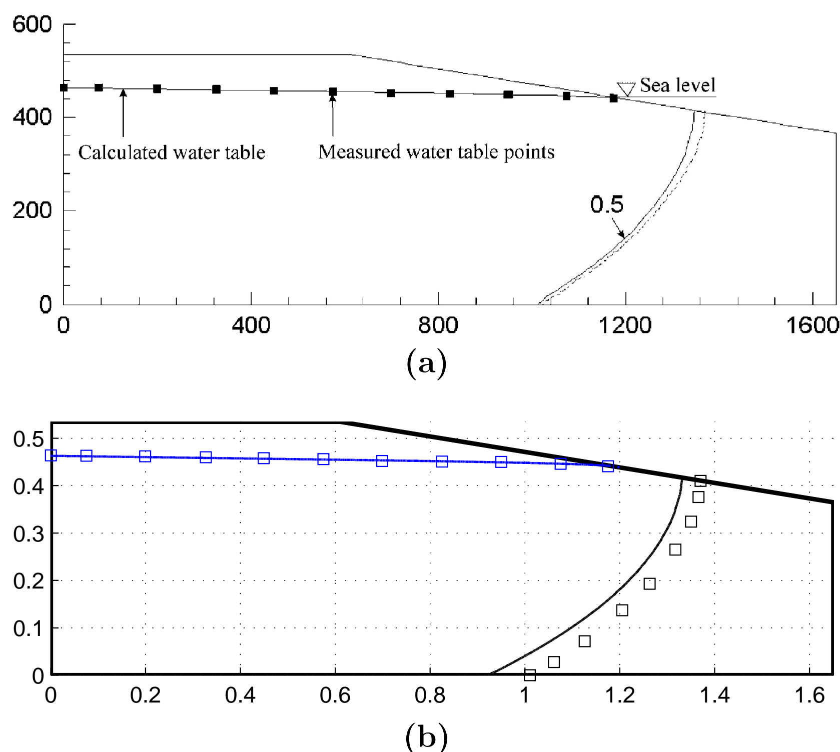


Figure 6.16: (a) Zhang's experimental and numerical results for the *tank_steady* flow tank experiment. The measured experimental isochlor is dashed, and the numerical isochlor solid. (b) The reference fine mesh solution, with the experimental water table level indicated by blue squares and the experimental isochlor marked with black squares.

the interface at the beach by approximately 20 mm. However, the reference solution in Figure 6.16(b) differs more significantly from the experimental results, over-estimating the location of the interface at both the toe and beach by about 50 mm.

However, the upstream solution on the coarsest mesh in Figure 6.17, which has the same resolution as that used in Zhang's numerical tests, provide better agreement with the experimental results, closely matching the numerical results of Zhang. Then, as the mesh is refined, both the upstream and flux limited solutions approach the reference solution. The salt transport is almost entirely advection-driven for this test case because the dispersion val-

ues are very small, small enough that setting them to zero has no qualitative effect on the solution. Such advection-driven processes suffer considerably from numerical diffusion, and require very fine meshes to obtain accurate results when using upstream weighting, or careful treatment of the advection terms using flux limiting (Neumann et al., 2011).

This suggests that the agreement for the sea water interface between the numerical and experimental results in the original work may have been a coincidence, caused by numerical diffusion on the coarse mesh. While this may explain the disparity between the numerical result in Zhang (2000) and the reference solution, it does not address why the reference solution differs from the experimental result.

The difference between the reference solution and the experimental results can't be explained by inaccuracies in the measured parameters. To determine this, a sensitivity analysis of the hydraulic conductivity K and dispersion values α_L and α_T was performed. Varying the hydraulic conductivity in the range 3 mm/s to 5 mm/s had minimal effect on the location of the interface[4]. Additionally, dispersion is so small that it has almost no effect on the solution, and increasing its value causes excessive smearing of the interface, which does not agree with the sharp interface observed in the experimental results and the reference solution.

Another hypothesis is that the experiment was not run until it had reached the full steady state. However, in later simulations based on the same flow tank the numerical results from the methods in this work over-estimate the location of the foot of the interface. This suggests that the difference is more likely due to inaccuracies inherent in mapping the three-dimensional experimental domain to the two-dimensional computational domain, and/or in the treatment of the boundary condition at the sea-beach interface.

However, if the reference solution is taken to be the most accurate given

[4]This is a much larger range than the estimated range of between 3.9 mm/s to 4.2 mm/s measured by Zhang et al. (2004) and others (Ataie-Ashtiani, 1997) for the porous medium used in the experiments.

the parameters and model at hand, it is apparent that the flux limited solutions converge towards the solution much faster than the upstream results. The effect of numerical dispersion is particularly evident here, given that the upstream solution on a relatively fine 23,149 node mesh significantly under-estimates the location of the $c = 0.5$ isochlor. However, the results in Table 6.18 show that solutions obtained using flux limiting require considerably more computational work, between two to three times, than the upstream solutions on the same mesh. The flux limited solution on mesh 2 is more accurate than the upstream weighted solution on the finer mesh 3, however the computational benefits of using flux limiting on a coarser mesh are less obvious in this case.

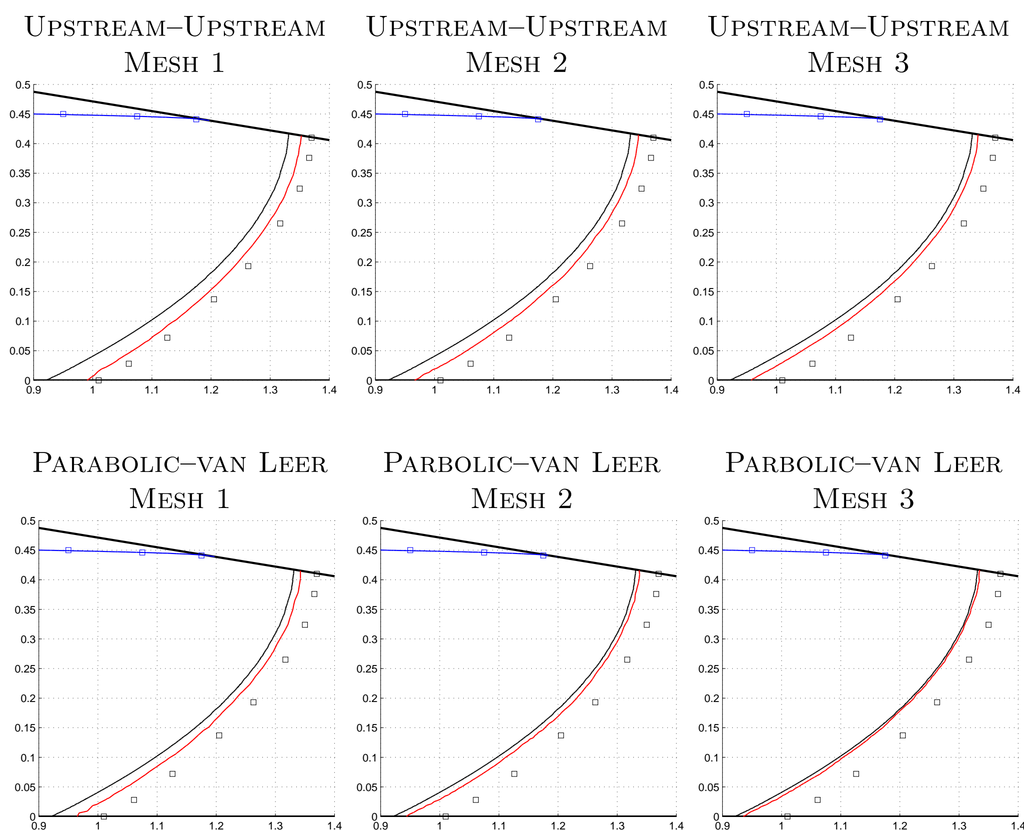


Figure 6.17: Steady state solutions for the *tank_steady* test case.

		upstream	flux limited
Mesh 1	$\boldsymbol{F}$ evaluations	3515	9057
	wall time (s)	5.8	16.3
Mesh 2	$\boldsymbol{F}$ evaluations	3802	8672
	wall time (s)	16.4	41.2
Mesh 3	$\boldsymbol{F}$ evaluations	4154	11825
	wall time (s)	31.2	104.5

Table 6.18: Computational performance metrics for the *tank_steady* test case using upstream weighting and flux limiting as mesh resolution increases.

The *tank_plume* experiment: the evolution of a dense contaminant plume with stationary sea level

This experiment was performed in two phases. First, the height of the tide and inland fresh water head were held fixed at the values prescribed for the *tank_steady* experiment, and the steady state sea water interface was allowed to form. Then, contaminated water with density 1015.7 g/L was injected over the 180 mm wide injection site above the beach in Figure 6.5(b) at a constant rate of $1.4 \cdot 10^{-4}$ m/s for 80 minutes.

The location of the plume of contaminated water from the injection site in the experimental results is shown in Figure 6.20 at times of 30, 40, 50 and 60 minutes, where $t = 0$ minutes corresponds to the introduction of the contaminant. The plume travels downward and towards the shore, with flow of contaminant into the sea, below the water table and above the sea water interface, established between 50 and 60 minutes. The higher density of the plume relative to the fresh water into which it is injected has a significant impact on the shape of the plume. The plume develops a “heel” that drops towards the bottom of the aquifer, and draws the plume closer to the sea water interface, as labelled at Figure 6.20(c). The plume also exhibits small irregular fingers due to instabilities introduced by the flow of the dense contaminated fluid above the fresh water, particularly along the leading edge of the plume.

To investigate the effect of upstream weighting and flux limiting on the shape of the contaminant plume, test runs were performed with mesh 2 and mesh 4. The results, illustrated in Figure 6.19, show the $0.5\,c_0$ contour, where c_0 is the concentration of salt in the injected contaminant (equivalent to $c = 0.345$), alongside the equivalent contours from the experimental results[5].

The numerical results agree well with the experimental results at the inland side of the plume, with the diffuse region around the pocket (illustrated in Figure 6.20(c)), reproduced in the upstream solution in Figure 6.21. However, the location of the flat leading front and the depth of the contaminant plume are under-estimated at later times in each numerical solution.

The most obvious difference between the upstream and flux limited solutions is the unstable fingers observed in the flux limited solution along the leading edge of the plume. The fingers become more pronounced as the resolution of the mesh is increased, with the onset of fingering evident at 30 minutes for mesh 4 in Figure 6.19(b). The fingers develop in the same place, along the leading edge of the plume, for both the experimental and flux limited solutions. However, the numerical fingers grow in size as the plume descends, becoming significantly more pronounced than those in the experimental results after 60 minutes.

The formation of density-dependent fingers is a complicated process that can be initiated by a number of different physical phenomena, including uneven injection of the contaminant at the boundary and small heterogeneities in the porous medium (Schincariol et al., 1994). No such effects are included in the modelling presented here. Instead, the fingering in the numerical results is initiated by small perturbations in the numerical solution. To reproduce experimental fingering patterns numerically, it is necessary to explicitly model the cause for the onset of the instabilities in the experiments (Schincariol et al., 1994, Brovelli et al., 2007), which is beyond the scope of this thesis.

[5] The open source program G3Data `http://www.frantz.fi/software/g3data.php` was used to digitise the experimental results.

While it is not possible to use the numerical results to draw conclusions about the exact nature of the fingering patterns in the experimental observations, it is interesting to note the effect that the upstream weighting and flux limiting schemes have on the development of fingering in the numerical results. The numerical diffusion introduced by upstream weighting may smooth over the numerical perturbations that cause the initial onset of fingering. Indeed, sensitivity analysis tests (not illustrated here) showed that fingering was not present in flux limited solutions for larger diffusion or dispersion parameters. Furthermore, this suggestion is supported by the observation that the upstream solutions on the very fine 214,012 node mesh 6, on which the effect of numerical diffusion is reduced, exhibit fingering on the leading edge in Figure 6.22. Hence, without attempting to include the cause for the fingering in the numerical model, we choose the upstream weighted solution on a fine mesh as the most representative of the experimental results. The upstream weighted solution computed on mesh 3 is shown in Figure 6.21, and is in good agreement with the experimental solutions in Figure 6.20.

Another significant feature of the plume in the experimental results is the diffuse zone that develops in the unsaturated region between the main body of the plume and the beach at 30 minutes in Figure 6.20. As the plume migrates down through the saturated region towards the beach, the extent of the diffuse region becomes larger. The diffuse region has not been reproduced in numerical solutions, both in past investigations (Zhang, 2000, Volker et al., 2002, Brovelli et al., 2007), and in the simulations presented here.

In their analysis of the experimental and numerical results from Zhang's thesis, Volker et al. (2002) suggest that the diffuse region arises due to mixing in the unsaturated zone at the beach. However, the diffusion region is not captured in either the original numerical models where the unsaturated region was modeled explicitly (Zhang, 2000, Volker et al., 2002), nor in this work. A possible explanation for the formation of the diffuse region near the beach may be the method by which the contaminant was injected into the domain. The contaminant was injected through rows of small holes, which are not

directly adjacent to the edge of the tank. This might suggest that a three-dimensional model, which explicitly includes the location of the injection points may be required to explain the development of the diffuse region at the beach.

Finally, we investigate how the time step size selection in IDA is affected by the change in the boundary condition at $t = 0$ minutes at the contaminant injection site. The representative solution in Figure 6.21 required a wall time of 225 seconds, and took 721 time steps to compute on the 23,149 node mesh. The time step before and after the commencement of contaminant injection at $t = 0$ min is shown in Figure 6.18, which illustrates the effectiveness of the automatic time step size selection by IDA. The time step decreased when the contaminant was first injected, to capture the initial infiltration of fluid through the unsaturated region, before increasing as the flow in the unsaturated region became steady.

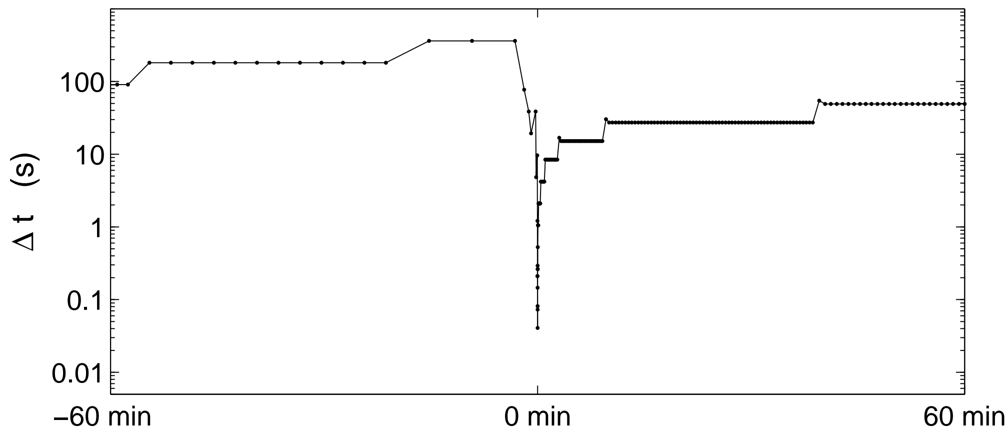
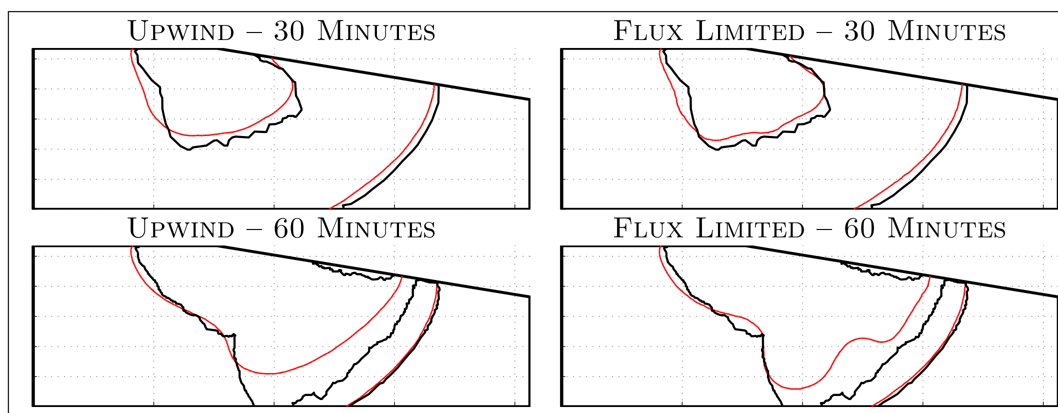
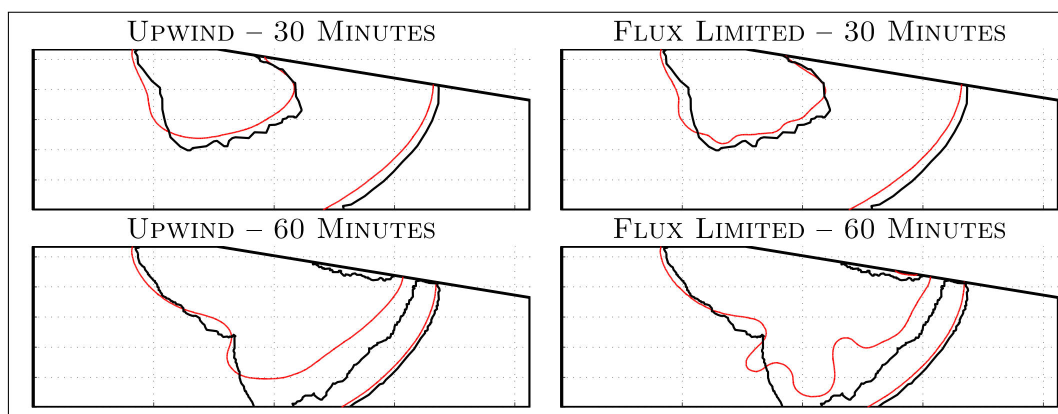


Figure 6.18: Time step size before and after the injection of contaminant at $t = 0$ min in the *tank_plume* test case.

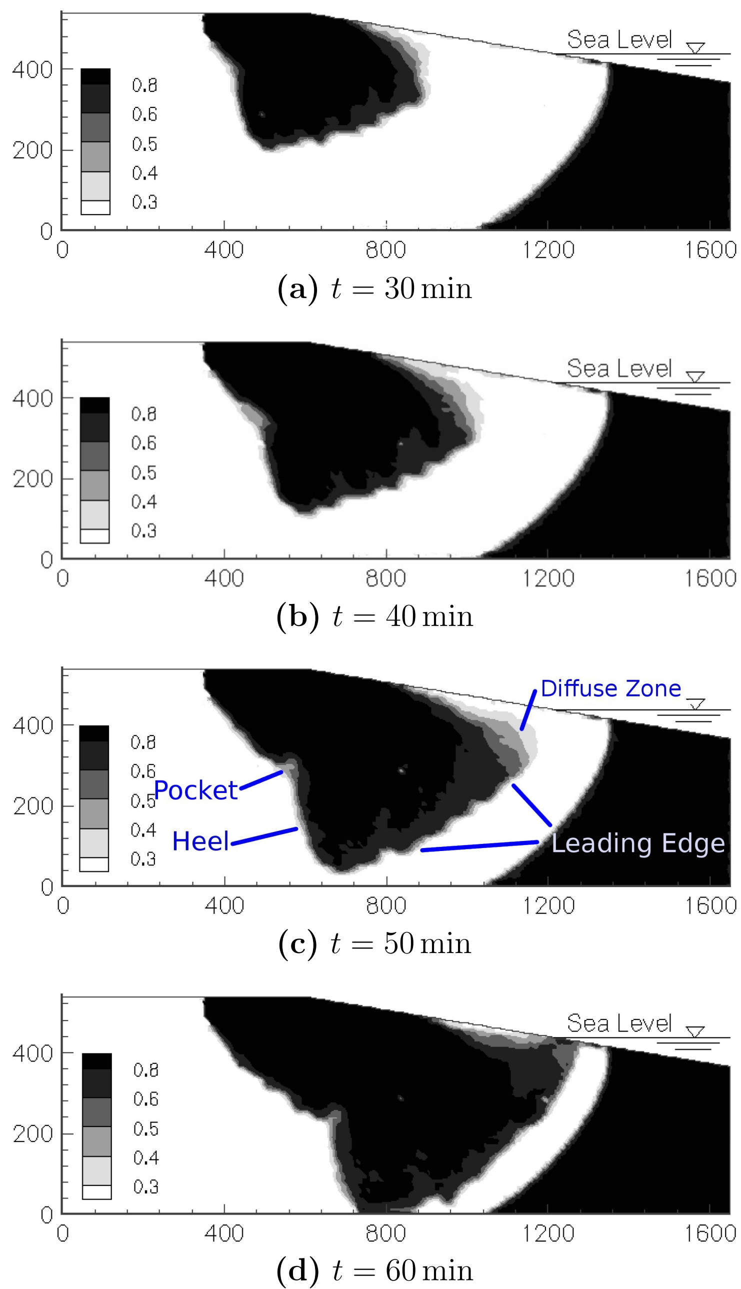


(a) Mesh 2



(b) Mesh 4

Figure 6.19: Comparison between experimental and numerical results (upstream and flux limited) on meshes 2 and 4 for the *tank_plume* experiment. The numerical results are coloured red, and the experimental black. Contours are at $c_0 = 0.5$, where $c_0 = 0.689$ is the concentration of salt at the contaminant source.



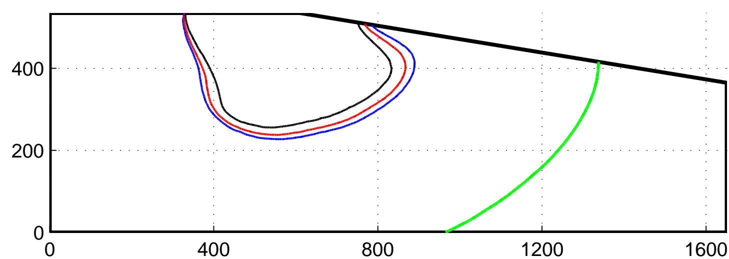
(a) $t = 30\,\text{min}$

(b) $t = 40\,\text{min}$

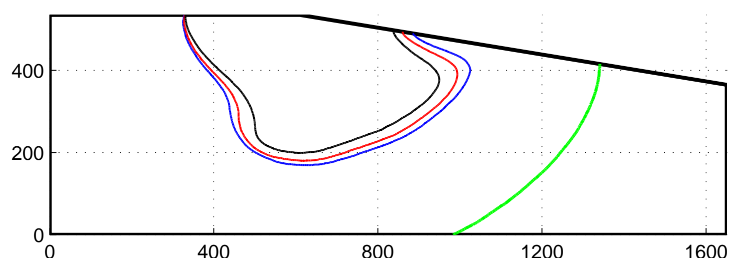
(c) $t = 50\,\text{min}$

(d) $t = 60\,\text{min}$

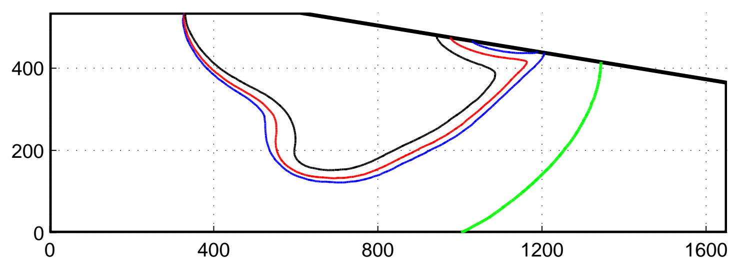
Figure 6.20: Experimental results for the *tank_plume* test case, where contaminant is first injected at $t = 0\,\text{min}$.



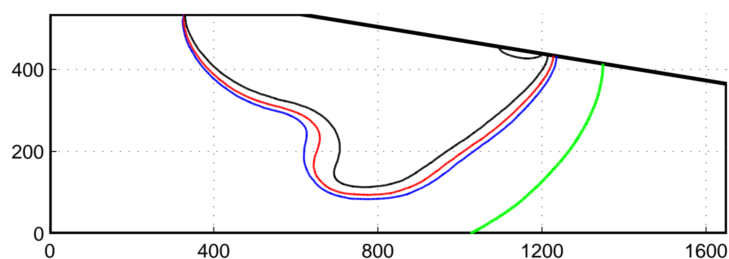
(a) $t = 30\,\text{min}$



(b) $t = 40\,\text{min}$



(c) $t = 50\,\text{min}$



(d) $t = 60\,\text{min}$

Figure 6.21: Solution contours for the *tank_plume* test case determined using upstream weighting and Mesh 3. The contours are for $0.3\,c_0$ (black), $0.5\,c_0$ (red) and $0.8\,c_0$ (blue), where c_0 is the concentration of salt in the injected contaminant. The location of the sea water interface at $c = 0.5$ is marked in green.

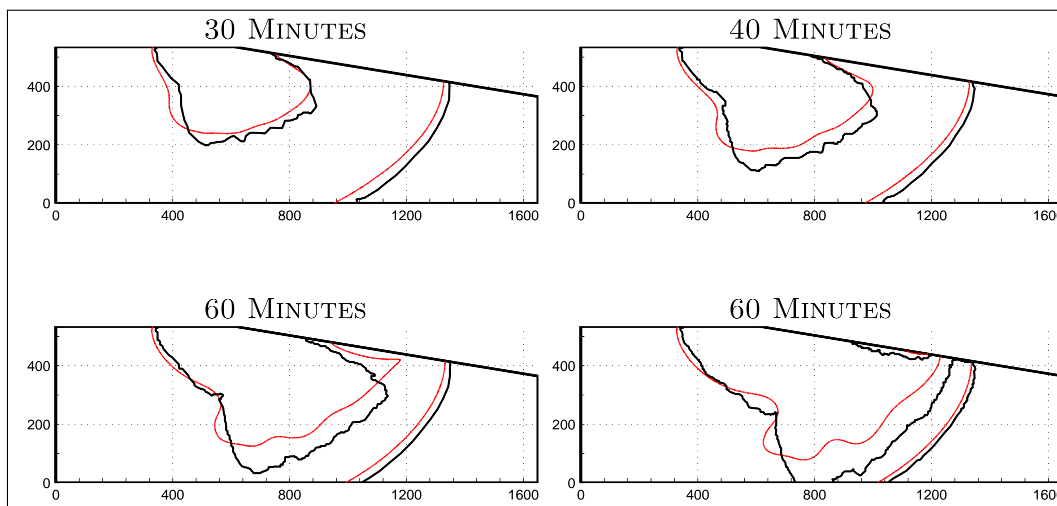


Figure 6.22: The location of the $c_0 = 0.5$ isochlor for the *tank_plume* test case using the finest mesh (mesh 6 with 214,012 nodes) and upstream weighting. The equivalent ischolors measured in the experiment are shown in black.

The *tank_tidal_plume* experiment: the evolution of a dense contaminant plume under tidal forcing

The effect of tidal variation on the evolution of a dense contaminant plume was investigated in this experiment. The experiment proceeded identically to the *tank_plume* test case, however the height of the sea level was varied according to the tidal boundary condition specified in Figure 6.4 when the contaminant was first introduced at $t = 0$ minutes.

The location of the plume in the experimental results for both the *tank_plume* and *tank_tidal_plume* experiments is illustrated in Figure 6.24. Zhang (2000) observed that there appear to be significant qualitative differences between the shape of the plume in each experiment, particularly on the inland side. However, both plumes exhibit a similar extent of intrusion towards the sea water interface and the beach, with the plume reaching the beach between 50 and 60 minutes in both experiments. This observation was used to justify performing numerical experiments without tidal fluctuation to investigate the

travel time of the plume in the seawards direction. However, the considerable difference in the shape of the plume under tidal forcing, particularly on the inland side, suggests that the tide plays a significant role in the evolution of the plume, warranting further investigation.

Figure 6.25 shows the $c_0 = 0.5$ isochlors from numerical simulations of the *tank_plume* and *tank_tidal_plume*, computed using upstream weighting on mesh 3. The numerical results reproduce the shape and location of the plumes and sea water interface well, though the numerical results slightly under-predict both the vertical and horizontal intrusion of the plume towards the sea water interface at later times in both experiments.

There are two distinct flow regimes in the tidal cycle. First, as the tide lowers, flow from inland towards the sea increases, carrying the plume downwards and towards the sea. Conversely, as the tide rises, flow of fresh water from inland towards the sea decreases, then reverses when the level of the tide rises above the height of the inland water table. When the flow reverses, the inland side of the plume below the injection site moves inland, which forms a body of contaminant that is separated from the lower part of the plume. The first high tide occurs at 30 minutes, at which point the upper inland body of contaminant is evident in both the experimental and numerical results in Figure 6.24(a) and Figure 6.25(a) respectively. Then, as the tide lowers to low tide at 50 minutes, the inland body is carried down and towards the beach as the flow from inland to the sea increases in Figure 6.24(c) and Figure 6.25(c).

Figure 6.23 shows a comparison of the solution obtained using upstream weighting and flux limiting on meshes 1 and 3. It is interesting to note that the flux limited solution on the fine mesh does not exhibit the extent of fingering observed in the *tank_plume* experiment. This is likely due to the action of the tide dispersing the leading edge of the plume, which smooths out the numerical perturbations that give rise to fingers in the *tank_plume* experiment. The flux limited solution on mesh 1 reproduces the experimental results better than the upstream solution on mesh 3. The flux limited solution

on the fine mesh reproduces the shape of the inland side of the plume very well, however it has an exaggerated upwelling of fresh water half way along the bottom of the plume, an effect that was also observed in the numerical results of Brovelli et al. (2007). For this reason the flux limited solution on the coarse mesh is chosen as the best fit for this test case, and the full solutions are plotted in Figure 6.27.

IDA takes significantly more time steps in the *tank_tidal_plume* simulations due to the tidal boundary condition, which results in considerably longer wall times than the *tank_plume* simulations. For example, the upstream solution on mesh 3 took 2,760 seconds to compute, compared to 225 seconds for the equivalent *tank_plume* simulation. On the other hand, the flux limited solution computed on mesh 1 took only 186 seconds to compute, which is a speedup of 14.8 times over the upstream solution. This illustrates the benefits of using flux limiting to obtain accurate solutions on coarse meshes.

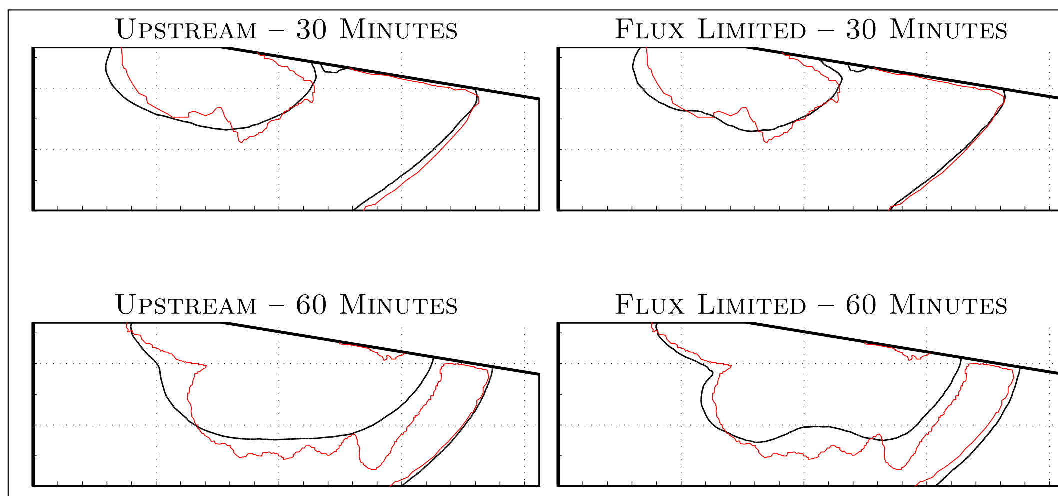
Summary of Zhang's experiments

The *tank_plume* and *tank_tidal_plume* experiments investigated here are both very challenging to model numerically. Major features of the plumes observed in the experiments, such as the diffuse region near the shore and the fingering in the dense plumes have yet to be reproduced in numerical investigations (Zhang, 2000, Volker et al., 2002, Brovelli et al., 2007). While it is beyond the scope of this thesis to fully investigate the test cases, we note that the observations presented here suggest that the following ideas will have to be studied in more detail to better understand the dynamics of the system.

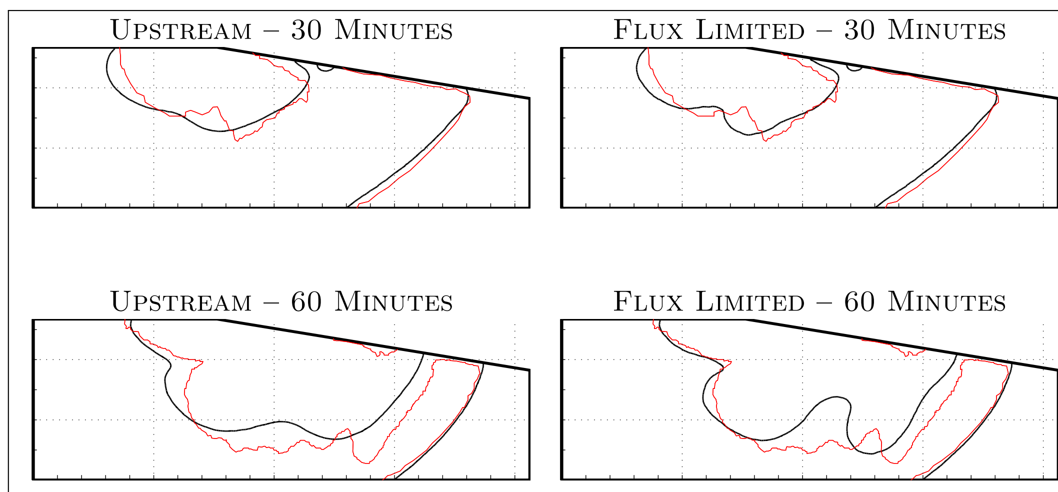
- It may be necessary to model the transient location of the seepage face at the beach, which is not possible using the usual approach in IDA (see Appendix D). Explicitly modelling a transient seepage face has been shown to more accurately capture the shape of contaminant plumes near the beach in similar experiments (Boufadel et al., 2011).

- The cause of the onset of density-dependent fingering, and the reason that the fingers do not grow large over time, needs to be explicitly modelled. This would entail investigating the effect of small heterogeneities in the porous medium and the effect of uneven injection of the contaminant.

- Full three-dimensional simulation should be performed to better understand the development of the diffuse region near the shore line, and gain further insight into the development of density-dependent fingers in three dimensions.

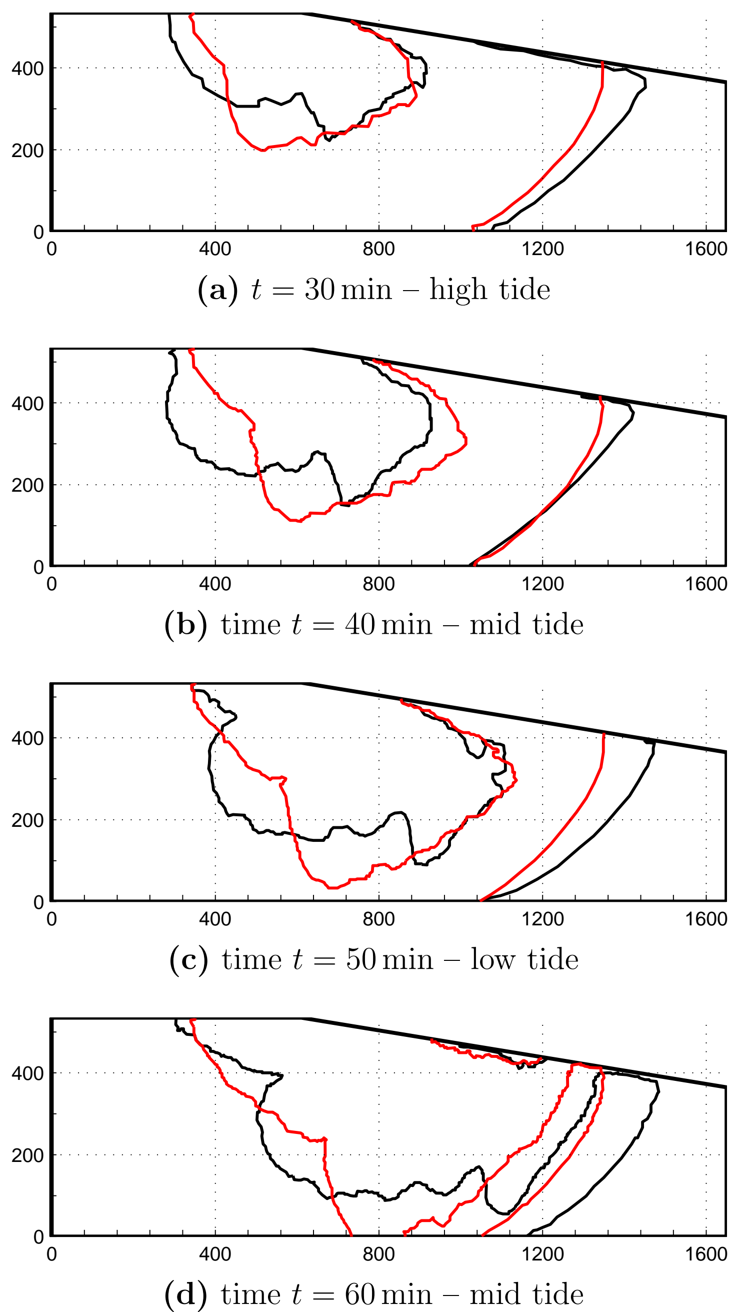


(a) Mesh 1



(b) Mesh 3

Figure 6.23: Comparison between experimental and numerical results (upstream and flux limited) on meshes 1 and 3 for the *tank_tidal_plume* experiment. The numerical results are coloured black, and the experimental red. Contours are at $c_0 = 0.5$, where c_0 is the concentration of salt in the contaminant source.



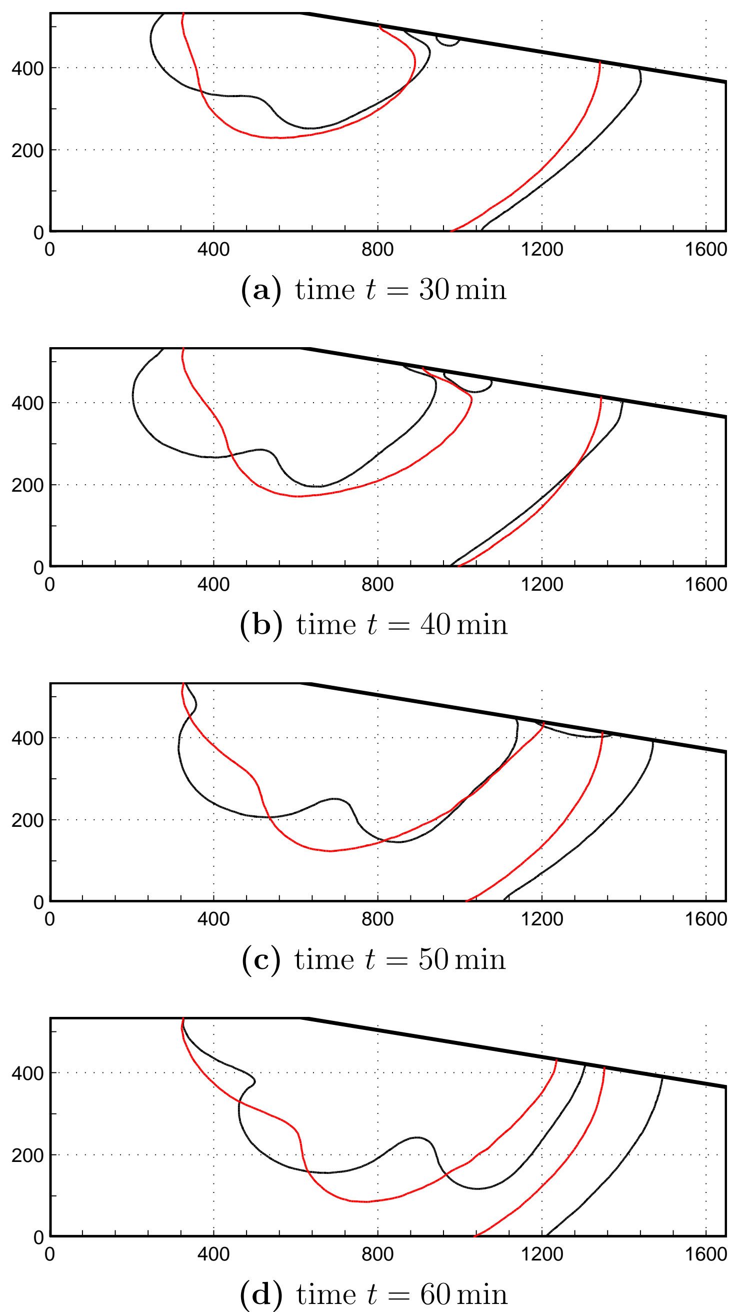
(a) $t = 30\,\text{min}$ – high tide

(b) time $t = 40\,\text{min}$ – mid tide

(c) time $t = 50\,\text{min}$ – low tide

(d) time $t = 60\,\text{min}$ – mid tide

Figure 6.24: Comparison of experimental results for the contaminant plume subject to: tidal sea level (black); and stationary sea level (red).



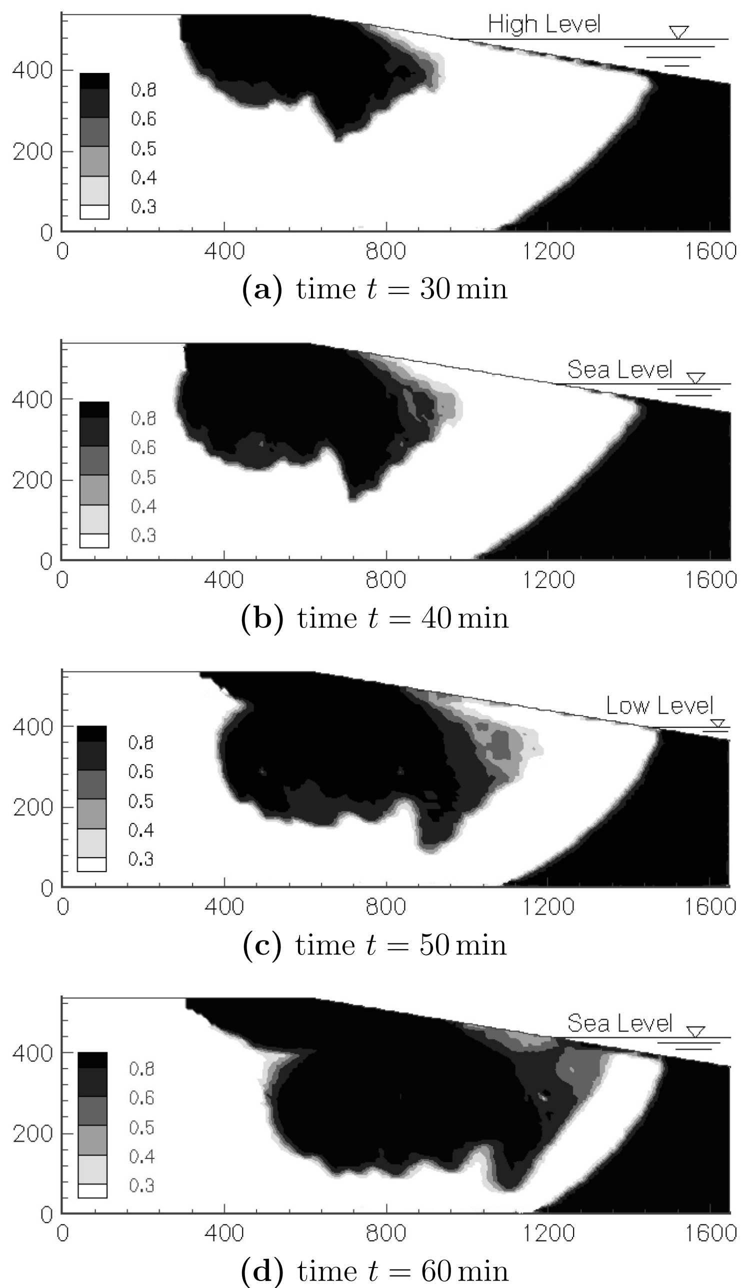
(a) time $t = 30\,\text{min}$

(b) time $t = 40\,\text{min}$

(c) time $t = 50\,\text{min}$

(d) time $t = 60\,\text{min}$

Figure 6.25: Comparison of numerical results for the contaminant plume subject to: tidal sea level (black); and stationary sea level (red).



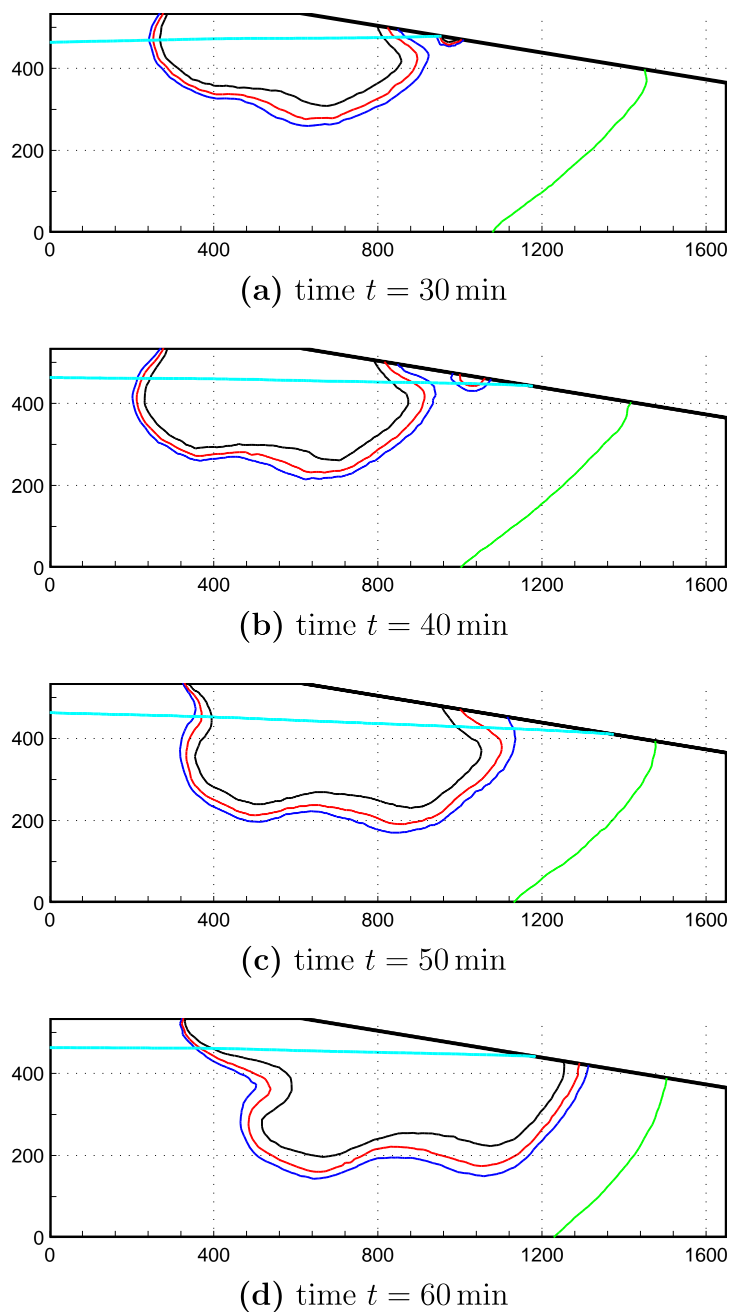
(a) time $t = 30\,\mathrm{min}$

(b) time $t = 40\,\mathrm{min}$

(c) time $t = 50\,\mathrm{min}$

(d) time $t = 60\,\mathrm{min}$

Figure 6.26: Experimental results for the *tank_tidal_plume* test case.



(a) time $t = 30\,\text{min}$

(b) time $t = 40\,\text{min}$

(c) time $t = 50\,\text{min}$

(d) time $t = 60\,\text{min}$

Figure 6.27: Numerical results using flux limiting on the coarse mesh (mesh 1 with 3,883 nodes) for the *tank_tidal_plume* test case. The contours are for $0.3\,c_0$ (black), $0.5\,c_0$ (red) and $0.8\,c_0$ (blue), where c_0 is the concentration of salt in the injected contaminant. The location of the sea water interface at $c = 0.5$ is marked in green.

6.6 Transport Model: Heap Leaching Simulation

This test case was formulated to validate the flux limited weighting scheme in three dimensions. To that ends, very small dispersion and diffusion parameters are prescribed to give advection-dominated transport. In Figure 6.28 and Figure 6.29, isosurfaces for the concentration are shown at 9, 45 and 81 days of the simulation on the 47,264 node reference mesh. Over the first 9 days the contaminant in the heap is leached downwards, into the unsaturated layer above the aquifer and directly below the heap. The contaminant plume then enters the saturated region, where its movement is affected by the source term.

In the absence of the source term, the fluid flowing from the heap would flow directly towards the hydrostatic boundary (on the $x = 0$ plane), as this is the only boundary over which fluid can leave the domain. However, the source term is located between the heap and the hydrostatic boundary. This pushes fluid that leaches from the heap away from the hydrostatic boundary, forcing it to flow out of the hydrostatic boundary near the $y = 4.1\,\mathrm{m}$ boundary. The concentration isosurfaces in Figures 6.28 and 6.29 show that the contaminant plume is carried in this flow, towards the boundary at $y = 4.1$, where it then moves towards the hydrostatic boundary.

Mesh convergence tests were performed to determine a reference solution. As was the case for the simulations based on Zhang's flow tank experiments the PC formulation is used for these tests because the PC and MMPC methods produce qualitatively identical results. The Van Leer limiter is used for the mobility term, because the parabolic limiter failed to converge or required many iterations on some of the tests. Of the limiters applied to the advection term, the parabolic limiter was slightly more efficient, and is used in these numerical experiments. In Figure 6.30(a) the concentration contours on a slice at $z = -0.5\,\mathrm{m}$ are illustrated for the upstream weighting and flux limiting schemes. The solutions are in reasonable agreement, except at the

$c = 0.25$ contour, which the upstream solution underestimates significantly. The flux-limited solution on the reference mesh is chosen as the reference solution, illustrated in Figure 6.30(b), for these tests because both the upstream and flux limited solutions on coarser meshes converge to this solution, as will be shown below.

Three meshes with between 7,932 and 28,788 nodes, summarised in Table 6.19, were used to compare solutions computed using upstream weighting and flux limiting against the reference solution. Figure 6.31 shows concentration contours on the plane $z = -0.5\,\mathrm{m}$ computed on each mesh with upstream weighting. The solutions converge slowly towards the reference solution, although it is necessary to include the upstream solution on the reference mesh in Figure 6.30(a) to verify this for the $c = 0.25$ contour. To accurately determine the profile of the contaminant plume, particularly for the high-concentration region, would require a mesh with much higher resolution than the reference mesh.

The corresponding solutions computed with flux limiting, shown in Figure 6.32, converge quickly to the reference solution, and the solution on the medium mesh accurately reproduces the profile of the plume. Even on the coarse mesh, flux limiting accurately captures that shape of the contaminant plume: indeed, it is more accurate than upstream weighing on the reference mesh.

Metrics for computational performance of each of the spatial weighting schemes on the different meshes are summarised in Table 6.20. Both flux limiting and upstream weighting require a comparable number of residual evaluations and wall times on the coarse and medium meshes, and on the fine and reference meshes upstream weighting has wall times between 1.4 and 2.3 times lower than flux limiting. Overall, flux limiting requires much less work to reproduce the shape of the plume: the coarse flux limited solution is more accurate than the fine mesh solution with upstream weighting, while requiring 18 times less wall time to solution.

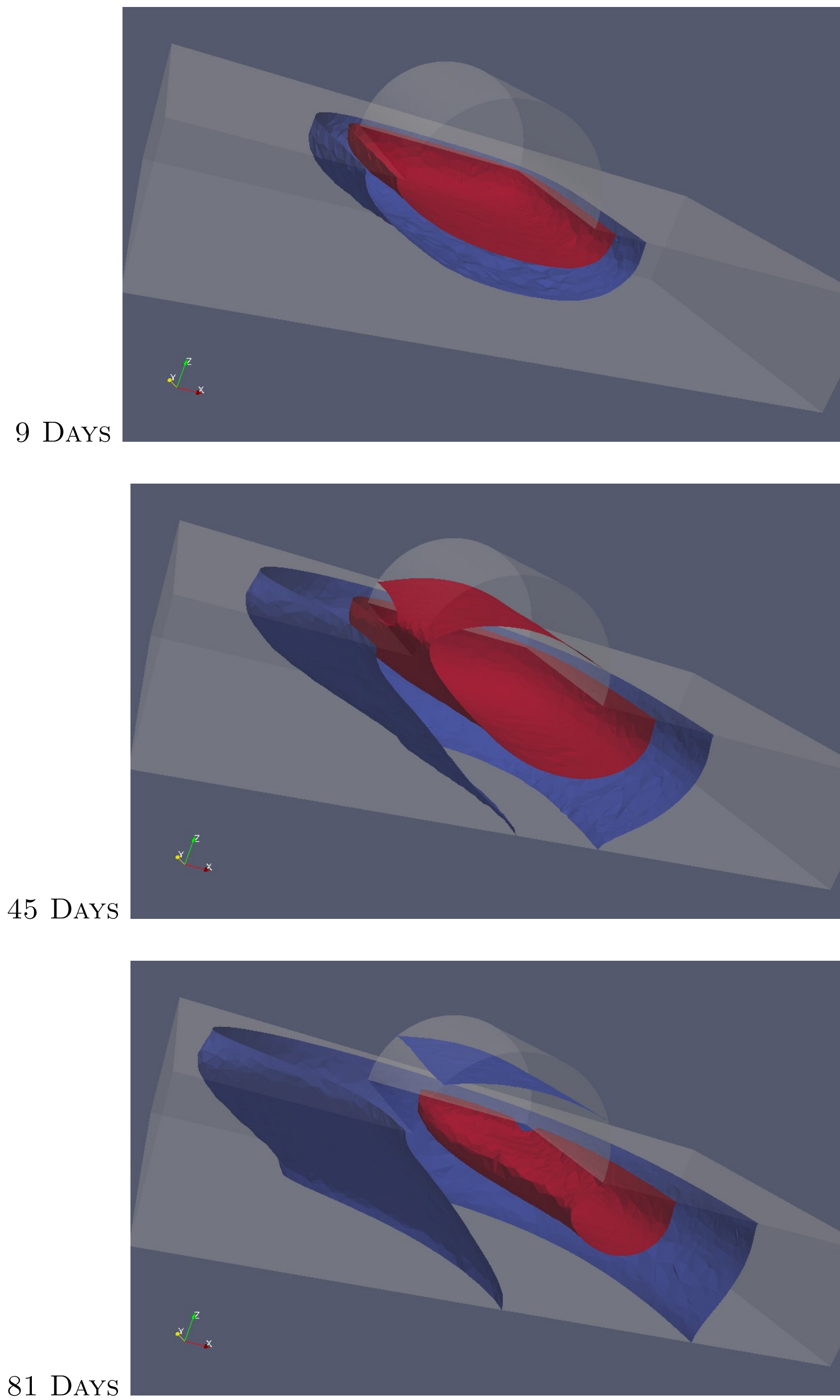


Figure 6.28: Front view of concentration isosurfaces for the reference solution of the *heap_leaching* test case. The blue surface at c =0.05 shows the extent of the plume, and the red surface at c =0.3 shows high-concentration centre of the plume.

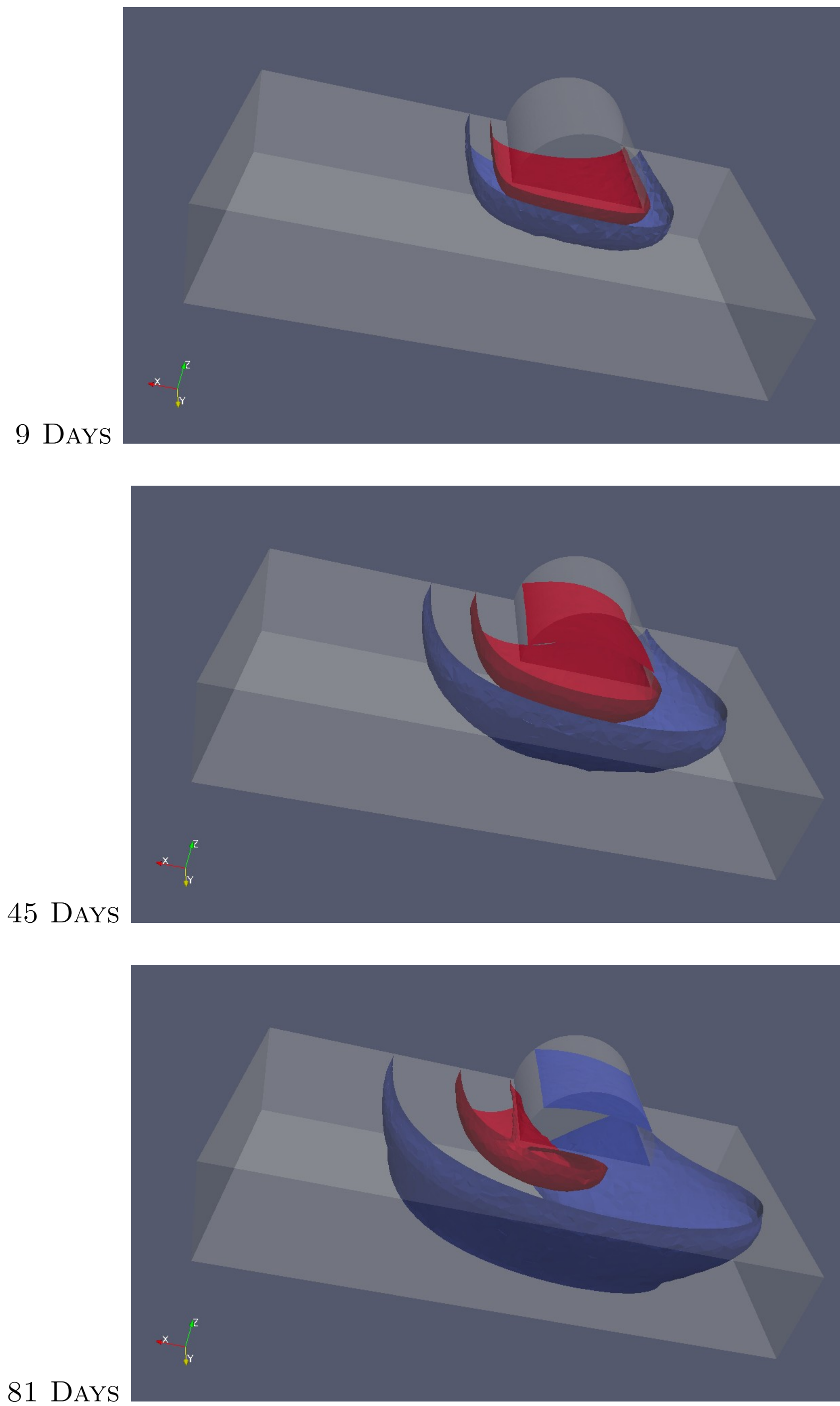


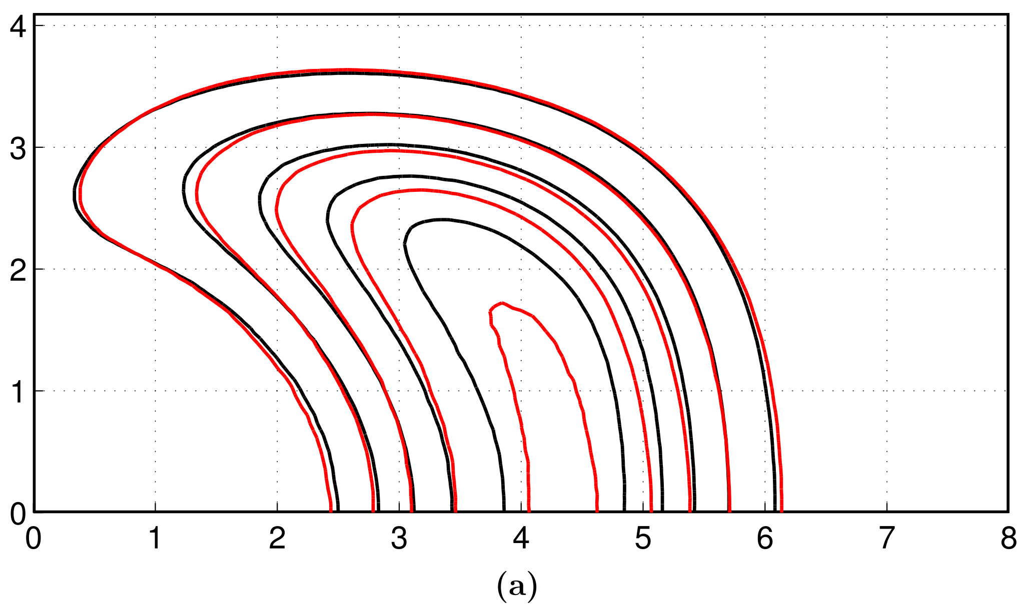
Figure 6.29: Back view of concentration isosurfaces for the reference solution of the *heap_leaching* test case. The blue surface at c =0.05 shows the extent of the plume, and the red surface at c =0.3 shows high-concentration centre of the plume.

	edge length	nodes
coarse	0.20 m	7,932
medium	0.15 m	13,322
fine	0.10 m	28,788
reference	0.075 m	47,264

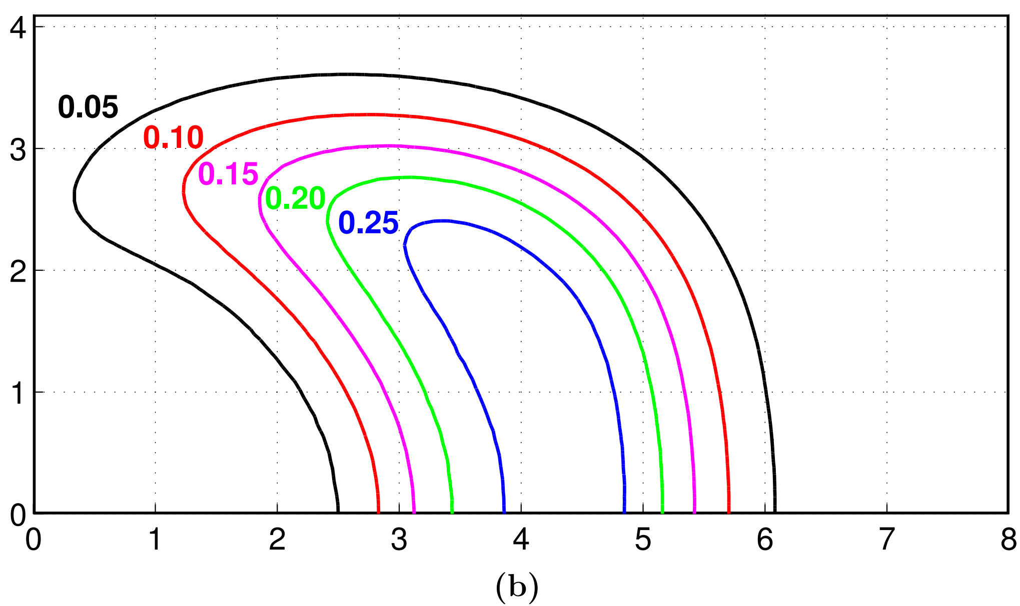
Table 6.19: Details of the tetrahedral meshes used for testing the *heap_leaching* test case.

		upstream	flux limited
Coarse	$\boldsymbol{F}$ evaluations	4046	4518
	wall time (s)	34.1	37.3
	time steps	102	108
Medium	$\boldsymbol{F}$ evaluations	4404	4215
	wall time (s)	55.1	52.5
	time steps	117	105
Fine	$\boldsymbol{F}$ evaluations	7130	14334
	wall time (s)	177.4	339.8
	time steps	142	205
Reference	$\boldsymbol{F}$ evaluations	17860	25256
	wall time (s)	670.4	940.2
	time steps	294	338

Table 6.20: Computational performance metrics for the *heap_leaching* test case using upstream weighting and flux limiting as mesh resolution increases.



(a)



(b)

Figure 6.30: Concentration contours for the *heap_leaching* test case computed on a slice of the reference mesh at $z = -0.5\,\mathrm{m}$ at 81 days. In (a) solutions computed using flux limiting (black) and upstream weighting (red) are illustrated. In (b) the same contours are drawn and labelled for the flux limited solution, which is used as the reference solution in these testss.

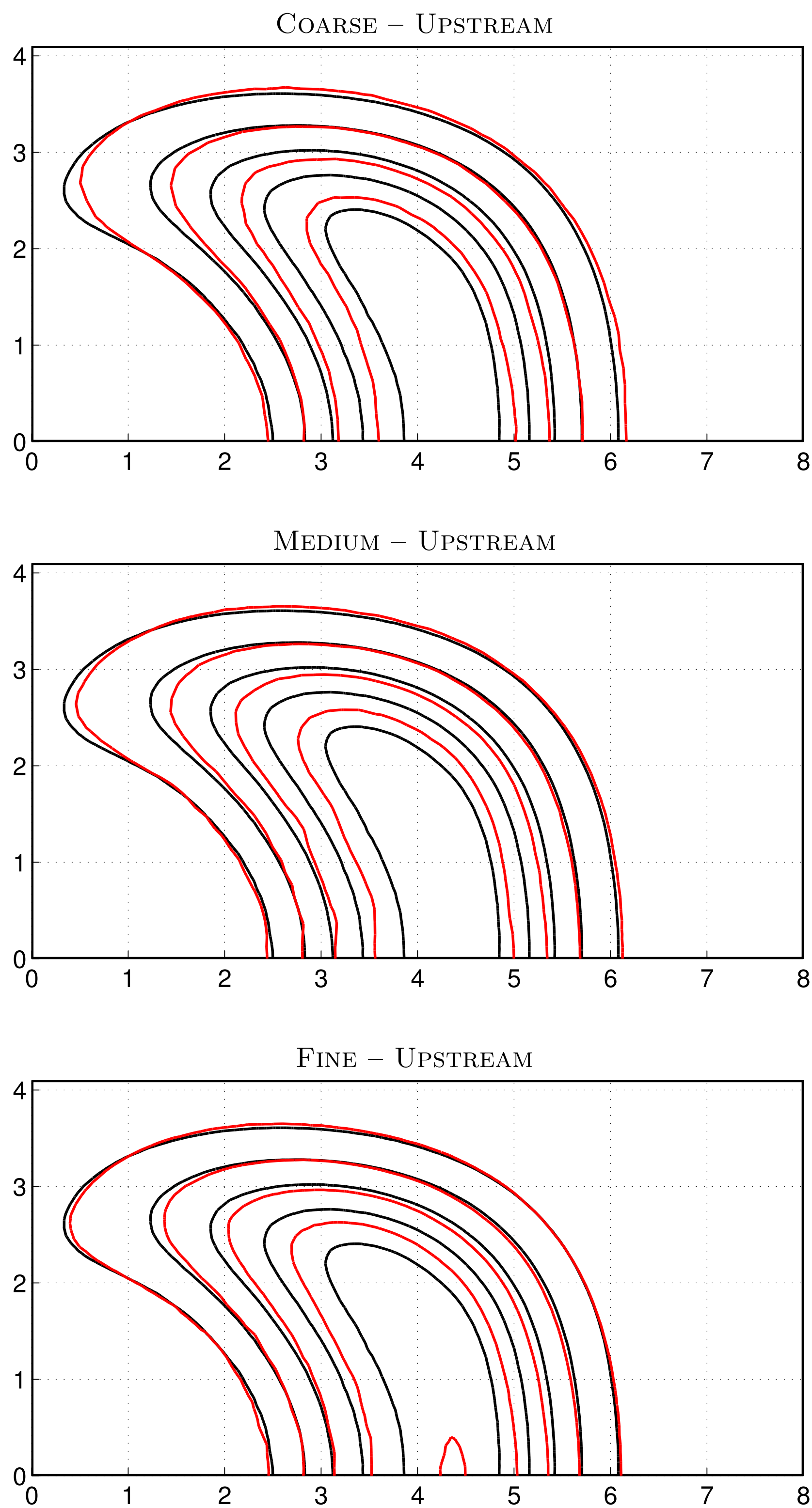


Figure 6.31: Comparison of concentration contours for the *heap_leaching* test case computed using upstream weighting (red) at 81 days on the plane $z = -0.5\,\mathrm{m}$. The reference solution is in black. Concentration contours are $c = 0.05$, $c = 0.10$, $c = 0.15$, $c = 0.20$, $c = 0.25$.

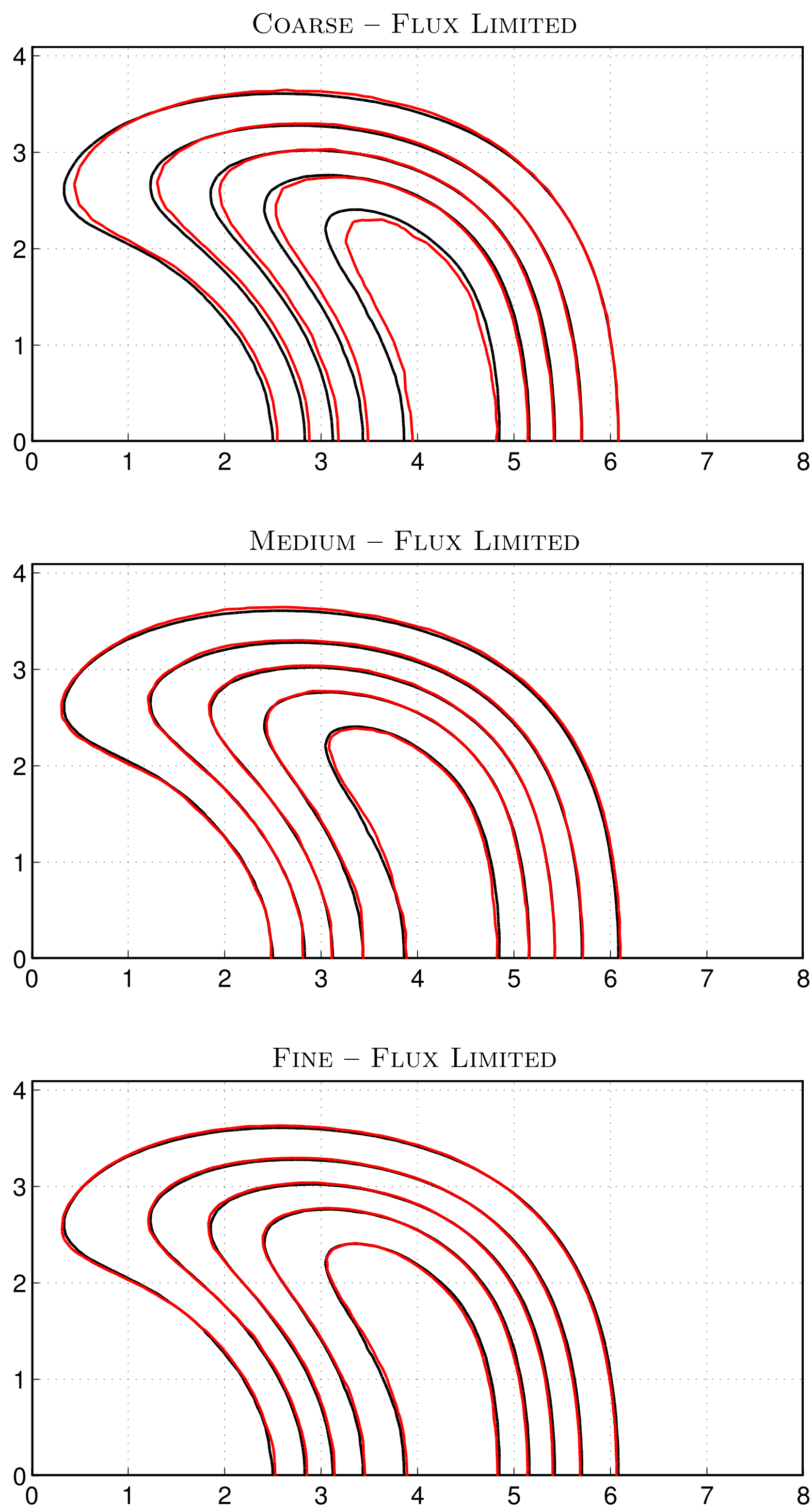


Figure 6.32: Comparison of concentration contours for the *heap_leaching* test case computed using flux limiting (red) at 81 days on the plane $z = -0.5$ m. The reference solution is in black. Concentration contours are $c = 0.05$, $c = 0.10$, $c = 0.15$, $c = 0.20$, $c = 0.25$.

6.7 Time Stepping With IDA

The accuracy and validity of the spatial discretisation and its implementation was verified in the test cases in §6.2–§6.5. In each test case the parameters and choices relating to the temporal solution were given and held fixed, so as to focus on the spatial discretisation. However, the choice of parameters and preconditioners used by IDA in the temporal integration also play a significant role in the computational efficiency and accuracy of solutions. Thus, in this section, the effect of the choice of preconditioner, temporal integration order and tolerances used by IDA is investigated.

6.7.1 The role of different preconditioners

Here we investigate the effect that the choice of preconditioner has on both the efficiency of time stepping and on the accuracy of the solution. It will be shown that the choice of preconditioner is more important for variably saturated flows, where fully saturated regions develop. To illustrate this we investigate the *dry_infiltration* test case in which full saturation never occurs, and the *water_table* test case that has a transient fully-saturated region.

The block Jacobi preconditioner introduced in §4.4.1 does not include coupling between the sub-domains, because the off-diagonal blocks that represent sub-domain coupling in the global iteration matrix are ignored. To understand the importance of including sub-domain coupling in the preconditioner, each preconditioner is tested on two domain decompositions. The first is the *serial* run, where no domain decomposition is performed, and no information is ignored in forming and applying the preconditioner. The second is the *distributed* test run, with two sub-domains, in which sub-domain coupling is ignored by the preconditioner. For each test case, each of the full LU, ILUT and ILU(0) preconditioners described in §5.8.1 is tested. We also test a simple block-jacobi preconditioner with 2×2 diagonal blocks, labelled BJac(2).

The computational performance of different preconditioners for each test case on the medium resolution meshes (see Tables 6.6 and 6.11) is presented in Table 6.21. The *dry_infiltration* test case is solved using the CPU implementation, and the *water_table* runs were performed on the GPU, and in each case the preconditioner is applied on the host.

		serial MPI×1				
		nt	$n\boldsymbol{F}$	npc	time	average e_{mb}
dry_infiltration	Full LU	225	3,747	25	4.85	$7.9 \cdot 10^{-5}$
	ILUT	225	3,745	25	4.78	$7.9 \cdot 10^{-5}$
	ILU(0)	231	3,672	25	4.58	$6.5 \cdot 10^{-5}$
	BJac(2)	258	5,505	244	7.24	$8.3 \cdot 10^{-5}$
water_table	Full LU	178	2,516	17	50.0	not available
	ILUT	210	2,938	26	57.8	not available
	ILU(0)	5,054	128,146	2,847	2331	not available
	BJac(2)	–	–	–	–	–

		distributed MPI×2				
		nt	$n\boldsymbol{F}$	npc	time	average e_{mb}
dry_infiltration	Full LU	231	3,756	25	2.70	$6.2 \cdot 10^{-5}$
	ILUT	231	3,651	25	2.54	$6.1 \cdot 10^{-5}$
	ILU(0)	230	3,843	25	2.62	$6.4 \cdot 10^{-5}$
	BJac(2)	396	8,544	456	5.84	$5.6 \cdot 10^{-5}$
water_table	Full LU	258	4,293	43	42.9	not available
	ILUT	237	4,122	40	40.7	not available
	ILU(0)	–	–	–	–	–
	BJac(2)	–	–	–	–	–

Table 6.21: Performance of the different choices of local preconditioner for the *dry_infiltration* and *water_table* test cases, for *serial* and *distributed* runs. For each run, the number of time steps (*nt*), number of residual evaluations ($n\boldsymbol{F}$), number of times the preconditioner is formed (npc), wall time (time) and mass balance error (average e_{mb}) are listed. Runs marked with — failed to converge.

Each of the preconditioners based on sparse factorisations have almost identical computational performance and mass balance error for the *dry_infiltration* test case. For both *serial* and *distributed* runs, each of the sparse factorisations requires (almost) the same number of time steps and residual evalua-

tions to reach the final solution. As a result, the strong scaling of the method is very good, whereby the *distributed* run (two sub-domains) has a speedup of about 1.8 over the one *serial* (one sub-domain) run.

Because the ILUN preconditioner is very effective with block jacobi for unsaturated flow, it is worth investigating whether it is possible to use a preconditioner that drops further information. With this in mind, results for the BJac(2) preconditioner that uses only nonzero values in 2×2 blocks on the diagonal are also listed in Table 6.21. While solutions with good mass balance error are obtained using the BJac(2) preconditioner, they take considerably more computational work to obtain. For MPI×1, the number of time steps increases by only 12%, however the preconditioner is computed far more frequently (244 times as opposed to 25, or almost once per time step). The additional residual evaluations required to approximate the Jacobi matrix so frequently impose a considerable computational overhead, which is reflected in the 60% increase in wall time.

Tests performed on other test cases show that this observation is true in general for unsaturated flow problems, including the *unsaturated_transport* test problem for coupled unsaturated flow and transport (indeed, the ILU(0) preconditioner was used to find the results in §6.4). The system matrices for the unsaturated flow problems are relatively well-conditioned, so that while preconditioning is required to guarantee timely convergence of the GMRES iterations, a less-accurate ILU(0) preconditioner performed as well as the more computationally expensive full LU and ILUT preconditioners. It should be noted that for the relatively small mesh size used for the test case here, with 1,406 nodes, the computational overhead of using the full LU preconditioner in place of the ILUT or ILU(0) preconditioners[6] is not apparent, however for larger systems this cost becomes very significant.

The solution of saturated flow is more sensitive to the choice of preconditioner (see Appendix E), as is evident in the results for the *water_table* test case in Table 6.21. For the *serial* case, the full LU and ILUT preconditioners

[6]See §5.8.1 for details of the different sparse factorisations.

are both effective, with wall times of 50 and 58 seconds respectively. On the other hand, the ILU(0) preconditioner is not good enough for the poorly conditioned linear system that arises under saturated conditions, with a wall time of 2331 seconds. Additionally, we note that the BJac(2) preconditioner failed to converge, which is not surprising given the poor performance of the ILU(0) preconditioner under such conditions.

Furthermore, the block Jacobi preconditioner performs poorly when using the *distributed* preconditioner for the *water_table* test. The wall time for the full LU and ILUT decreases for the *distributed* case, however only by a factor of between 1.17 and 1.44 respectively. Furthermore, the block Jacobi preconditioner failed for the *distributed* case.

An interesting observation is that for the *distributed* case, the number of time steps only increases by 13% for the ILUT preconditioner relative to the *serial* case, however the number of residual evaluations increases by 40%. This is because the number of inner iterations of the GMRES method required at each time step increases due to the lower-quality preconditioner, and also because of the residual evaluations to compute the preconditioner matrix, which is computed more often. This illustrates that when using inexact Newton methods, the number of inner iterations of the linear solver, or the number of residual evaluations, is often a better metric of computational efficiency than the number of time steps.

In this thesis, considerable effort was spent assessing the preconditioners in the pARMS library (Saad and Sosonkina, 2004) for the distributed matrices. pARMS provides preconditioners for distributed systems, with ILU preconditioners used to precondition local blocks, and iterative Schur complement and Schwarz procedures handling coupling between sub domains. However, it was not possible to implement pARMS in FVMPor due to problems that we encountered with memory and MPI. To allow the software to model saturated media with multiple sub-domains, further investigation of methods that introduce coupling between sub-domains, and suitable local preconditioners need to be considered.

6.7.2 Higher-Order Temporal Integration

In §6.2–§6.5 the error tolerance was fixed for each problem, and IDA was able to use up to third-order integration, so that the relative accuracy of the different edge weighting methods and formulations could be compared. We now turn our attention to the effect of integration order and error tolerances on the accuracy of the solution of Richards' equation using the PR and MPR formulations with the CV-FE discretisation. We note that the observations that follow are also true for the PC and MMPC formulations, however the effects were less pronounced. The analysis in this section was presented in the paper Cumming et al. (2011).

The *dry_infiltration* test case is used for this analysis, because exact mass balance error can be determined. The results were determined for both sets of initial conditions, $\psi^0 = -7.34\,\mathrm{m}$ and $\psi^0 = -100\,\mathrm{m}$, using upstream weighting on an unstructured 1607-node triangular mesh in Figure 6.33, with refinement in the region of interest on the left hand side of the domain. Finally, the solution was determined on the host using a single domain with an ILUT preconditioner.

First, we consider the effect of the order of temporal integration on the computational efficiency and mass balance errors of the solution for both formulations by varying both the maximum integration order chosen by IDA, $k \in \{1, 2, 3, 4, 5\}$, and the tolerance $\tau_a = \tau_r = \tau \in \{10^{-3},\ 10^{-4},\ 10^{-5}\}$ (see equation 3.92). For each integration order–tolerance pair, the maximum mass balance error and total work[7] were recorded, and plotted in Figure 6.34. Each line in the plot shows the effect of tightening the tolerance for a given BDF order, with a general trend of increasing work and decreasing mass balance error.

For first-order integration, the PR formulation has relatively poor conser-

[7]The total number of residual evaluations in each run is used as the metric for total work because residual evaluation takes over 80% of computation time for this problem, and the wall time for the solution varies linearly with the number of residual evaluations.

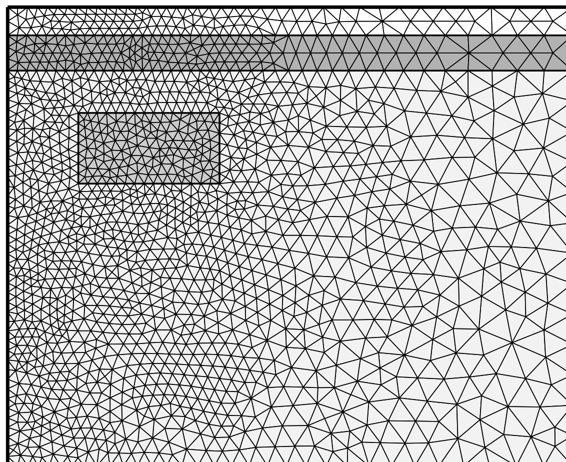


Figure 6.33: Triangular mesh for testing the efficacy of higher-order integration methods on the *dry_infiltration* test case.

vation of mass (two significant digits) compared with the MPR formulation (five significant digits). For higher-order integration, conservation of mass improves for both methods, particularly for $k \geq 3$, and the MPR formulation is between one and two orders of magnitude more accurate than the PR formulation. The observation that higher-order integration is necessary to obtain accurate solutions of the PR formulation is consistent with observations made elsewhere (Tocci et al., 1997, Kees and Miller, 2002). However, we note that while it is less accurate, the mass balance error does not grow with time for the PR formulation and first-order integration, and hence the CV-FE discretisation is mass-conservative for both formulations.

For the MPR formulation, computational efficiency improves dramatically as the order is increased to third-order, after which there is stagnation or small deterioration of performance for fourth-order and fifth-order integration. The same trend is observed for the PR formulation for $\tau = 10^{-3}$, however the efficiency and mass balance are superior for $k = 5$ for $\tau \leq 10^{-4}$. Nevertheless, the mass balance error of the fifth-order PR formulation is not competitive with that for the third-order MPR formulation: to obtain six significant digits accuracy, third-order solution of the MPR formulation requires 60% of

the work for the fifth-order solution PR formulation for both sets of initial conditions.

The stagnation of performance when using fourth-order and fifth-order BDFs and the MPR formulation is because in this case IDA rarely uses $k > 3$, and when it does, convergence issues force it to reduce the order and time step size. The computational overheads associated with changing time step size and order, namely those associated with reforming the preconditioner each time this occurs, add to the computational overhead of higher-order integration for the MPR formulation. As the tolerance is tightened, both formulations take advantage of higher-order time stepping and fourth-order and fifth-order integration become competitive, particularly for the PR formulation where they outperform third-order.

Table 6.22 shows different performance metrics for each formulation for the caisson test case for $\tau = 10^{-3}$, with the most competitive BDF order for each method. The MPR formulation is more mass-conservative, by between one and two orders of magnitude for both $\psi^0 = -7.34$m and $\psi^0 = -100$m, while requiring comparable work. Wall times for both the PR formulation and the MPR formulation are very close, with a small computational overhead of 10% per residual evaluation for the MPR formulation.

	formulation	order	$\boldsymbol{F}$	nt	wall time	average e_{mb}
$\psi^0 = -7.34$m	PR	3	3757	258	4.1s	$3.7 \cdot 10^{-5}$
	MPR	3	2932	224	3.6s	$3.6 \cdot 10^{-6}$
$\psi^0 = -100$m	PR	5	6359	452	6.9s	$5.0 \cdot 10^{-5}$
	MPR	3	6762	448	8.2s	$8.6 \cdot 10^{-7}$

Table 6.22: Computational efficiency and mass balance error for the *dry_infiltration* test case for both the PR and MPR formulations.

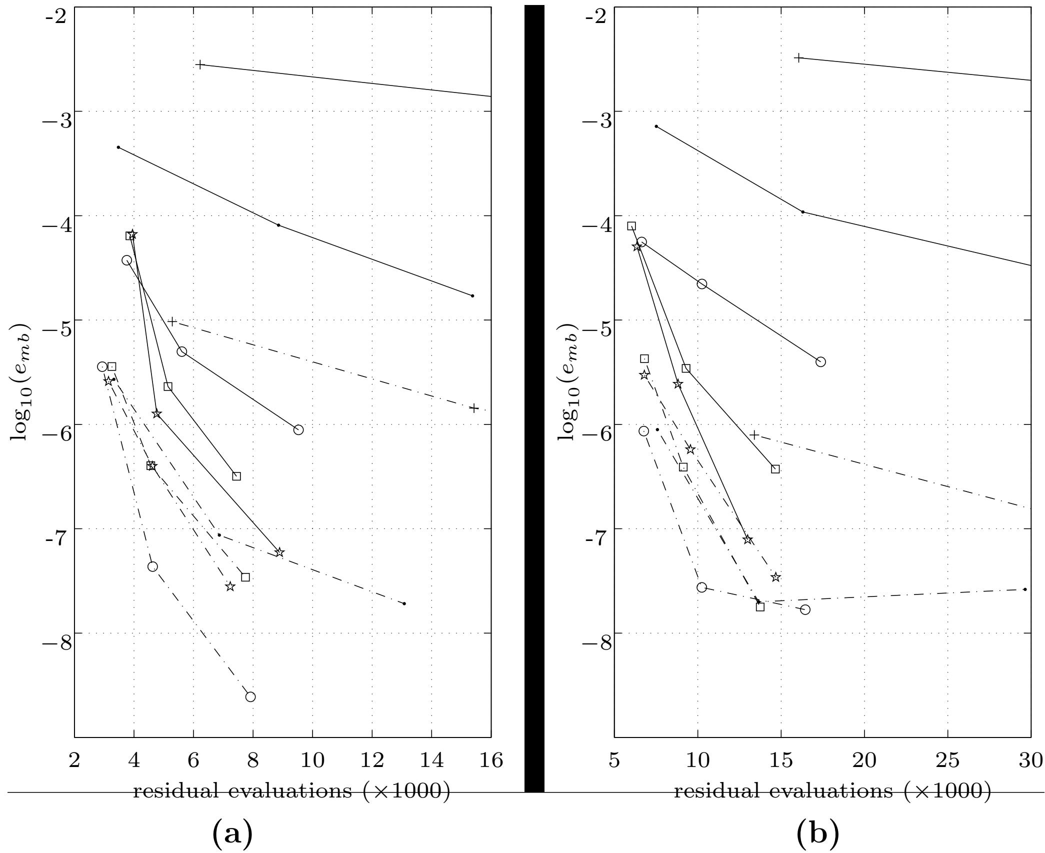


Figure 6.34: Mass balance error for the *dry_infiltration* test case in terms of computational work, measured as number of residual evaluations, for initial conditions: (a) $\psi^0 = -7.34$m; (b) $\psi^0 = -100$m. The solid and dashed lines are for the PR and MPR formulations respectively, with the order of integration for each denoted by: + k=1; . k=2; ∘ k=3; □ k=4; ⋆ k=5.

6.8 Conclusions

In this chapter the formulations and numerical methods introduced in Chapter 3 were verified against some challenging test cases. We summarise the findings of these numerical case studies in the following paragraphs.

The solution of variably-saturated flow with Richards' equation: the *dry_infiltration* and *water_table* tests cases.

The *dry_infiltration* and *water_table* test cases presented in §6.2 and §6.3 were used to test the proposed methods for solving variably-saturated flow. For both test cases, it was possible to accurately reproduce the references solution on relatively coarse meshes by applying flux limiting to the mobility term, whereas very fine meshes were required to obtain solutions with equivalent accuracy using upstream weighting. By virtue of using coarser meshes, flux limiting required between 5 times to 20 times less computational work to determine solutions of equivalent accuracy to upstream weighting.

Of the parabolic and van Leer limiters, the parabolic limiter was more efficient and converged for all mesh sizes and choice of flow direction indicator. The van Leer limiter produced solutions of equivalent accuracy to the parabolic limiter, however was slow to converge for fine meshes in the *water_table* test case, and also failed using FDI(φ) on some meshes. The flow direction indicator FDI($\boldsymbol{\nabla}\phi$) was the best overall choice for the solution of Richards' equation, because it is simpler to form than FDI($\boldsymbol{q}$) to implement, and produced good results for each of upstream weighting and the different limiters.

The solutions for both test cases for Richards' equation obtained using the MPR formulation were indistinguishable from those from the PR formulation in §6.2 and §6.3. The number of residual evaluations required to obtain the solution for the MPR was equivalent to the PR, and the MPR formulation had a computational overhead of between 10% to 20% relative to the PR formulation, despite having twice as many equations and variables. The low computational overhead for the MPR formulation is due to the efficiency of the Schur complement preconditioner proposed for the modified mixed formulations in §3.3.2.

Additionally, in §6.7.2 the MPR formulation was shown to have considerably better conservation of mass than the PR formulation, particularly for first-

order temporal integration. Hence, if conservation of mass is a high priority, or if first-order implicit time stepping is used, the mixed MPR formulation should be used. Otherwise the PR formulation may be used because it is a little easier to implement, and offers a saving of between 10% to 20% in computational overhead.

The solution of coupled variably-saturated flow and contaminant transport: the *unsaturated_transport* and *heap_leaching* test cases

The *unsaturated_transport* test case in §6.4 was used to test unsaturated flow and contaminant transport. Again, the flux limiting improved the quality of solutions on coarse meshes, required between 4 times to 5 times less work to find solutions of equivalent accuracy to upstream weighting. This test case was more sensitive to the choice of limiter for the mobility term than the two test cases for Richards' equation. The parabolic limiter was more accurate and required significantly less computational work than the van Leer limiter. The *unsaturated_transport* test case had very little dispersion, so that the solute transport was advection-driven, and flux limiting was required to reduce numeric diffusion in the concentration solution. However, unlike the mobility term, both limiters produced equivalent solutions for the same amount of computational work when applied to the advection term.

The benefits of using the modified mixed MMPC formulation instead of the PC formulation are less obvious than for Richards' equation. Indeed, the MMPC formulation can only be justified with first-order temporal integration, because for higher-order integration both formulations had equivalent mass balance errors, while the MMPC formulation imposed a computational overhead of 10% to 20%.

The *heap_leaching* test case in §6.6 was formulated to verify the efficacy of flux limiting on unstructured three-dimensional meshes. The test case had very small diffusion and dispersion parameters, such that the contaminant transport was advection-dominated. As was the case for the two-dimensional

test cases, solutions computed using upstream weighting were prone to numerical diffusion. Numerical diffusion was reduced significantly by using flux limiting, to the extent that solutions computed on the coarse mesh with flux limiting were more accurate than those computed on the reference mesh with upstream weighting.

Contaminant transport on tidal beaches: Zhang's laboratory experiments

A numerical investigation into the laboratory experiments of near-shore contaminant transport under tidal forcing was performed in §6.5. Phenomena observed in the experiments, performed by Zhang (2000), are very challenging to reproduce numerically due to unsteady, density-dependent flow. The simulations reproduced the shape and time evolution of the contaminant plume both with and without tidal variation of the sea water level. Other features, such as the formation of density-dependent fingers in the contaminant plume, and a diffuse contaminant region near the beach, were not reproduced accurately in the numerical solutions. We concluded that to reproduce these features, it will be necessary to perform further simulations that explicitly model the underlying causes of the features.

The impact of different preconditioners

The effect of the different preconditioners on the time stepping scheme was investigated in §6.7.1. It was found that the ideal choice of preconditioner was dependent on the presence of saturated flow, because the matrices that arise under saturated flow conditions were not as well-conditioned as those for unsaturated flow.

For unsaturated flow, the less-accurate ILU(0) factorisation was as effective as the more expensive LU and ILUT factorisations, and block Jacobi approach for distributed matrices scaled well as the number of sub-domains

increased. For variably-saturated flow, however, tighter restrictions were placed on the choice of preconditioner. The block Jacobi approach scaled very poorly compared to the serial preconditioner. Furthermore, the ILU(0) preconditioner was not sufficiently accurate for the local blocks, and factorisations that allowed fill in, specifically the full LU and ILUT preconditioners, were required.

Higher-order temporal integration

The effect of temporal integration order and tolerances was investigated In §6.7.2. Higher-order integration (third-order and higher) greatly reduced computational effort required to compute solutions for both the PR and MPR formulations. Mass balance error for the MPR formulation was considerably better (by three orders of magnitude) than the PR formulation when using first-order integration. For higher-order integration, both formulations exhibited excellent conservation of mass, however the MPR formulation was still more accurate by two orders of magnitude. These observations suggest that for first-order temporal integration, the modified mixed formulation should be used. However for higher-order integration where both formulations are very accurate, and the modified mixed formulation is only required if very precise mass balance errors are critical.

Chapter 7

Computational Performance

In this chapter, the compuational performance of the CPU and GPU implementations of FVMPor will be evaluated. The chapter is organised as follows. First, in §7.1 the test problems used to evaluate computational performance will be introduced, along with an overview of the hardware utilised in our numerical experiments. Then, in §7.2 a detailed comparison and analysis of computational performance the CPU and GPU implementations will be presented.

7.1 Test Setup

In this section, two-dimensional and three-dimensional test cases that can be used to evaluate both coarse-grained and fine-grained parallelism in the MPI-OpenMP-CUDA framework are chosen[1]. Coarse-grained parallelism is implemented through domain decomposition, which uses the block Jacobi preconditioner for the distributed linear system illustrated in Figure 4.7. Furthermore, the only local preconditioner that offers fine-grain parallelism

[1]Recall, that coarse-grained parallelism uses domain decomposition to distribute the problem across MPI processes, and fine-grained parallelism uses either OpenMP or CUDA on each sub-domain.

on the GPU is the multi-colour version of the ILU(0) preconditioner (MC-ILU(0)) described in Algorithm 4.6. This restricts the choice of problems that can be implemented in the full MPI-OpenMP-CUDA framework, because in §6.7.1 it was shown that the block Jacobi with ILU(0) preconditioner is unsuitable for saturated flows. Hence, to investigate the performance of FVMPor when the preconditioner can be applied in parallel, an unsaturated flow test case is required.

The *dry_infiltration* test case with the *dry* initial conditions of $\psi^0 = -7.34\,\text{m}$ that was introduced in §6.1.1 is used for this testing in both two dimensions and three dimensions. The two-dimensional domain is the same as that described in Figure 6.1. For the three-dimensional test case, the two-dimensional problem is extruded by 0.5 m, and we focus on the top left region of the domain (3 m high by 5 m wide). Furthermore, the block of zone 3 is only extended 0.25m, to guarantee a true three-dimensional flow. The meshes are constructed using unstructured triangles and tetrahedra, with four levels of refinement in two dimensions, and five in three dimensions, details of which are provided in Table 7.1. Simulations were run to 30 days and 20 days for the two-dimensional and three-dimensional cases respectively.

Two Dimensions		Three Dimensions	
Name	Nodes	Name	Nodes
2D1	23,570	3D1	23,316
2D2	95,052	3D2	47,005
2D3	387,818	3D3	83,529
2D4	856,317	3D4	139,561
		3D5	184,186

Table 7.1: The name and number of nodes in each of the meshes used to test the parallel performance of FVMPor.

Only the PR formulation of Richards' equation is used for all of the tests, however we note that identical results are observed for the MPR formulation. This is because the additional steps performed in the residual evaluation and preconditioner of the MPR formulation are computationally inexpensive, and can be implemented efficiently on the GPU.

To balance spatial and temporal errors we set the relative tolerance τ_r and absolute tolerance τ_a as follows

$$\begin{aligned}\tau_t &= \delta \Delta x_{\min}^2, \\ \tau_a &= \delta \times 10^{-6},\end{aligned} \tag{7.1}$$

where $\Delta x_{\min}$ is the minimum element radius[2] of the mesh, and δ is a tuning parameter. This heuristic approach is based on the assumption that spatial error is second order in space (Ewing et al., 2002, Kees and Miller, 2002). For the *dry_infiltration* test case used in these tests, it was found that choosing $\delta = 0.3$ ensured stable convergence for all mesh resolutions in both two and three dimensions.

7.1.1 Test Hardware

A high-performance desktop machine with a total of 8 cores in 2 sockets (2 by 4-core Intel Xeon E5620 CPUs rated at 2.4GHz) and two NVIDIA Tesla C2050 GPU cards, each with 3GB of RAM were used for all of these tests. The ECC error-checking on the GPU memory was turned off, which gave a speedup of roughly 10% over the entire solution. The Intel C++ compiler version 11.1 and CUDA toolkit version 4.0 were used to build the code. Both CPU and GPU versions of the code use double precision floating point arithmetic.

7.2 Results

The results of the numerical experiments are presented here. Throughout the discussion that follows, the CPU-only version of the code is run with one core assigned to each MPI process. Thus, CPU×4 indicates that four CPU cores

[2] The minimum element radius is the smallest distance between two adjacent nodes, or the shortest edge length, in the mesh.

were used, each assigned to a sub-domain. For the GPU implementation, one CPU core and one GPU are assigned to each MPI process, except when the B-ILU(0) preconditioner is used, in which case all 8 CPU cores are sub-divided amongst the MPI processes to compute and apply the local block Jacobi preconditioners in parallel.

The results will be presented as a series of observations, each followed by justification and references to the associated figures at the end of the chapter.

The strong scaling up to 8 cores is good for the CPU-only MPI implementation

First, we investigate the speedup of the CPU implementation relative to the serial version (CPU$\times$1) as the number of sub-domains and cores increases, that is, the *strong scaling*. The strong scaling of wall time is plotted against the ideal speedup for the largest meshes in two and three dimensions, mesh 2D4 and mesh 3D5 respectively, in Figure 7.1. The scaling is close to ideal when between 2 to 4 CPU cores are used. However the scaling is not as good for more than 4 cores: 8 cores gives a speedup of 6.1 for the two-dimensional mesh, and 5.2 for three dimensions.

There are two reasons for the less than ideal scaling for more than 4 cores. The first reason is that the block Jacobi preconditioner is less accurate as the number of sub-domains increases, because more inter-domain coupling information is ignored. Using a less-accurate preconditioner leads to an increase in the number of residual evaluations in two ways: first, the number of inner iterations taken by the GMRES method increases; and secondly, the preconditioner is formed more frequently, which entails the shifted residual evaluations described in equations (4.25) and (4.26). This is the case for the block Jacobi preconditioner in both the two-dimensional and three-dimensional tests, where the number of residual evaluations for 8 cores is approximately 10% higher than for 1 core. However, the increase of 10% in the number of residual evaluations does not account for all of the scaling

drop of between 20% to 30% in the Figure 7.1 when going from 4 cores to 8 cores.

The second reason for the less than ideal scaling for more than 4 cores is not as obvious. The problem is not with load balancing, which is very good for the static domain decomposition precomputed using ParMETIS. Furthermore, communication overheads are very small when using 8 cores: less than 1% of time to solution in Figure 7.2.

A key observation is that the performance drops for all of the steps in the preconditioner and residual evaluation when more than 4 cores are used. The most likely reason for this is memory affinity for the MPI processes when the cores in more than one CPU socket are used. The test computer described in §7.1.1 has two sockets with 4 cores each, and each socket has six memory slots for a total of 12 memory slots. A CPU core can access memory in any slot, however the memory latency is higher if the slot is not one of the six slots associated with the core's socket (Levinthal, D., 2009). To minimise memory latency, each MPI process should have fixed *affinity* with one core, whereby the process uses that core exclusively, and allocates memory in a slot associated with that core's socket. The following steps were taken to ensure that this is the case:

- OpenMP thread affinity was set so that OpenMP threads in each MPI process were fixed to run on cores in the same socket (Intel, 2008, §3.2). In the case of one OpenMP thread per MPI process, each MPI process is fixed on one core.
- High-performance libraries provided by Intel, the chip manufacturer, were used for allocating memory in the hope that they would allocate memory in appropriate slots. To this end, the C++ allocator in the MKL library and the `icpcMalloc` function in the Intel Performance Primitives library were both tested.

These approaches did not address the problem, which may suggest that the

problem is due to the affinity of the MPI process not agreeing with the affinity of the OpenMP thread. Further investigation is required to ensure that the MPI and OpenMP affinity are in agreement (Zhang et al., 2010).

Despite the less than ideal scaling for more than 4 cores, the performance of the solver improves significantly as the number of cores increases to the maximum possible of 8 cores. The scaling of between 5.2 times and 6.1 times speedup on 8 cores offers the fastest time to solution on our test computer, and is competitive when compared to similar published results (Cai et al., 1994). Hence, for the analysis presented in this chapter, the 8 core (CPU×8) timings on each mesh will be used as the *baseline* test case for comparison with the GPU implementation.

Evaluating the residual is the dominant computational cost, representing between 80% to 90% of the total computational cost for the baseline CPU×8 case.

To better understand the computational bottlenecks in the CPU-only implementation of FVMPor, Figure 7.2 illustrates the the proportion of the time to solution spent in each part of the solver for the 2D4 and 3D5 meshes. The first main observation is that the majority of time, 81% in three dimensions and 91% in two dimensions, is spent performing residual evaluations.

The second important observation from Figure 7.2 is that the cost of evaluating the residual relative to the cost of applying the preconditioner is much higher in three dimensions[3], so that preconditioning costs account for 5 times as much of the total time to solution in two dimensions compared to three dimensions. This is because the computational cost of applying the preconditioner is equivalent for two-dimensional and three-dimensional meshes with the same number of nodes, whereas residual evaluation is significantly more

[3]The two operations of residual evaluation and preconditioner application are performed on each inner iteration of GMRES, and as such they account for the majority of computational effort. Later analysis will show that applying the preconditioner accounts for over 98% of all the costs associated with preconditioning on the host (see Figure 7.8(b)).

expensive in three-dimensions. To illustrate this, Table 7.2 shows the cost of performing a residual evaluation and applying the preconditioner for the two-dimensional mesh 2D2 and the three-dimensional mesh 3D3, which have a similar number of nodes. The time taken to apply the preconditioner is almost identical for both meshes, with the slightly larger matrix associated with mesh 2D2 taking 1.12 times longer to apply than for mesh 3D3, whereas the residual evaluation takes over 6 times longer for the three-dimensional mesh 3D3. This is because computing and gathering the face fluxes for each control volume is considerably more expensive in three dimensions where each control volume has an average of 32 faces, compared to an average of 6 faces in two dimensions.

Furthermore, in both two and three dimensions, the cost of residual evaluation relative to the cost of applying the preconditioner, shown as ratios in Table 7.2, is the same regardless of the number of nodes in the mesh. That is, both residual evaluation and preconditioner application scale nearly identically as the number of nodes in the mesh increases, because the sparse triangle solves and sparse matrix-vector products in the preconditioner application and residual evaluation both scale linearly[4].

	2D2	3D3	ratio 2D2/3D3 (3D3/2D2)
nodes	95,052	83,529	1.14 (0.88)
residual evaluation	$1.2 \cdot 10^{-2}$ s	$7.4 \cdot 10^{-2}$ s	0.16 (6.17)
preconditioner apply	$1.28 \cdot 10^{-3}$ s	$1.13 \cdot 10^{-3}$ s	1.13 (0.88)

Table 7.2: The average time spent evaluating the residual and applying the preconditioner for baseline 8-core simulations performed using two-dimensional and three-dimensional meshes with a similar number of nodes.

The remaining computational overheads measured in Figure 7.2 are the internal operations of IDA, and halo communication. For both two and three

[4]For dense matrices the combined forward/backward substitution and matrix-vector products are $O(n^2)$. However, for sparse matrices where the number of non zeros in a row is bounded by a small scalar, such as the maximum number of neighbours for a node, these operations are linear, or $O(n)$.

dimensions, IDA accounts for less than 10% of the total solution cost, and the cost of updating halo information using asynchronous MPI communication is negligible, less than 1%, even with 8 sub-domains[5].

From these observations, it is clear that the key to optimising the solution in the Newton-Krylov framework is to target the residual evaluation, which takes between 80% to 90% of the total time to solution. The next two observations focus on performing the residual evaluation on the GPU. Then the remaining steps, namely the NVector routines in IDA and different methods for applying the preconditioner on the GPU, will be investigated.

The GPU implementation offers significant speedup over the baseline case for residual evaluations: up to 3 for one GPU, and 6 for two GPUs.

We begin the investigation of the GPU implementation by looking at the efficiency of the residual evaluation on the GPU. Figure 7.3 shows the speedup of residual evaluation when using one and two GPUs relative to the baseline CPU×8 case. Performing the residual evaluation on the GPU gives good weak scaling, that is, the relative performance of the GPUs improves as the number of nodes in each mesh increases. For fine meshes, the speedup is over 3 on one GPU (between 3.1 and 3.2 for two-dimensional meshes with greater than 400,000 nodes, and between 3.3 and 3.4 for three-dimensional meshes with more than 80,000 nodes).

An important observation from Figure 7.3 is that speedup in three dimensions is considerably better than in two dimensions for meshes with fewer nodes: the mesh 2D1 offers a speedup of 1.4, whereas the mesh 3D1, which has a similar number of nodes, offers a speedup of 2.8. The reason for this is the much higher number of edges and control volume faces per node in three dimensions (recall that each control volume has over 5 times as many faces

[5] Communication costs between processes that use shared memory are very small, however it is reasonable to expect that they would become more significant on clusters.

in three dimensions). Thus, the sparse matrices used for shape function interpolation have more rows in three dimensions[6], and the working vectors of face and edge values are also longer.

Both SPMV and vector operations exhibit good weak scaling on the GPU, whereby their performance improves as the size of the matrices and vectors increases. This is illustrated in Figure 7.4, which shows the speedup of the GPU implementation relative to the baseline CPU case in each stage of the residual evaluation on the smallest two-dimensional and three-dimensional meshes, 2D1 and 3D1 respectively, each of which has about 23,000 nodes. Each step of the residual evaluation sees better speedup on the three-dimensional mesh, for example:

- The sparse matrices used for shape function interpolation in the interpolation step have 5 times more rows in three dimensions, and the interpolation step has speedup of 2.7 for 3D1 compared to a speedup of 1.5 for 2D1.
- Flux assembly (see Listing 5.8) uses only vector addition and vector multiplication with vectors of face values such as pressure head gradient and relative permeability. The face vectors are longer, again by a factor of 5, in three dimensions. The good weak scaling of GPU operations on longer vectors is evident, with speedup of 4.1 on the longer vectors in three dimensions, relative to speedup of 2.6 in two dimensions.

Finally, we note that the strong scaling for the residual evaluation on two GPUs is very good. In Figure 7.3 we see that two GPUs is almost twice as fast as using one GPU.

[6] Recall from (4.10) that the interpolation matrices have n_f rows, where n_f is the number of rows in the mesh.

Renumbering nodes, edges and faces to improve cache performance accelerates the residual evaluation of both the GPU and CPU implementations.

We now investigate the impact of the renumbering scheme proposed in §5.7 on the computational performance of the residual evaluation. The scheme was proposed to improve the GPU performance of indirect indexing in the gather and scatter operations in the residual evaluation. However, the renumbering is used for both the CPU and GPU implementation of FVMPor, so its impact on both the CPU and GPU implementations is presented here.

The speedup of each step in the residual evaluation due to the renumbering scheme is illustrated in Figure 7.5. Both the CPU and GPU implementations benefit from the renumbering, with very similar speedup for both two and three dimensions. Flux assembly is the only step of the residual evaluation that is not affected by the renumbering scheme. This is because flux assembly does not use any of the indirect indexing that the scheme was designed to optimise[7].

The renumbering scheme accelerates the other steps in the residual evaluation, all of which use indirect indexing between nodes, edges and faces:

- **Interpolation**

  This step computes the pressure head gradient and the density in the buoyancy term (using SPMV to perform shape function interpolation as described in §4.2.2) and the density in the advection term (using edge-based weighting in Algorithm 4.4). Each of these operations use indirect indexing, for which the renumbering scheme leads to speedup of approximately 1.2 on the CPU, and between 1.3 and 1.4 on the GPU.

  It is worth noting that the renumbering scheme had a much greater

[7] The flux assembly in §5.6.5 computes the flux at each control volume face using Listing 5.8 using variable values and gradients computed in previous *interpolation* and *fluid properties* steps, in which gather and scatter operations are performed.

impact on the SPMV operations with early versions of CUSPARSE[8]. Subsequent versions of CUSPARSE have reduced the sensitivity of the SPMV to the sparsity pattern of the matrix, which reduced the impact of the scheme on operations such as interpolation that use SPMV.

- **Fluid Properties**

  Computing the fluid properties with Listing 5.7 involves two large computational expenses. The first is computing the moisture content, relative permeability and storage term from the van Genuchten-Mualem model, which is particularly expensive on the CPU[9]. This computation is not affected by the renumbering scheme, and is performed very efficiently in parallel on the GPU.

  The second computational overhead is the edge-based weighting used to compute the permeability at each control volume face, which uses the scatter operation discussed in detail in §5.7.1. This is a larger overhead on the GPU than the CPU, because it uses indirect indexing, and due to the relative efficiency of the van Genuchten-Mualem computations on the GPU. As such, the renumbering scheme has the greatest impact here on the GPU, with a speedup of between 2 and 2.2.

- **Residual Assembly**

  The residual assembly is a relatively inexpensive step, taking about 10% of the residual evaluation (see Figure 7.6). The over-riding cost is in the sparse matrix-vector product that gathers the face fluxes for each control volume in Listing 5.9. This is the only step of the residual evaluation for which the renumbering gives a larger speedup on the CPU than the GPU. The sparse matrix has fewer rows than the interpolation matrices, and more nonzero values in each row, particularly in three dimensions.

[8] The scheme was first implemented with the first version of CUSPARSE, released with CUDA 3.0.

[9] Determining the fluid properties according the van Genuchten-Mualem model involves expensive floating point operations such as square root and power operations in equations (2.9), (2.10) and (2.29).

From these observations, we see that the renumbering scheme has a greater overall impact on the GPU, with a speedup of between 1.4 to 1.5. The larger cache and more sophisticated memory hardware on the CPU made it less sensitive to the numbering scheme, however it still had a significant impact with speedup factor between 1.15 and 1.2.

The MPI-CUDA implementation of NVector is significantly faster than the MPI-CPU implementation, and exhibits very good weak scaling.

We now investigate the performance of the Newton-Krylov solver in IDA with the GPU implementation of NVector that was discussed in §5.5. As noted earlier, the weak scaling of vector operations such as those in NVector is very good on the GPU. This is illustrated in the plot of speedup on the GPU for the two-dimensional mesh in Figure 7.7. For one GPU we observe a speedup of 2 for small meshes, and a speedup of 7 as the mesh is refined.

Unlike the CPU version, costs associated with preconditioning are a major bottleneck in the GPU implementation. Furthermore, costs associated with applying the preconditioner account for 99% of the total preconditioner cost, with the factorisation accounting for the remaining 1%.

The relative time spent in each part of the solution process for the 2D4 mesh on one GPU using the ILU(0) host preconditioner is shown in Figure 7.8(a). The effect of implementing the residual evaluation and IDA on the GPU is evident relative to the baseline case in Figure 7.2(a). The residual now accounts for only 45% of the time, compared to 81% for the baseline case, and likewise IDA only takes 2% compared to 7%. The remaining 51% of time is spent on performing preconditioning operations, which is a five-fold increase from 10% in the CPU implementation. There are two reasons for

this:

- The first reason is that the absolute amount of time spent evaluating the residual and IDA operations has decreased for the efficient GPU implementation.
- The second reason is that with only one GPU, there is only one MPI process. Hence, the entire global iteration matrix is preconditioned in serial. This is in contrast to the baseline 8-core case, which benefits from processing eight smaller local diagonal blocks in parallel.

The breakdown of time spent in operations related to preconditioning (factorising, applying and copying data between the host and the device) is illustrated in Figure 7.8(b). We note that applying the preconditioner (solving the triangular systems in (4.28) and copying vectors between host and device each time the preconditioner is applied) dominate the preconditioning costs. Remarkably, computing the sparse factorisation takes only 1% of the total preconditioning time. This is because the preconditioner is formed and factorised only periodically by IDA (56 times in this case), while it is applied once for every inner iteration of the GMRES method (21,784 times). Because factorisation has negligible cost for the problem investigated here, it is performed on the host, and we focus on methods for accelerating the application of the preconditioner.

The B-ILU(0) and MC-ILU(0) preconditioners have the fastest application times, with the MC-ILU(0) preconditioner outperforming the host ILU(0) preconditioner by an order of magnitude for fine meshes.

In the results presented so far in this chapter, the host ILU(0) preconditioner has been used to compute both the CPU and GPU results. We now investigate the efficiency of the three additional local preconditioners based on ILU(0) factorisations that were presented in §5.8.2 for the GPU version:

- The host preconditioner B-ILU(0), which performs a block Jacobi preconditioner to the local matrix. If one GPU is used, then all 8 CPU cores are used to perform a block Jacobi preconditioner with 8 blocks. If two GPUs are used, 4 GPUs are assigned to each process where a block Jacobi preconditioner with 4 blocks is used.

- The device preconditioner MC-ILU(0), which performs the application phase on the GPU using the matrix stored in block form (4.29).

- The device preconditioner CUSP-ILU(0), which performs the application phase on the GPU using the sparse triangle solves implemented in the CUSPARSE library.

Figure 7.9 shows the average time taken to apply each of the preconditioners (including time spent copying between host and device for the ILU(0) and B-ILU(0) methods) using one GPU on the largest two-dimensional mesh, 2D4. Of the two preconditioners applied on the host, the B-ILU(0) preconditioner is approximately 4.2 times faster than the ILU(0) preconditioner at all mesh sizes. The speedup of 4.2 is not ideal, given that B-ILU(0) uses 8 cores instead of the 1 core used by ILU(0). However, both methods have identical overheads associated with copying data between host and device, such that 43% of the application time for the B-ILU(0) preconditioner is spent on copying.

MC-ILU(0) exhibits the best weak scaling of any of the preconditioners. For small meshes it is more efficient than the ILU(0) preconditioner, but not as good as B-ILU(0). However, for meshes larger than 100,000 nodes it is the most efficient preconditioner: up to 11 times faster than the ILU(0) preconditioner, and 2.6 times faster than the B-ILU(0). The benefit of not copying between the host and device is apparent for the MC-ILU(0) preconditioner. Applying the preconditioner alone is 1.4 times faster for MC-ILU(0) than B-ILU(0) on the 2D4 mesh — the total speedup of 2.6 is because no data is copied between the host and device, which we recall took nearly half of the total application time for the B-ILU(0) preconditioner.

On the other hand, the CUSP-ILU(0) preconditioner is the least efficient of all the methods, despite using the same multi-colouring technique as MC-ILU(0) to obtain fine-grained parallelism. This is because the way that the factorised matrix is stored is not amenable to good data access patterns on the GPU. Unlike the MC-ILU(0), the factors are not stored in permuted form. Instead, the permutation vector is used to traverse the matrix and solution vector during the triangle solves. In this manner, the matrix rows and entries in the right hand side vector are not processed contiguously, which leads to low cache reuse on the GPU.

The optimised preconditioners give us similar speedup over the entire solution process as for the residual evaluation alone.

We now look at the speedup for the entire time to solution for the GPU implementation. Figure 7.10 shows the speedup for the 1 GPU and 2 GPU relative to the 8-core baseline case, for each of the ILU(0), B-ILU(0) and MC-ILU(0) preconditioners[10].

First, consider the two-dimensional case in Figure 7.10(a). The GPU implementation was faster than the 8-core baseline case, however the choice of preconditioner has a large affect on the total speedup. The 1 GPU code with the host ILU(0) preconditioner was almost twice as fast for the large two-dimensional meshes – which is considerably less than the speedup of the residual function. To obtain scaling across all parts of the solver that is comparable to the speedup in residual evaluation alone, either of the B-ILU(0) or MC-ILU(0) preconditioners must be used. Indeed, with the MC-ILU(0) preconditioner, the speedup for the two-dimensional test case in Figure 7.10(a) approaches that of the residual evaluation on large meshes.

Figure 7.10(b) shows that the speedup of time to solution is better in three dimensions than in two dimensions. This is by virtue of the better speedup

[10] The results for the CUSP-ILU(0) preconditioner are omitted due to their poor performance relative to the other preconditioners.

of the residual evaluations in three dimensions, and the lower relative cost of preconditioning in three dimensions.

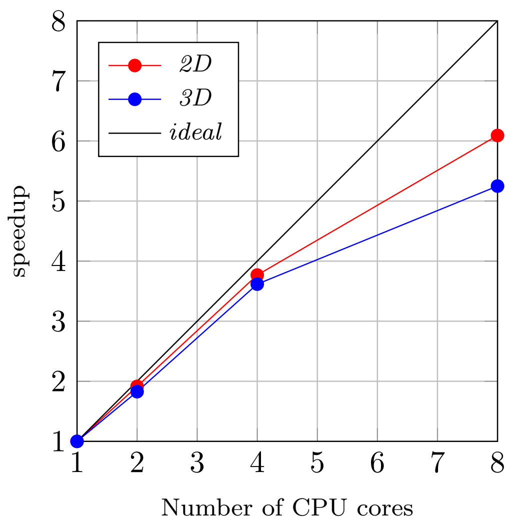


Figure 7.1: The strong scaling for the CPU version.

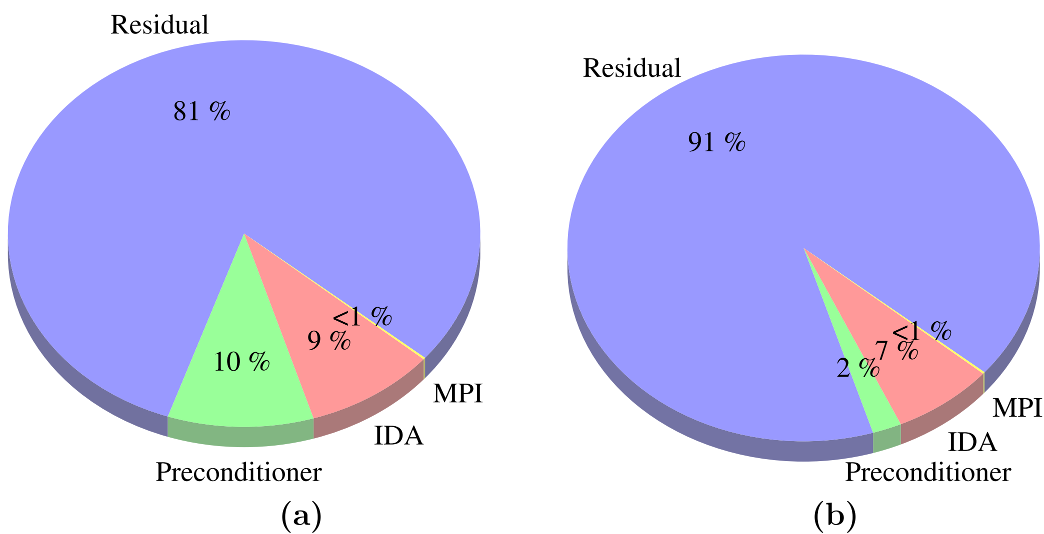


Figure 7.2: Breakdown of time spent in each part of the solver for CPU×8 for mesh 2D4 (a) and mesh 3D5 (b).

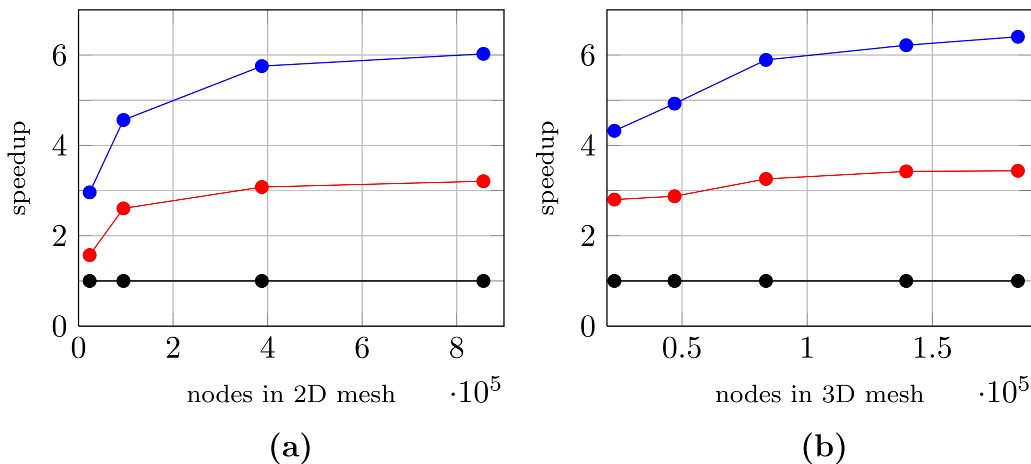


Figure 7.3: Speedup of physics computation for GPU×1 and GPU×2 (red and blue respectively) relative to CPU×8 (black) as the number of nodes in the mesh increases for two dimensions (a) and three dimensions (b).

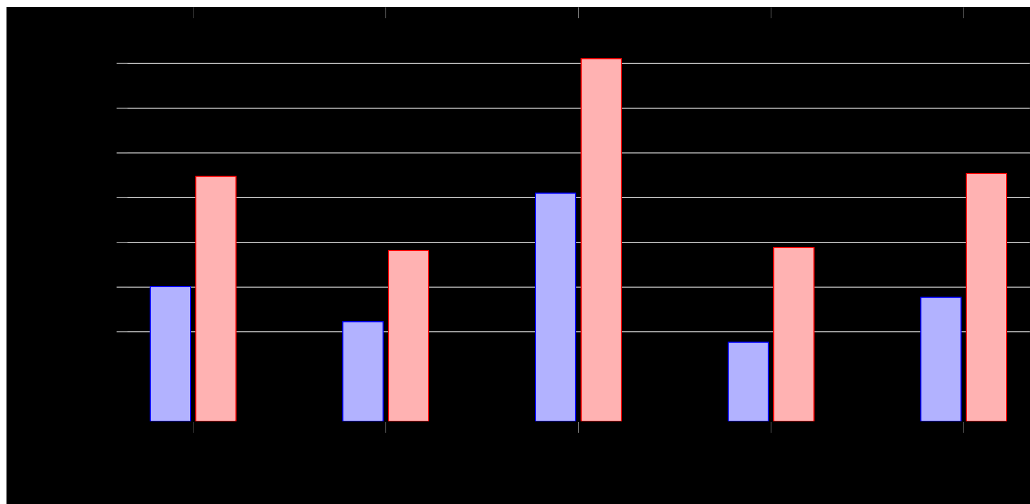
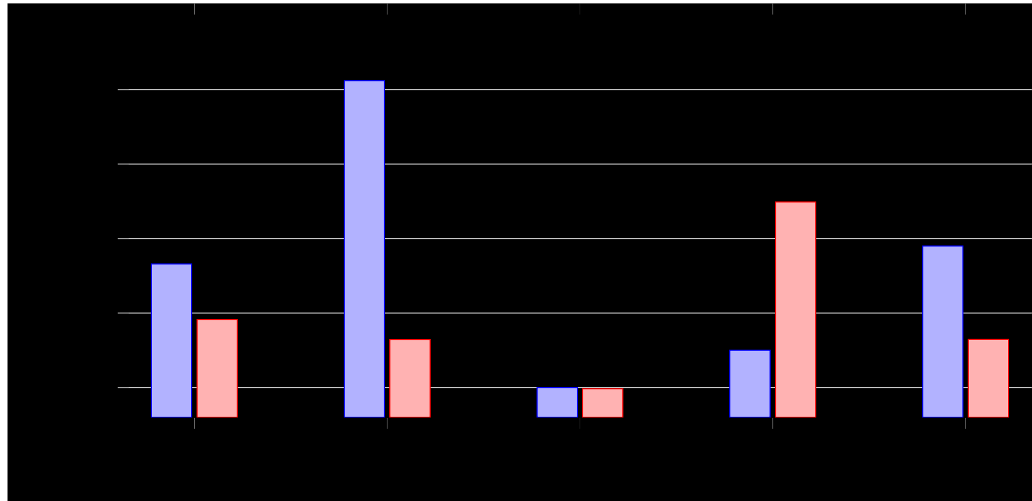
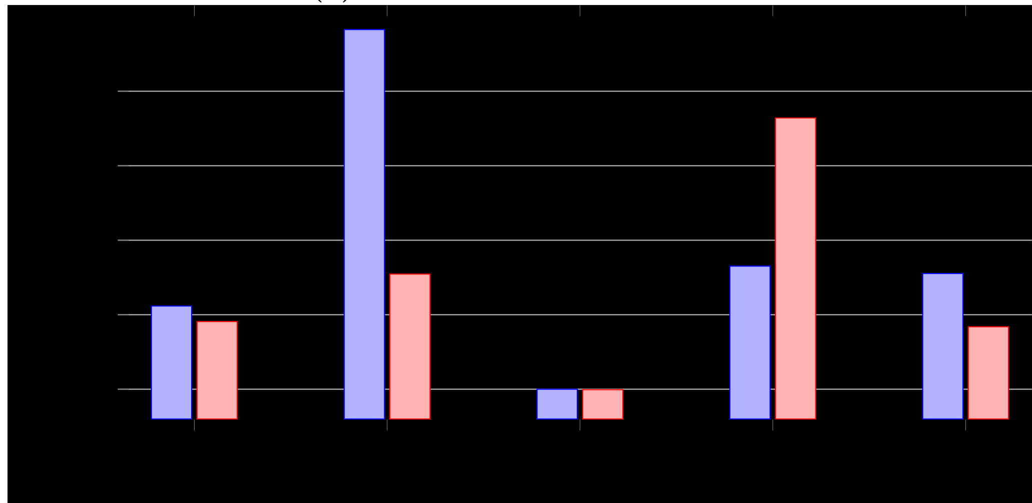


Figure 7.4: Speedup of each step in the residual evaluation on the GPU for the coarse two-dimensional mesh 2D1 (blue); and the coarse three-dimensional mesh 3D1 (red).



(a) Two dimensions: mesh 2D4



(b) Three dimensions: mesh 3D3

Figure 7.5: Speedup of each step in the residual evaluation due to renumbering of nodes, edges and faces to obtain better cache performance. The speedup of the GPU implementation is in blue, and the speedup of the CPU implementation is in red.

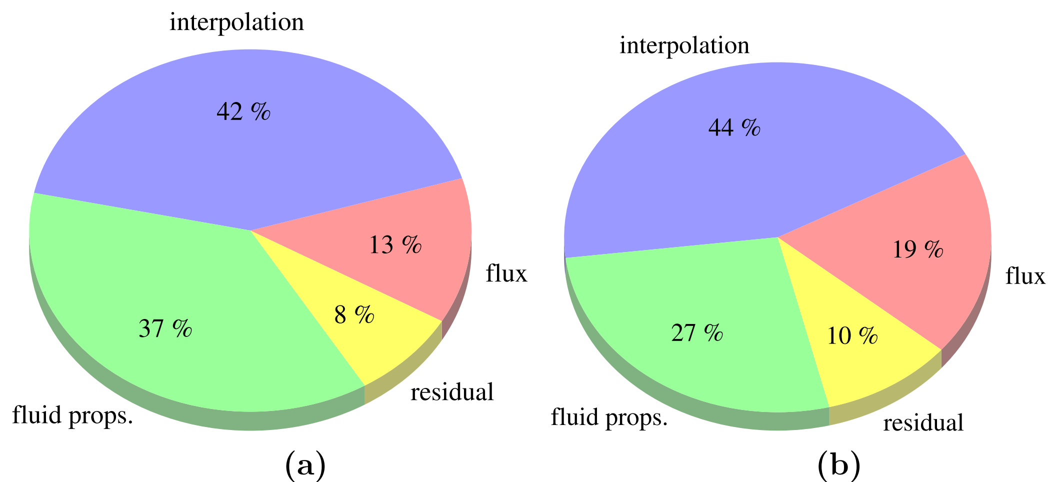


Figure 7.6: Proportion of time spent in each step of the residual evaluation on the GPU for the 2D4 mesh. (a) Without the renumbering scheme for node, edge and faces. (b) With the renumbering scheme.

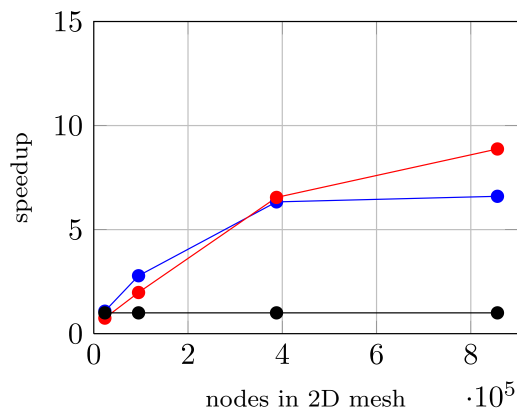


Figure 7.7: Speedup of IDA for GPU×1 and GPU×2 (blue and red respectively) relative to CPU×8 (black) as the number of nodes in the mesh increases for two dimensions.

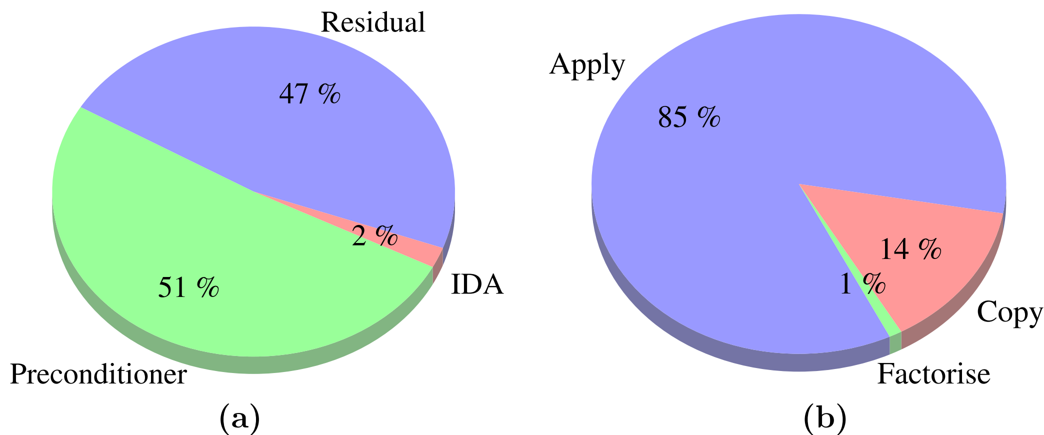


Figure 7.8: Breakdown of time spent in each part of the solver for GPU×1 and the mesh 2D4 when using the host preconditioner (a), and further breakdown of time spent in different parts of the preconditioner in this case (b).

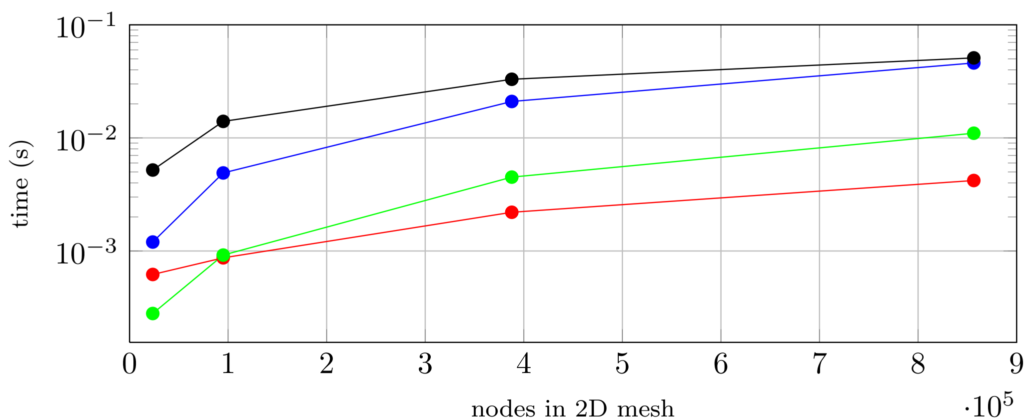


Figure 7.9: Average time to apply the local preconditioner, including any copying between host and device, as size of matrix increases for each preconditioner on one GPU (without MPI). The preconditioners are MC-ILU(0) (red), CUSP-ILU(0) (black), B-ILU(0) (green) and ILU(0) (blue).

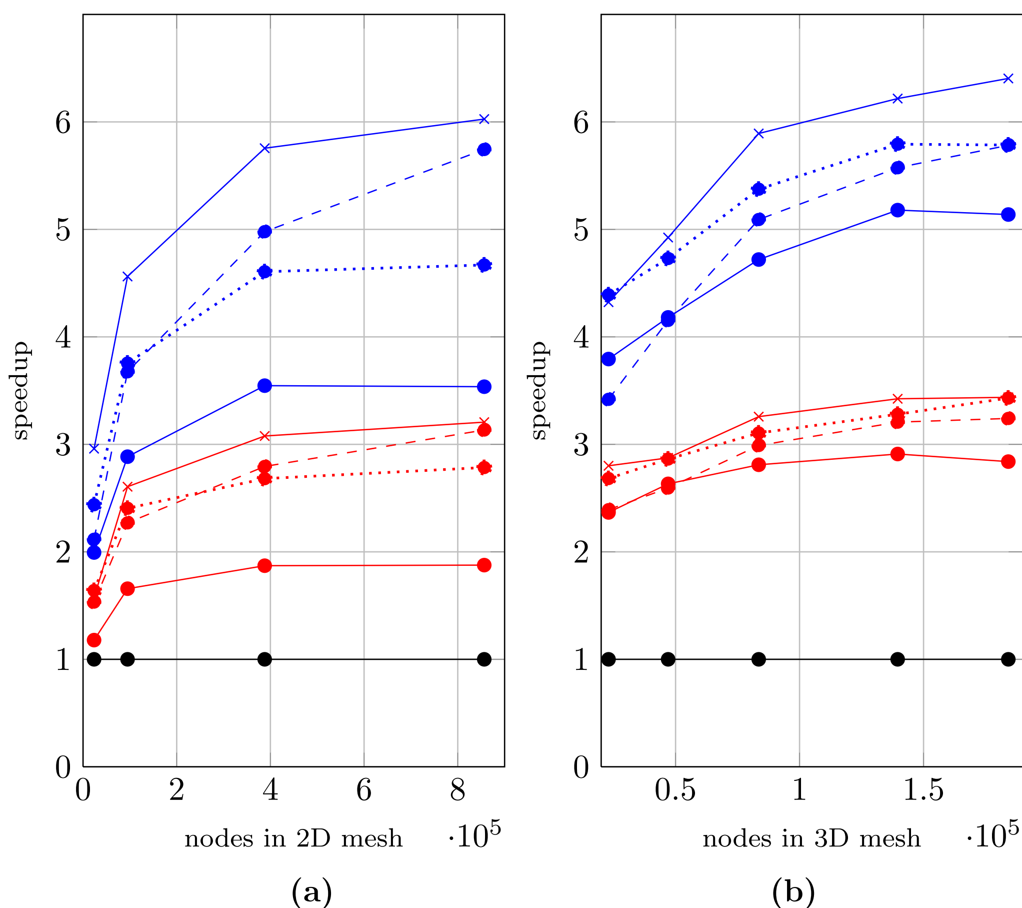


Figure 7.10: Speedup of the entire solver when using GPUs compared to CPU×8 as the number of nodes in the mesh increases for (a) two dimensions and (b) three dimensions. GPU×1 =red, GPU×2 =blue. ILU(0) = solid, B-ILU(0) =dotted, MC-ILU(0) =dashed. The speedup of the residual evaluation alone for each case is marked by the solid lines with ⨯ markers.

7.3 Conclusions

In this chapter the computational performance of the CPU and GPU implementations of FVMPor was investigated. The *dry_infiltration* test case was used for testing in both two dimensions and three dimensions, because it was possible to solve efficiently using both the block Jacobi and ILU(0) preconditioners.

First, the performance of FVMPor was investigated on the CPU, with a single core assigned to each MPI process. The strong scaling of the solution process was very good, despite a small loss of performance for more than 4 cores caused by a loss in accuracy of the block Jacobi preconditioner, and due to problems with memory affinity. The main bottleneck for the 8-core CPU simulations was computation of the residual function, which accounted for at least 80% of the total time to solution.

The next set of tests investigated the performance of the GPU implementation relative to the 8-core CPU implementation. The residual evaluation performed very well on the GPU, achieving a speedup of 3 on one GPU relative to the 8 CPU cores, and speedup of 6 when two GPUs were used. The level 1 BLAS vector operations in IDA also performed very well relatively to the baseline results.

This meant that the preconditioner became the bottleneck for the GPU implementation, taking 54% of the total time to solution, compared to only 10% on the CPU. However, a very interesting observation was made: the preconditioner is dominated by the cost of applying the preconditioner, with only 1% of time spent forming the preconditioner.

Then several different methods for applying the preconditioner on the host and device were tested. Of these, the B-ILU(0) method that used OpenMP to form and apply a block Jacobi preconditioner for the local matrix on the host was found to be the most efficient approach for small systems with less than 100,000 nodes. For larger problems, the MC-ILU(0) preconditioner,

which applied an ILU(0) preconditioner on the device using a multi-colouring algorithm was the best performer, with a speedup of 11 over the ILU(0) method, and 2.2 over the B-ILU(0) preconditioner.

By applying each step of the implicit time stepping scheme on the GPU, we achieved very good speedup across the entire solution process, with 6 times speedup for two GPUs relative to 8 CPU cores on large problems.

The results presented here come with a caveat: we are only able to obtain parallel speedup over the entire solution process, both coarse and fine-grained, for unsaturated flow problems for which the block Jacobi and ILU(0) preconditioners are well suited. However, the GPU implementation is still much preferable to the CPU implementation for saturated problems, despite having to use ILUT and LU preconditioners that are applied on the host. For example, the numerical experiments for Zhang's flow tank experiments presented in §6.5 were performed on the GPU using the host for the full LU preconditioner on a single domain. By using the GPU for these tests, results that would took hours to compute on the CPU were obtained in 15 minutes.

Chapter 8

Conclusions

This chapter gives a summary and analysis of the findings of this research program, followed by proposals for further work based on the findings presented in this thesis.

8.1 Summary and Discussion

In Chapter 1 the objectives of this work were outlined. In this section I will restate each of these objectives and discuss how they were addressed.

Investigate a mass-conservative control-volume finite element method for modelling variably-saturated groundwater flow and contaminant transport in heterogeneous porous media

In Chapter 3, the CV-FE spatial discretisation was applied to the governing equations for variably-saturated groundwater flow and contaminant transport. It was shown that this spatial discretisation is well-suited to modelling mass balance laws in heterogeneous media because the method finds consistent fluxes at quadrature points where material properties of the porous

medium are well-defined. This avoids approximating material properties at any point in the spatial discretisation, and ensures that the scheme is locally mass conservative. Furthermore, to reduce the numerical dispersion introduced by upwind weighting on coarse meshes, the van Leer and parabolic flux limiters were applied to both the mobility term in Richards' equation and the advection terms in the transport model.

The spatial discretisation was validated in §6.2–§6.5 by reproducing solutions for both benchmark problems and laboratory experiments from the literature. The following observations were made:

- Solutions computed using upstream weighting suffered from numeric dispersion on all but very fine mesh resolutions. Flux limiting reduced the amount of numerical dispersion significantly, particularly on coarse meshes. On average, flux limited solutions were between 10% to 20% more expensive to compute than solutions with upstream weighting on the same mesh. However, because they made it possible to obtain accurate solutions on coarse meshes, the flux limiters were shown to be between five to twenty times more efficient than upstream weighting to determine solutions of equivalent accuracy.
- For the solution of Richards' equation, both the parabolic and van Leer limiters applied to the mobility term gave equivalent solutions, with the parabolic limiter being slightly more efficient. For the coupled flow and transport problems, the choice of limiter applied to the mobility term was more important, with the parabolic limiter producing more accurate and efficient solutions in §6.4. However, both limiters were equivalent in terms of both accuracy and computational overhead when applied to the advection terms in the full transport model.
- Analysis of the mass balance error for test cases where the mass balance could be determined exactly showed that our implementation of the CV-FE method is mass conservative for both Richards' equation and the full transport model.

Extend the mixed formulation for Richards' equation due to Kees and Miller (2002) to the CV-FE method and coupled contaminant transport

The CV-FE discretisation was applied to the modified mixed formulation of Richards' equation proposed by Kees and Miller (2002), then extended to the coupled contaminant transport equations. The formulation imposes an additional algebraic equation for each partial differential equation in the system, which doubles the size of the nonlinear and linear systems that are solved at each time step. A Schur-complement preconditioner that used the structure of the Jacobian to effectively halve the size of the system of linear equations was proposed in §3.3.2. This preconditioning method made it possible to solve the modified mixed formulation using general higher-order implicit time stepping codes for the solution of DAE systems. The findings of the investigation into the modified mixed formulations in Chapter 6 can be summarised:

- For the solution of Richards' equation, the modified mixed MPR formulation had conservation of mass an order of magnitude more accurate than the PR formulation for the same amount of computational work for higher-order integration, and two orders of magnitude for first-order integration. The modified mixed formulation also offered superior conservation of mass for the coupled transport model in the tests performed in §6.4. However, the differences for higher-order integration were not as large for the transport model.
- The efficiency of the Schur complement preconditioner for the modified mixed formulation meant that the modified mixed formulation had a computational overhead of between 10% to 20%, despite having twice as many equations and primary variables.
- The mass balance error of solutions from the modified mixed formulations was superior to those of the PR and PC formulations, particularly for first order integration and the formulation of Richards' equation.

However, all of the formulations are mass conservative, with mass balance errors that do not grow in time. Ultimately, the choice of which formulation to use will depend on how important conservation of mass is in the final solution, and on whether first-order or higher-order temporal integration is being used.

An important motivation for the development of the modified mixed formulation is that the formulation is amenable to solution using robust packages for higher-order implicit time stepping. The IDA library (Hindmarsh et al., 2005) was used to perform the time stepping, and the role of higher-order integration and preconditioner choices was investigated in §6.7:

- Higher-order integration considerably reduced both the mass balance error, and the time to solution for a given error tolerance. Furthermore, higher-order integration was required to obtain good conservation of mass for the PR and PC formulations.
- The Newton-Krylov solver in IDA required a preconditioner to ensure timely convergence of GMRES. The choice of preconditioner was dependent on the presence of saturated conditions: for unsaturated flow and transport problems the ILU(0) and block Jacobi preconditioners performed well; whereas simulations with saturated regions required sparse factorisations with fill in, specifically the ILUT and full LU decompositions, and showed poor weak scaling of the block Jacobi preconditioner.

Implement the proposed methods in a flexible software package that can be used on a desktop computer, and scale up to run on large clusters

The software package FVMPor was developed to implement the methods developed in this thesis. It was written in C++ with a modular design, so that

components such as integrators and preconditioners can be easily changed. The integrator used in this thesis was based on IDA, and the hybrid MPI-OpenMP programming model was used to obtain both coarse-grained and fine-grained parallelism on clusters and multi-core CPUs respectively. The interface and implementation of FVMPor use the vectorlib library, which provides a syntax for memory allocation and linear algebra operations.

An important objective in the design of FVMPor was that it could be easily adapted to other models. The modularity of the software makes it simple to do this by changing the physics implementation in §5.6. FVMPor has already been used to model multiphase flow and heat transfer in biomass, and future work will see it used to develop models of coal seam gas.

To date, FVMPor has been used on a range of different computers. These include: desktop PCs; the high-performance GPU workstation used in Chapter 7; and on the GPU nodes of the Lyra cluster at the Queensland University of Technology. With the development of preconditioners better-suited to distributed problems, future versions of FVMPor will be used on larger clusters.

Investigate using GPUs to accelerate the unstructured mesh solver for groundwater flow

The vectorlib library was adapted to provide a common interface to low-level hardware implementations on the GPU and CPU of commonly-used linear algebra operations. The spatial discretisation in the residual evaluation was expressed in terms of the data parallel operations implemented in vectorlib, and some operations specific to the CV-FE method that were implemented in FVMPor. In this manner, the residual evaluation, which was the biggest computational overhead for the CPU implementation, was implemented entirely on the GPU.

To improve the performance of the residual evaluation on unstructured meshes, a novel renumbering scheme for nodes, edges and faces that improved the

correlation between the location of information in memory and in space was presented developed in §5.7. The renumbering is simple to implement, and is based on abstract analysis of the connectivity graph of the mesh, so that it can be applied to arbitrary geometries in two and three dimensions. It significantly improved the performance of the indirect indexing in the gather and scatter operations in the residual evaluation, with a speedup of between 1.4 to 1.5 on the GPU, and a speedup of 1.25 on CPU.

After the residual evaluation, the other two computational overheads in FVMPor were the vector operations in IDA, and the application of the preconditioner in GMRES. The Newton-Krylov method used by IDA was implemented on the GPU by rewriting the NVector library to support GPU implementation[1]. Then, a method for efficiently solving sparse triangular systems for which *a priori* knowledge of the sparsity pattern is available was implemented to allow the application of ILU(0) preconditioners on the GPU.

In this manner, each of the steps in the implicit time stepping method were implemented to run efficiently on the GPU. The performance of the GPU and CPU implementations of FVMPor were then tested in §7.2. The GPU version performed very well, with a total speedup of 3 times for one GPU, and 6 times for 2 GPUs over a baseline 8-core CPU version run. However, to obtain this speedup it was necessary that an efficient parallel preconditioner could be used. For the unsaturated flow problem considered in the tests, the ILU(0) preconditioner worked well.

8.2 Directions For Further Research

The outcomes of this research suggest a number of possible directions for further research, some of which are currently being pursued, which are briefly discussed here.

[1]The CUDA implementation of NVector developed in this thesis is available under an open source license (BSD) at `github.com/bencumming/NVectorCUDA`

Further investigation of preconditioning methods, both for distributed systems and on the GPU

FVMPor showed very good scaling, both coarse-grained using domain decomposition, and fine-grained on the GPU for unsaturated flow problems, where a block Jacobi ILU(0) preconditioner was very effective. However, for simulations with variably-saturated conditions, neither the ILU(0) preconditioner nor block Jacobi approaches were adequate. This limited the extent to which simulations with saturated conditions could be parallelised.

To model saturated conditions more efficiently in parallel, appropriate preconditioners need to be investigated. One possibility for fine-grained parallelism of the local block on the GPU is the recent work by Heuveline et al. (2011a,b), who have shown good results for the GPU implementations of sparse factorisations with fill-in on the GPU. For coarse-grained parallelism, methods that introduce coupling between sub-domains, such as Schwartz and Schur complement approaches, will need to be investigated.

Explicit methods for time stepping stiff initial value problems

Implicit methods are preferred for solving Richards' equation and more detailed treatments of multiphase flows, due to the stability and large time step sizes that they allow given the stiffness of the governing equations. Explicit methods are generally simpler to implement than implicit methods, by virtue of not requiring the solution of a nonlinear system of equations (which in turn requires the solution of linear systems), at each time step. However, explicit methods typically take many more time steps than implicit methods for stiff problems, justifying the extra effort required to implement implicit methods.

Typically, the biggest computational overhead for both implicit and explicit methods is performing residual evaluations – for the Newton-Krylov approach taken in this work, residual evaluation was the dominant cost of the CPU implementation. However, the GPU implementation of the residual evalua-

tion was very efficient, and the cost of applying the preconditioner became a significant overhead.

Recent research by Carr et al. (2011) investigated using an exponential Euler method against using second and fifth order BDFs in IDA. The authors found that the method was outperformed the second order BDFs, however was not as not as efficient as the fifth-order solver. However, given that the exponential Euler approach does not require a preconditioner, it is worth investigating its performance on the GPU.

Investigation of More Realistic Boundary Conditions

Practical applications of software like that developed here involve a wide variety of boundary conditions, which are often time varying, and may involve changing the type of boundary condition used. The simulations that involved time-varying tidal level in §6.5 showed the implicit time stepping method struggled when changing from a Dirichlet boundary condition to a no flux boundary condition. For the software to be more useful, a flexible system for implementing a broad range of boundary conditions is required, as is further investigation into how best to implement them efficiently in the chosen time stepping framework.

Further development of vectorlib library

The vectorlib library is currently being rewritten to be more flexible and efficient. The implementation of vectorlib used in this thesis requires that a CUDA kernel is written and compiled separately for any operation performed on the GPU. The new version of vectorlib uses the Thrust library (NVIDIA, 2011d) with lazy evaluation to implement arbitrary GPU operations at compile time. Thrust is a C++ template library for CUDA based on the Standard Template Library (STL) that also provides data parallel operations such as scan, sort and reductions.

The new vectorlib library will also include better support for threading and vectorisation for data stored on the CPU, and built in support for MPI communication.

Develop a three-dimensional model for Zhang's experiments

The laboratory experiments of Zhang (2000) have been investigated numerically, both by the original researchers (Zhang et al., 2002, Volker et al., 2002, Zhang et al., 2004) and more recently by Brovelli et al. (2007) and in this thesis. Although some features of the experiment have been reproduced reasonably well in the numerical investigations, there are other features such as the density-driven fingering, and the formation of a diffuse contaminant region near the beach, that have yet to be reproduced. The results presented in this thesis suggest that to reproduce these features it may be necessary to perform three-dimensional simulations, that explicitly model heterogeneity, uneven injection of the contaminant at the surface, and the formation of a transient seepage face at the beach.

Appendix A

Computing Derivative Coefficients

In the derivation of the PR and PC formulations, for Richards' equation and coupled flow and transport respectively, the accumulation terms in the governing partial differential equations are differentiated with respect to the primary variables in each formulation. This expresses the accumulation term as a function of the primary variables and their derivatives. In this appendix we derive functional forms for the coefficients of the derivatives in the accumulation terms.

A.1 The PR formulation of Richards' Equation

The chain rule is first applied to the accumulation term in Richards' equation (2.1) to express it as a function of pressure head ψ and its derivative, as follows

$$\frac{\partial(\rho\theta)}{\partial t} = \frac{\partial(\rho\theta)}{\partial\psi}\frac{\partial\psi}{\partial t}. \tag{A.1}$$

Thus, the coefficient of the derivative for the PR formulation is the storage term $n(\psi)$, which is defined

$$n(\psi) = \frac{\partial(\rho\theta)}{\partial\psi}. \tag{A.2}$$

By substituting the equation for moisture content (2.13) into (A.2), the storage term can be expanded as follows

$$\begin{aligned} &= \theta\frac{\partial\rho}{\partial\psi} + \rho\left[S_w\frac{\partial\phi}{\partial\psi} + \phi\frac{\partial S_w}{\partial\psi}\right] \\ &= \rho_0\beta_\psi\theta + \rho\left[\alpha_\psi S_w + \phi\frac{\partial S_w}{\partial\psi}\right], \end{aligned} \tag{A.3}$$

where the coefficients α_ψ and β_ψ are defined

$$\alpha_\psi = (1-\phi_0)\alpha\rho_0 g, \tag{A.4}$$

$$\beta_\psi = \rho_0 g\beta. \tag{A.5}$$

The analytic forms for each of the terms in (A.3) can be found using the constitutive relationships defined in equations (2.8), (2.9) and (2.13). Finally, we note that the storage term in (A.3) is dependent on material properties, and care must be taken to use the appropriate formula (3.25) when finding the volume average of the storage term.

A.2 The PC Formulation of the Full Transport model

For the PC formulation, the chain rule is applied to the accumulation terms for Richards' equation (2.1) and the solute mass balance equation (2.5). First,

take the accumulation term for Richards' equation:

$$\frac{\partial(\rho\theta)}{\partial t} = \frac{\partial(\rho\theta)}{\partial\psi}\frac{\partial\psi}{\partial t} + \frac{\partial(\rho\theta)}{\partial c}\frac{\partial c}{\partial t}. \tag{A.6}$$

The first coefficient in (A.6) is the storage term in from (A.3)

$$\frac{\partial(\rho\theta)}{\partial\psi} = \rho_0\beta_\psi\theta + \rho\left[\alpha_\psi S_w + \phi\frac{\partial S_w}{\partial\psi}\right], \tag{A.7}$$

The expression for the coefficient in front of the concentration derivative in (A.6) is found using (2.11) and by noting that moisture content is not dependent on concentration:

$$\begin{aligned}\frac{\partial(\rho\theta)}{\partial c} &= \theta\frac{\partial\rho}{\partial c}\\ &= \rho_0\eta\theta.\end{aligned} \tag{A.8}$$

Applying the chain rule to the accumulation term of the solute mass balance equation (2.5) gives

$$\frac{\partial(c\theta)}{\partial t} = \frac{\partial(c\theta)}{\partial\psi}\frac{\partial\psi}{\partial t} + \frac{\partial(c\theta)}{\partial c}\frac{\partial c}{\partial t}. \tag{A.9}$$

The first coefficient in (A.9) can be simplified as follows

$$\begin{aligned}\frac{\partial(c\theta)}{\partial\psi} &= c\frac{\partial\theta}{\partial\psi}\\ &= c\left(\alpha_\psi S_w + \phi\frac{\partial S_w}{\partial\psi}\right),\end{aligned}$$

where α_ψ and β_ψ are defined in (A.4) and (A.5). We introduce the term $a(\psi) = \partial\theta/\partial\psi$ to simplify the notation of the PC formulation as follows

$$\frac{\partial(c\theta)}{\partial\psi} = ca(\psi). \tag{A.10}$$

Finally, it is simple to show that the coefficient of the concentration derivative

in (A.9) is the moisture content

$$\frac{\partial(c\theta)}{\partial c} = \theta. \tag{A.11}$$

Appendix B

Verification of Hydrostatic Boundary Condition

In this appendix the mixed beach boundary condition defined in §2.3 is validated. The boundary condition, defined in equation (2.23), is a Cauchy condition of the form

$$\boldsymbol{q}_j \cdot \boldsymbol{n}_j = R_b(\psi_i - \psi_b(z,t)). \tag{B.1}$$

The prescribed pressure head at the boundary, ψ_b, is dictated by the height of the tide, namely

$$\psi_b(z,t) = (1+\eta)(h(t) - z, \tag{B.2}$$

where $h(t)$ is the height of the tide at time t. The penalty term R_b in (B.1) is chosen as follows

$$R_b = \begin{cases} 0, & z > h(t) \\ \frac{K}{L}, & z \leq h(t) \end{cases}, \tag{B.3}$$

where K is the hydraulic conductivity and L is the length coupling scale (Chui and Freyberg, 2009). For points above the tide, a no flow boundary condition is imposed, that is $\boldsymbol{q}_j \cdot \boldsymbol{n}_j = 0$. When evaluated at control volume faces below the level of the tide, there is flux over the interface. This approach can also

be thought of as introducing a source term to the control volume (Forsyth and Kropinski, 1997). If the parameter R_b is large, the Dirichlet boundary condition is strictly enforced. Whereas, if the value of R_b is lowered, the Dirichlet boundary condition is not strictly enforced.

We will now present a simple numerical investigation into the accuracy of the mixed beach boundary condition, and its sensitivity to the length scale parameter L. The test case is based on the flow tank experiments performed by Zhang (2000), that are described in §6.1.5. The experiment is similar to the *tank_tidal_plume* test case, except no contaminant plume is injected, so that we can focus on the tidal fluctuations. The height of the tide was fixed for the first 170 minutes, to allow the steady state sea water interface to form, before the sea level was varied according to Figure 6.4.

Figure B.1(a) shows the effect of varying the parameter L (between 0.01 to 100) on the accuracy of the computed pressure head values, compared to the prescribed pressure values, at the boundary. The solution is more accurate for smaller values of L (with corresponding increase in R_b). For the first 170 minutes, when the prescribed values are held constant, the error is constant for each case. Then, as the prescribed pressure head values are varied, the error also varies, however the error is still reduced by using smaller values of L. In our tests, it was found that if L was decreased even further, these fluctuations in the error became larger, and the Newton iterations failed to converge in some cases. The more accurate solutions found using smaller values of L also required considerably more computational effort, as illustrated in Figure B.1(b).

It is not pictured here, however the shape of the sea water interface is dependent on the accuracy of the condition at the boundary. However, as the length coupling parameter decreases, the solution converges such that the interfaces computed using $L = 0.1$ and $L = 0.01$ are indistinguishable from those computed using the fixed Dirichlet condition, or type 2, boundary condition for the constant sea level case.

Based on these observations, a value of $L = 0.1$ is chosen for our tests, because it enforces the pressure head at the boundary to agree with the prescribed value to three significant digits. This corresponds to the relative error tolerance of $\tau_r = 1\text{e-}3$ used in the numerical experiments in §6.5, while giving stable and reasonably efficient computation times.

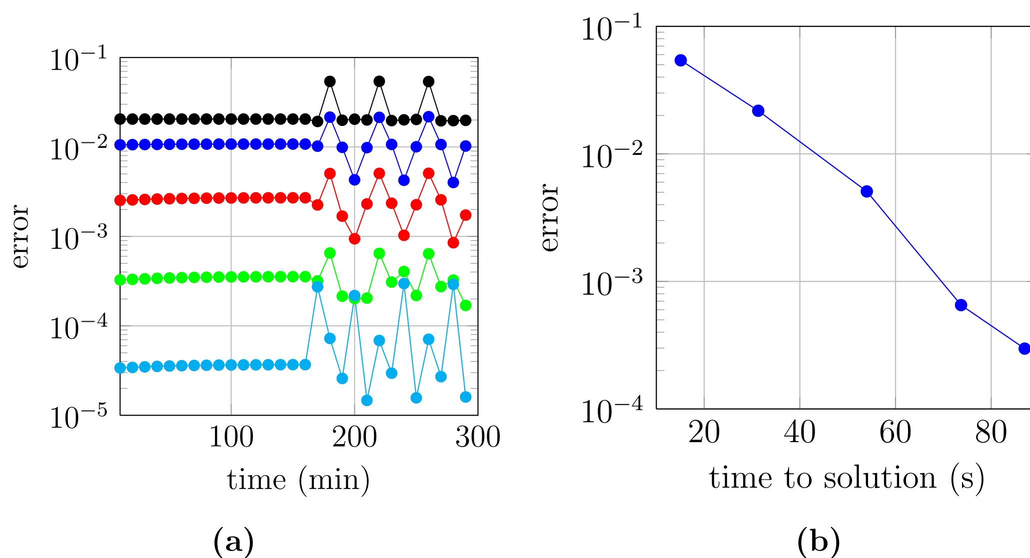


Figure B.1: (a) The maximum error at the boundary over time for different values of the scaling parameter, labelled as: $L = 100$–black; $L = 10$–blue; $L = 1$–red; $L = 0.1$–green; $L = 0.01$–cyan. (b) The maximum error over the course of a simulation as a function of work. The work is increased, and the error decreased, by decreasing the value of L.

Appendix C

Shape Function Interpolation For Quadrilaterals and Hexahedra

The shape function interpolation weights for triangle and tetrahedral elements used in FVMPor were derived in §3.1.3. Here we derive additional weights for quadrilateral and hexahedral elements.

C.1 Shape function interpolation on quadrilaterals

For a quadrilateral element we use a bilinear interpolation function

$$s_\ell(\boldsymbol{x}) = \alpha_1 + \alpha_2 x + \alpha_3 y + \alpha_4 xy, \tag{C.1}$$

which gives a coefficient matrix, as in (3.10), of the form

$$C_{\epsilon_\ell} = \begin{bmatrix} 1 & x_1 & y_1 & x_1y_1 \\ 1 & x_2 & y_2 & x_2y_2 \\ 1 & x_3 & y_3 & x_3y_3 \\ 1 & x_4 & y_4 & x_4y_4 \end{bmatrix}. \tag{C.2}$$

Using the same arguments as (3.11) and (3.12), the shape functions and their derivatives are defined by

$$\boldsymbol{N}(\boldsymbol{x}) = \begin{bmatrix} 1 & x & y & xy \end{bmatrix} C_{\epsilon_\ell}^{-1} \tag{C.3a}$$

$$\frac{\partial \boldsymbol{N}}{\partial x} = \begin{bmatrix} 0 & 1 & 0 & y \end{bmatrix} C_{\epsilon_\ell}^{-1}, \tag{C.3b}$$

$$\frac{\partial \boldsymbol{N}}{\partial y} = \begin{bmatrix} 0 & 0 & 1 & x \end{bmatrix} C_{\epsilon_\ell}^{-1} \tag{C.3c}$$

where we note that the derivatives are linear functions on ϵ_ℓ.

C.2 Shape function interpolation on hexahedra

For a hexahedral element a trilinear interpolation function of the form

$$s_\ell(\boldsymbol{x}) = \alpha_1 + \alpha_2 x + \alpha_3 y + \alpha_4 z + \alpha_5 xy + \alpha_6 xz + \alpha_7 yz + \alpha_8 xyz, \tag{C.4}$$

is used. The coefficient matrix in this case is

$$C_{\epsilon_\ell} = \begin{bmatrix} 1 & x_1 & y_1 & z_1 & x_1y_1 & x_1z_1 & y_1z_1 & x_1y_1z_1 \\ \vdots & & & & \ddots & & & \\ 1 & x_8 & y_8 & z_8 & x_8y_8 & x_8z_8 & y_8z_8 & x_8y_8z_8 \end{bmatrix}, \tag{C.5}$$

and the shape functions and their derivatives are defined:

$$\boldsymbol{N}(\boldsymbol{x}) = \begin{bmatrix}1 & x & y & z & xy & xz & yz & xyz\end{bmatrix} C_{\epsilon_\ell}^{-1} \tag{C.6a}$$

$$\frac{\partial \boldsymbol{N}}{\partial x} = \begin{bmatrix}0 & 1 & 0 & 0 & y & z & 0 & yz\end{bmatrix} C_{\epsilon_\ell}^{-1} \tag{C.6b}$$

$$\frac{\partial \boldsymbol{N}}{\partial y} = \begin{bmatrix}0 & 0 & 1 & 0 & x & 0 & z & xz\end{bmatrix} C_{\epsilon_\ell}^{-1} \tag{C.6c}$$

$$\frac{\partial \boldsymbol{N}}{\partial z} = \begin{bmatrix}0 & 0 & 0 & 1 & 0 & x & y & xy\end{bmatrix} C_{\epsilon_\ell}^{-1} \tag{C.6d}$$

where, as was the case for quadrilateral elements, the gradient weights vary linearly.

Appendix D

Transient Seepage Faces in IDA

Seepage boundary conditions are not implemented in this thesis due to limitations imposed by the IDA library. If the location of a seepage face is transient, its location at each time step has to be determined using an iterative procedure (Cooley, 1983). The iterative process first estimates the location of the seepage face, and then finds the solution for the given seepage face location. The solution is rejected if there is flux into the domain over any part of the seepage face, in which case the time step is repeated with a new estimate for the location of the seepage face. The process is repeated until flux everywhere on the seepage face is out of the domain. Such an iterative process is not possible in IDA, which does not provide any mechanism for rejecting a time step once it has been completed.

Appendix E

The Iteration Matrix G Under Saturated Conditions

We now give a brief analysis of the properties of the iteration matrix G under saturated and unsaturated flow conditions. Consider the iteration matrix G for the PR formulation, which is found by substituting the residual function $\boldsymbol{f}(t, \boldsymbol{\psi}, \boldsymbol{\psi}')$ in (3.70) into the equation (3.90) as follows

$$\begin{aligned} G &= \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\psi}} + \alpha \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\psi}'} \\ &= A + \alpha D, \end{aligned} \tag{E.1}$$

where the matrix $A = \partial \boldsymbol{f}/\partial \boldsymbol{y}$, the positive constant $\alpha = \alpha_0/\tau$ defined in (3.87) is the ratio of the leading BDF coefficient α_0 and the time step size τ, and the diagonal matrix D has diagonal entries defined as follows

$$d_{ii} = n_i(\psi_i). \tag{E.2}$$

It is straightforward to show that the storage term $n_i(\psi_i)$ in (E.2) satisfies

the following constraints:

$$
\begin{aligned}
n_i &> 0, \qquad \psi_i < 0 \qquad \text{(unsaturated conditions)} \\
n_i &\approx 0, \qquad \psi_i \geq 0 \qquad \text{(saturated conditions)}
\end{aligned}
$$

Thus, under unsaturated conditions, the term αD in (E.1) makes a positive contribution to the diagonal entries of G that is inversely proportional to the time step size τ[1].

It can be shown that for upstream weighting the diagonal entries of A in (E.1) are positive, and the off-diagonal entries are negative. Thus, the iteration matrix can be an M-matrix by adding positive values to the diagonal, which is possible under unsaturated conditions by choosing an appropriate time step size. In practice, the adaptive time stepping method used by IDA automatically chooses the time step sizes that lead to well conditioned iteration matrices that can be solved efficiently using the GMRES method with an ILU(0) preconditioner.

However, when saturated conditions arise the contribution to the diagonal is either zero or negligibly small, and it is not possible to improve the conditioning of the iteration matrix by decreasing the time step size. In these circumstances, it was found that the ILU(0) preconditioner does not guarantee timely convergence of the GMRES iterations, and more robust method such as ILUT are required (see §6.7.1).

[1] Note that the expression for the iteration matrix in (E.1) is equivalent to the Schur complement for the MPR formulation in (3.102).

Bibliography

P. Arminjon and A. Dervieux. Construction of TVD-like artificial viscosities on two-dimensional arbitrary FEM grids. *Journal of Computational Physics*, 106:176–198, 1993.

B. Ataie-Ashtiani. *Contaminant transport in coastal aquifers.* PhD thesis, University of Queensland, Brisbane, Australia, 1997.

J. Bear. *Hydraulics of Groundwater.* McGraw-Hill, Inc., New York, 1979.

J. Bear and A. Cheng. *Modeling Groundwater Flow and Contaminant Transport*, volume 23 of *Theory and Applications in Transport in Porous Media.* Springer, Dordrecht, Holland, 2010.

J. Bear and A. Verruijt. *Modeling Groundwater Flow and Pollution.* D. Reidel Publishing Company, Dordrecht, Holland, 1987.

M.C. Boufadel, Y. Xia, and H. Li. Modeling solute transport and transient seepage in a laboratory beach under tidal influence. *Environmental Modelling & Software*, 26(7):899–912, 2011.

K.E. Brenan, S.L. Campbell, and L.R. Petzold. *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, volume 14 of *Classics In Applied Mathematics.* SIAM, 1996.

R.H. Brooks and A.T. Corey. Properties of porous media affecting fluid flow. *Journal of the Irrigation and Drainage Division*, 92:61–87, 1966.

A. Brovelli, X. Mao, and D.A. Barry. Numerical modeling of tidal influence on density-dependent contaminant transport. *Water Resources Research*, 40:W10426, 2007.

X.C. Cai, W.D. Gropp, and D.E. Keyes. A comparison of some domain decomposition and ILU preconditioned iterative methods for nonsymmetric elliptic problems. *Numerical Linear Algebra with Applications*, 1(5): 477–504, 1994.

E.J. Carr, T.J. Moroney, and I.W. Turner. Efficient simulation of unsaturated flow using exponential time integration. *Applied Mathematics and Computation*, 217(14):6587–6596, 2011.

M.A. Celia, E.T. Bouloutas, and R.L. Zarba. A general mass-conservative solution for the unsaturated flow equation. *Water Resources Research*, 26: 1483–1496, 1990.

T.F. Chan and T.P. Mathew. Domain decomposition algorithms. *Acta Numerica*, 3:61–143, 1994.

T.F.M. Chui and D.L. Freyberg. Implementing hydrologic boundary conditions in a multiphysics model. *Journal of Hydrologic Engineering*, 14(12): 1374–1377, 2009.

R.L. Cooley. Some new procedures for numerical solution of variably saturated flow problems. *Water Resources Research*, 19:1271–1285, 1983.

A. Corrigan, F.F. Camelli, R. Löhner, and J. Wallin. Running unstructured grid-based CFD solvers on modern graphics hardware. *International Journal For Numerical Methods In Fluids*, published online, 2010.

B.D. Cumming, T. Moroney, and I.W. Turner. A mass-conservative control volume-finite element method for solving Richards' equation in heterogeneous porous media. *BIT Numerical Mathematics*, 51(4):845–864, 2011.

E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

M. de la Asuncin, J.M. Mantas, M.J. Castro, and E.D. Fernndez-Nieto. An MPI-CUDA implementation of an improved roe method for two-layer shallow water systems. *Journal of Parallel and Distributed Computing*, In Press, Accepted Manuscript, 2011.

H.J.G. Diersch and O. Kolditz. Variable-density flow and transport in porous media: approaches and challenges. *Advances in Water Resources*, 25(8-12): 899–944, 2002.

H.J.G. Diersch and P. Perrochet. On the primary variable switching technique for simulating unsaturated-saturated flows. *Advances in Water Resources*, 23(3):271–301, 1999.

R.E. Ewing, T. Lin, and Y. Lin. On the accuracy of the finite volume element method based on piecewise linear polynomials. *SIAM Journal on Numerical Analysis*, 39(6):1865–1888, 2002.

M. Fahs, A. Younes, and F. Lehmann. An easy and efficient combination of the mixed finite element method and the method of lines for the resolution of Richards' equation. *Environmental Modelling & Software*, 24(9):1122–1126, 2009.

M.W. Farthing, C.E. Kees, and C.T. Miller. Mixed finite element methods and higher order temporal approximations for variably saturated groundwater flow. *Advances in Water Resources*, 26(4):373–394, 2003.

J.H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. Springer, Berlin, 3rd edition, 2002.

P.A. Forsyth. A control volume finite element approach to NAPL groundwater contamination. *SIAM Journal on Scientific and Statistical Computing*, 12(5):1029–1057, 1991.

P.A. Forsyth and M.C. Kropinski. Monotonicity considerations for saturated–unsaturated subsurface flows. *SIAM Journal on Scientific Computing*, 18 (5):1328–1354, 1997.

P.A. Forsyth, Y.S. Wu, and K. Pruess. Robust numerical methods for saturated–unsaturated flow with dry initial conditions in heterogeneous media. *Advances in Water Resources*, 18:25–38, 1995.

P.A. Forsyth, A.J.A. Unger, and E.A. Sudicky. Nonlinear iteration methods for nonequilibrium multiphase subsurface flow. *Advances in Water Resources*, 21:433–449, 1996.

C. Geuzaine and J.F. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S.H.M. Buijssen, M. Grajewski, and S. Turek. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Computing*, 33(10-11):685–699, 2007.

M. Harris. Optimising parallel reduction in CUDA. NVIDIA developer website: www.developer.download.nvidia.com, 2007.

V. Heuveline, D. Lukarski, N. Trost, and J. Weiss. Parallel smoothers for matrix-based multigrid methods on unstructured meshes using multicore CPUs and GPUs. EMCL Preprint Series, 2011a.

V. Heuveline, D. Lukarski, and J. Weiss. Enhanced parallel ILU(p)-based preconditioners for multi-core CPUs and GPUs – the power(q)-pattern method. EMCL Preprint Series, 2011b.

A.C. Hindmarsh, P.N. Brown, K.E. Grant, S.L. Lee, R. Serban, D.E. Shumaker, and C.S. Woodward. SUNDIALS: Suite of nonlinear and differetial/algebraic equation solvers. *ACM Transactions on Mathematical Sciences*, 31(3):363–396, 2005.

M.G. Hodnett and J. Tomasella. Marked differences between van Genuchten soil water-retention parameters for temperate and tropical soils: a new

water-retention pedo-transfer functions developed for tropical soils. *Geoderma*, 108(3-4):155–180, 2002.

Intel. *Intel MPI Library for Linux OS Reference Manual*, 2008.

Intel. *Intel Math Kernel Library User's Guide*, 2010.

G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20 (1):359–392, 1999.

C.E. Kees and C.T. Miller. C++ implementations of numerical methods for solving differential-algebraic equations: design and optimization considerations. *ACM Trans. Math. Softw.*, 25:377–403, 1999.

C.E. Kees and C.T. Miller. Higher order time integration methods for two-phase flow. *Advances in Water Resources*, 25(2):159–177, 2002.

C.E. Kees, M.W. Farthing, and C.N. Dawson. Locally conservative, stabilized finite element methods for variably saturated flow. *Computer Methods in Applied Mechanics and Engineering*, 197(51-52):4610–4625, 2008.

C.T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*, volume 16 of *Frontiers in Applied Mathematics*. SIAM, Philadelphia, 1995.

C.T. Kelley, C.T. Miller, and M.D. Tocci. Termination of Newton/chord iterations and the method of lines. *SIAM Journal on Scientific Computing*, 19(1):280–290, 1998.

Khronos Group. *OpenCL Reference Pages*, 2010.

A. Klöckner, T. Warburton, J. Bridge, and J.S. Hesthaven. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, 228(21):7863–7882, 2009.

D. Komatitsch, G. Erlebacher, D. Gddeke, and D. Micha. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *Journal of Computational Physics*, 229(20):7692–7714, 2010.

C.D. Langevin and W. Guo. MODFLOW/MT3DMS-based simulation of variable-density ground water flow and transport. *Ground Water*, 44(3): 339–351, 2006.

Levinthal, D. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors*. Intel, 2009.

R. Li and Y. Saad. GPU-accelerated preconditioned iterative linear solvers. Technical Report umsi-2010-112, MSI, uofmad, 2010.

F. Liu, I. Turner, and V. Anh. An unstructured mesh finite volume method for modelling saltwater intrusion into coastal aquifers. *Journal of Applied Mathematics and Computing*, 9:391–407, 2002.

F. Liu, V.V. Anh, I. Turner, K. Bajracharya, W.J. Huxley, and N. Su. A finite volume simulation model for saturated-unsaturated flow and application to Gooburrum, Bundaberg, Queensland, Australia. *Applied Mathematical Modelling*, 30:352–366, 2006.

M.J. Martinez. Comparison of Galerkin and control volume finite element for advection-diffusion problems. *International Journal for Numerical Methods in Fluids*, 50(3):347–376, 2006.

C.T. Miller, G.A. Williams, C.T. Kelley, and M.D. Tocci. Robust solution of Richards' equation for nonuniform porous media. *Water Resources Research*, 34(10):2599–2610, 1998.

T.J. Moroney and S.L. Truscott. A three-dimensional finite volume method for modelling saltwater intrusion in coastal aquifers. *ANZIAM Journal*, Pending Publication, 2008.

T.J. Moroney and I.W. Turner. A finite volume method based on radial basis functions for two-dimensional nonlinear diffusion equations. *Applied Mathematical Modelling*, 30(10):1118–1133, 2006.

Y. Mualem. A new model for predicting the hydraulic conductivity of unsaturated porous media. *Water Resources Research*, 12:513–522, 1976.

M. Naumov. Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS. White Paper, 2011.

L.E. Neumann, J. Simunek, and F.J. Cook. Implementation of quadratic upstream interpolation schemes for solute transport into HYDRUS-1D. *Environmental Modelling & Software*, 26(11):1298–1308, 2011.

NVIDIA. NVIDIA's next generation CUDA compute architecture: Fermi. White Paper, 2009.

NVIDIA. *NVIDIA CUDA C Programming Guide Version 4.0*, 2011a.

NVIDIA. *CUDA Toolkit 4.0 CUBLAS Library*, 2011b.

NVIDIA. *CUDA CUSPARSE Library*, 2011c.

NVIDIA. *Thrust Quickstart Guide*, 2011d.

P.S. Pacheco. *Parallel Programming With MPI*. Morgan Kaufmann Publishers, 1997.

S.V. Patankar. *Numerical Heat Transfer and Fluid Flow*. Hemisphere Publishing Corporation, 1980.

D.W. Pepper and J.C. Heinrich. *The Finite Element Method*. Taylor & Francis, Boca Raton, FL, 2nd edition, 2006.

P. Perré and I.W. Turner. A heterogeneous wood drying computational model that accounts for material property variation across growth rings. *Chemical Engineering Journal*, 86:117–131, 2002.

K. Pruess. The TOUGH codes – a family of simulation tools for multiphase flow and transport processes in permeable media. *Vadose Zone Journal*, 3:738–746, 2001.

A. Quarteroni and A. Vialli. *Domain Decomposition Methods for Partial Differential Equations*. Oxford University Press, Oxford, England, 1999.

S. Rostrup and H. De Sterck. Parallel hyperbolic PDE simulation on clusters: Cell versus GPU. *Computer Physics Communications*, 181(12):2164–2179, 2010.

Y. Saad. ILUT: A dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1(4):387–402, 1994.

Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Berlin, 2 edition, 2000.

Y. Saad and M.H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

Y. Saad and M. Sosonkina. pARMS: A package for the parallel iterative solution of general large sparse linear systems user's guide. Technical Report UMSI2004-8, msi, uofmad, 2004.

O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, 20(3):475–487, 2004.

R.A. Schincariol, F.W. Schwartz, and C.A. Mendoza. On the generation of instabilities in variable density flow. *Water Resources Research*, 30: 913–927, 1994.

L.F. Shampine. Implementation of implicit formulas for the solution of ODEs. *SIAM Journal on Scientific and Statistical Computing*, 1(1):103–118, 1980.

J. Simunek, M.Th. van Genuchten, and M. Sejna. Development and applications of the HYDRUS and STANMOD software packages and related codes. *Vadose Zone Journal*, 7:587–600, 2008.

B. Stroustrup. *The C++ programming language*. Adisson-Wesley, Reading, MA, 1993.

P. Sweby. High resolution schemes using flux limiters for hyperbolic conservation laws. *SIAM Journal on Numerical Analysis*, 21(5):995–1011, 1984.

R. Therrien, R.G. McLaren, E.A. Sudicky, and S.M. Panday. *HydroGeo-Sphere – A Three-dimensional numerical model describing fully-integrated subsurface and surface flow and solute transport*. Groundwater Simulations Group, 2010.

M.D. Tocci, C.T. Kelley, and C.T. Miller. Accurate and economical solution of the pressure-head form of Richards' equation by the method of lines. *Advances in Water Resources*, 20(1):1–14, 1997.

M.G. Trefry and C. Muffels. FEFLOW: A finite-element ground water flow and transport modeling tool. *Ground Water*, 45(5):525–528, 2007.

S. Truscott. *A Heterogeneous Three-Dimensional Computational Model for Wood Drying*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2004.

I.W. Turner and P. Perré. The use of implicit flux limiting schemes in the simulation of the drying process: A new maximum flow sensor applied to phase mobilities. *Applied Mathematical Modelling*, 25:513–540, 2001.

A.J.A. Unger, P.A. Forsyth, and E.A. Sudicky. Variable spatial and temporal weighting schemes for use in multi-phase compositional problems. *Advances in Water Resources*, 19(1):1–27, 1996.

M.Th. van Genuchten. Closed-form equation for predicting the hydraulic conductivity of unsaturated soils. *Soil Science Society of America Journal*, 44:892–898, 1980.

B. van Leer. Towards the ultimate conservation difference scheme, II, monotonicity and conservation combined in a second order scheme. *Journal of Computational Physics*, 14:361–370, 1974.

B van Leer. Towards the ultimate conservative difference scheme, V. A second order sequel to Godunov's method. *Journal of Computational Physics*, 32: 101–136, 1979.

D. Vandevorde and N.M. Josuttis. *C++ templates: the complete guide*. Adisson-Wesley, Boston, MA, 2003.

M. Vauclin, D. Khanji, and G. Vachaud. Experiment and numerical study of a transient, two-dimensional unsaturated-saturated water table recharge problem. *Water Resources Research*, 15(5):1089–1101, 1979.

R.E. Volker, Q. Zhang, and D.A. Lockington. Numerical modelling of contaminant transport in coastal aquifers. *Mathematics and Computers in Simulation*, 59(1-3):35–44, 2002.

C.I. Voss. SUTRA – A finite-element simulation model for saturated-unsaturated, fluid-density-dependent ground-water flow with energy transport or chemically-reactive single-species solute transport. U.S. Geological Survey Report 84-4369, U.S. Geological Survey, Reston, Virginia, 1994.

S.D.C. Walsh, M.O. Saar, P Bailey, and D.J. Lilja. Accelerating geoscience and engineering system simulations on graphics hardware. *Computers & Geosciences*, 35(12):2353–2364, 2009.

C. Wu, S. Agarwal, B. Curless, and S.M. Seitz. Multicore bundle adjustment. grail.cs.washington.edu/projects/mcba/, 2011.

A. Younes, P. Ackerer, and F. Lehmann. A new mass lumping scheme for the mixed hybrid finite element method. *International Journal For Numerical Methods in Engineering*, 67:89–107, 2006.

A. Younes, M. Fahs, and A. Ahmed. Solving density driven flow problems with efficient spatial discretizations and higher-order time integration methods. *Advances in Water Resources*, 32:340–352, 2009.

C. Zhang, X Yuan, and A. Srinivasan. Processor affinity and MPI performance on SMP-CMP clusters. In *the 11th IPDPS Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC)*. IEEE, 2010.

Q. Zhang. *Seawater Intrusion and Contaminant Transport in Coastal Aquifers*. PhD thesis, University of Queensland, Brisbane, Australia, 2000.

Q. Zhang, R.E. Volker, and D.A. Lockington. Experimental investigation of contaminant transport in coastal groundwater. *Advances in Environmental Research*, 6(3):229–237, 2002.

Q. Zhang, R.E. Volker, and D.A. Lockington. Numerical investigation of seawater intrusion at Gooburrum, Bundaberg, Queensland, Australia. *Hydrogeology Journal*, 12(6):674–687, 2004.