MODERN ADVANCES IN SOFTWARE AND SOLUTION ALGORITHMS FOR RESERVOIR SIMULATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ENERGY
RESOURCES ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Rami M. Younis
August 2011

Abstract

As conventional hydrocarbon resources dwindle, and environmentally-driven markets start to form and mature, investments are expected to shift into the development of novel emerging subsurface process technologies. While these processes are characterized by a high commercial potential, they are also typically associated with high technical risk. The time-to-market along comparable development pipelines, such as for Enhanced Oil Recovery (EOR) methods in the Oil and Gas sector, is on the order of tens of years. It is anticipated that in the near future, there will be much value in developing simulation tools that can shorten time-to-market cycles, making investment shifts more attractive. There are two forces however that may debilitate us from delivering simulation as a scientific discovery tool. The first force is the growing nonlinearity of the problem base. The second force is the flip-side of a double edged sword; a rapidly evolving computer architecture scene.

The first part of this work concerns the formulation and linearization of nonlinear simultaneous equations; the archetypal inflexible component of all large scale simulators. The proposed solution is an algorithmic framework and library of data-types called the Automatically Differentiable Expression Templates Library (ADETL). The ADETL provides generic representations of variables and discretized expressions on a simulation grid, and the data-types provide algorithms employed behind the scenes to automatically compute the sparse analytical Jacobian. Using the library, large-scale simulators can be developed rapidly by simply writing the residual equations, and without any hand differentiation, hand crafted performance tuning loops, or any other low-level constructs. A key challenge that is addressed is in enabling this level of abstraction and programming ease while making it easy to develop code that runs

fast. Faster than any of several existing automatic differentiation packages, faster than any purely Object Oriented implementation, and at least in the order of the execution speed of code delivered by a development team with hand-optimized residuals, analytical derivatives, and Jacobian assembly routines. A second challenge is in providing a generic multi-layered software framework that incorporates plug-in low-level constructs tuned to emerging architectures. The inception of the ADETL spurred an effort to develop the new generation AD-GPRS simulator, which we use to demonstrate the powers of the ADETL. We conclude with a thought towards a future where simulators can write themselves.

The second part of this work develops nonlinear methods that can exploit the nature of the underlying physics to deal with the current and upcoming challenges in physical nonlinearity. The Fully Implicit Method offers unconditional stability of the discrete approximations. This stability comes at the expense of transferring the inherent physical stiffness onto the coupled nonlinear residual equations that are solved at each timestep. Current reservoir simulators apply safe-guarded variants of Newton's method that can neither guarantee convergence, nor provide estimates of the relation between convergence rate and timestep size. In practice, timestep chops become necessary, and they are guided heuristically. With growing complexity, convergence difficulties can lead to substantial losses in computational effort and prohibitively small timesteps. We establish an alternate class of nonlinear iteration that converges and that associates a timestep to each iteration. Moreover, the linear solution process within each iteration is performed locally. Several challenging examples are presented, and the results demonstrate the robustness and computational efficiency of the proposed class of methods. We conclude with thoughts to unify timestepping and iterative nonlinear methods.

Acknowledgements

It has been an interesting journey. I am fortunate to have had the company of many who have challenged and supported me. I would like to thank Khalid Aziz for taking me in and for accommodating my sometimes rogue ways. I would also like to thank Hamdi Tchelepi for reminding me to pause and appreciate intermediate milestones before "turning round more new corners". I would also like to thank Juan Alonso for reading this dissertation and providing valuable feedback. I have had the opportunity to learn from exceptional classroom teachers; I am especially indebted to Julie Levandosky, Doron Levy, Gene Golub, and Lou Durlofsky. Your classes were not easy and they were always fun. I would also like to thank the staff at the Center for Teaching and Learning for recruiting me as a Teaching Consultant during my studies. The Department of Energy Resources Engineering (known as Petroleum Engineering at the time I joined) is a very special one at the university primarily because of its administrative leadership and citizens. I would like to thank Roland Horne who served as Department Chair throughout most of the duration of my studies. I would also like to thank my colleagues for making every day (and night) at the office feel like a walk in the park; Saman Aryana, Marc Hesse, Shalini Krishnamurthy, Morten R. Kristensen, Felix Kwok, and Jonas Nilson it would not have been fun without you. I am indebted to Sylvianne Dufour and to Dr. B. for helping learn some of life's hardest lessons. I would also like to thank my girlfriend Stephanie E. Holm for putting up with my nonlinear humor. My thanks go to my parents, Hiyam and Mustafa Younis, for guiding me and for supporting me unconditionally at the times I needed it most. This work was prepared with financial support from the Stanford University Petroleum Research Institute-B (SUPRI-B) Industrial Affiliates Program.

Contents

List of Tables

List of Figures

Part I

The Automatically Differentiable Expression Templates Library

Chapter 1

Introduction

As conventional hydrocarbon resources dwindle, and environmentally-driven markets start to form and mature, investments are expected to shift into the development of emerging subsurface process technologies. While such emerging processes are characterized by a high commercial potential they are also typically associated with high technical risk. The development pipeline starts from basic and applied research before moving onto proof-of-concept studies and field testing, finally leading to commercialization. The time-to-market along comparable development pipelines, such as for Enhanced Oil Recovery (EOR) methods in the oil and gas industry, is on the order of tens of years. Subsequently, in addition to the value of simulation to commercial-scale stages, there is much value in developing simulation tools which can shorten the time-to-market, making investment shifts more attractive. Increasing computational power and adoption are likely to further the importance of simulation. Additionally, the steady growth in remote sensing technology and data integration methods makes simulation objectively relevant to decision making. There are two challenges towards using simulation as a scientific discovery tool in early development stages as well as a tool for decision making; the growing complexity of the problem base and a rapidly changing computer architecture scene.

Current examples of emerging processes include peculiar flow effects in Unconventional Gas Production, Underground Coal Gasification, Electro-thermal Stimulation in Heavy Oil, and Enhanced Geothermal Systems. The underlying commonality that

characterizes such emerging processes is their physical complexity. One aspect of the complexity is in the superposition of a wide range of physical phenomena each occurring at a different scale, and at different times throughout the life of the process. Another aspect of complexity is in the nonlinearity of the underlying constitutive relations which tend to evolve steadily and in parallel with experimentation. The technical ability to serve this application space therefore clearly relies on the ability to extend and customize commercial grade software to incorporate the new and continuously evolving models within accepted computational performance standards. This ability has become a technical software issue, raising questions regarding abstractions and software architecture.

A second motivating force for this work is a rapidly changing hardware scene. With the popularization and ubiquitous availability of multi-core multi-processor platforms, and the promise of heterogeneous or hybrid systems (e.g. Multi-core chips and General Purpose Graphics Processing Units GPGPU) the hardware problem has become a software problem, and according to Herb Sutter we are at a fundamental turning point in software development [65, 64]. Not only do we need to rethink software abstraction, but also, detail programmers need to become more adept at tuning and optimizing code with a sensitivity to the underlying target hardware architecture. The challenge of designing and developing simulation software for future platforms is significant, especially in light of the current and growing complexity. Simulation codes need to be designed so that they are very quickly adapted to new or multiple platforms without a need to re-gut or overhaul major pieces of software with each release. Without further advances, solutions to this challenge are very much orthogonal to the interest in comprehensive or general purpose tools.

Considering these two broad goals; rapid extendability and minimally invasive platform awareness, this work analyzes a major component of all serious simulation software. The formulation and linearization of nonlinear simultaneous equations is the archetypical inflexible component of all large scale simulators. The author has designed, developed, and tested a software framework that allows rapid customizations to the core of any simulation model using domain specific concepts only. Notions of

platform specific programming and tuning are incorporated into key underlying kernels which are effectively abstracted from the simulator developer. They exist behind the scenes.

1.1 Modern Simulation Software

With the hopes of meeting the present and upcoming challenges, today's simulation software developer may often find herself concerned with abstraction decisions and choices. These concerns are valid and can easily become overwhelming considering how challenging it is to deliver software that is comprehensive, rapidly extendable, and that possesses some form of minimally invasive platform awareness. How can we build simulation software that is both general or comprehensive, and yet, very specific without killing performance? The only duly diligent answers to this question are those that visit both the fundamental architectural choices and the programming abstractions with which simulators are conceived.

1.1.1 Simulation Software Architectures

A software system architecture is a logical organization of its components. Similar to organizational charts that are used to describe members of an organization and the interactions between them, software architectures are often represented and communicated using drawings. Garlan and Shaw [31, 30] propose a common framework with which to view software architectural styles. Their framework treats the architecture of a system as a collection of computational *components* along with a prescription of their interactions or the *connectors* between them.

Once there was a Monolithic Style

The earliest reported simulators (see for example [66, 55] were architected as Monolithic Systems that carry out the complete task of running a simulation from end to end. Figure 1.1 illustrates a monolithic simulator as simply nothing more than a box. In the context of these monolithic architectures, modularity into components took the

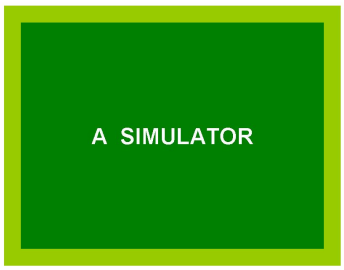


Figure 1.1: Depiction of an architecturally monolithic simulation software system.

form of programmer remarks and comments that may mark sequences and branches. Connectors between such components were simple; a pipeline flow of data through a sequence of operations. This architectural style earns the popular label, "spaghetti string code". Monolithic forms of modularity have proven to be practically of little use in terms of their maintainability and many such legacy codes have been shelved in favor of another breed of Monolithic system; structured modular simulators. The growing faith in compiled languages spurred the support of a structured programming style. This led simulator developers to modularize applications by introducing procedures and functions which would be re-used and would in theory improve maintainability. Nevertheless, the architecture remains monolithic in the sense that data still moves through a pipeline sequence of operations.

Monolithic systems must essentially be completely re-written to achieve variations on a specific task. In fact, the monolithic architecture leads to many codes and many executables. Figure 1.2 illustrates a matrix of distinct simulation solutions, each of which solves a particular set of problems in a particular manner. Clearly as the space of possible variations expands, the number of solutions required grows combinatorially.

Despite this shortcoming, even today, many would argue that such legacy codes have an undeniable edge in terms of their computational performance relative to more maintainable architectures. The reality is that the firm faith in monolithic architectures is fueled by more than just nostalgia. The undeniable fact is that such architectures almost force the designer and programmer to avoid expensive operations

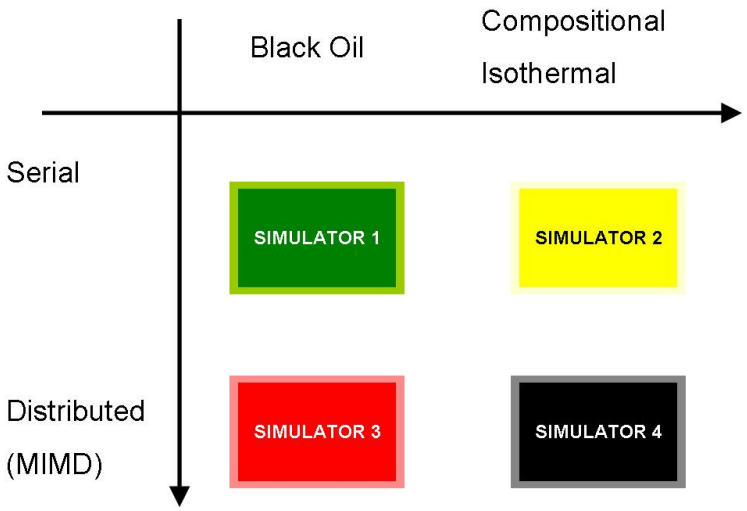
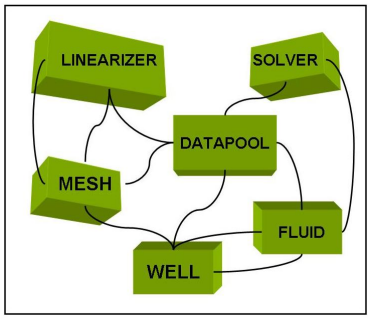


Figure 1.2: The space of monolithic simulators spanning two axes; instruction and memory model, and physical model. The result is a many-code, many-executable space of applications.

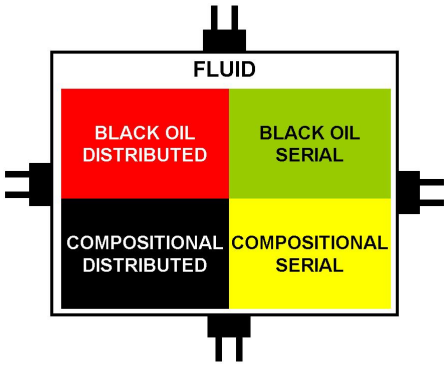
such as dynamic memory, copying memory back and forth, context switching, and to a large extent, indirection.

Data Abstraction and Object Orientation

As a first response to the frustration with the maintainance of monolithic codes, a different kind of architectural style quickly became the de facto standard for large scale simulator designs [13, 53, 22]. In Data Abstracted and Object Orientated architectures, data and associated operations are encapsulated in abstract data types. A specific instance of an abstract data type is called an object. While objects form the components of this style, the connectors are essentially a set of procedure or function invocations which form the interface of an object. Objects are responsible for maintaining some invariant property and the details of how it is implemented are hidden form other objects. Figure 1.3 illustrates an Object Oriented simulation architecture. The objects and their connections are depicted by Figure 1.3(a) while Figure 1.3(b) illustrates a zoomed view onto one module which is responsible for encapsulating



(a) Depiction of a simulation system with a Data Abstracted and Object Oriented Architecture. Each component is a module that hides its own functionality and data. Modules are connected by fixed interfaces.

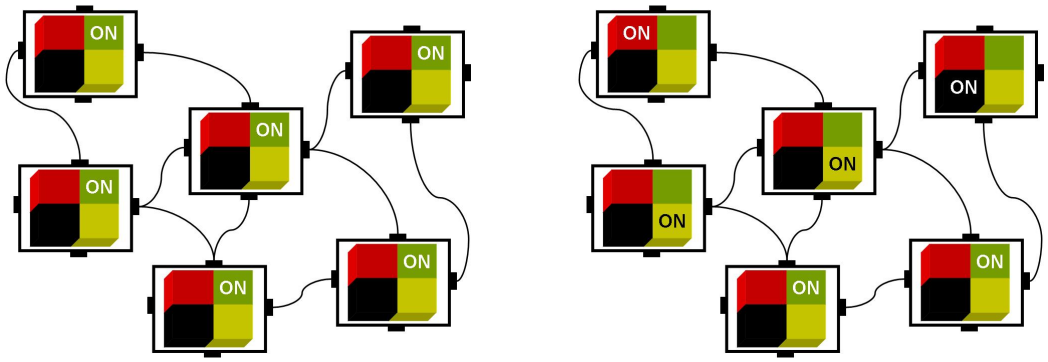


(b) A module of a fluid concept will typically have several implementations, each of which fulfilling the needs of some sort of specific instance. The interface to the module remains fixed regardless of the internal behavior. This is a form of polymorphism.

Figure 1.3: An illustration of the Data Abstracted and Object Oriented architectural style.

data and algorithms concerning fluids. The fluid object has many possible internal implementations, and yet all implementations need to fit within a prescribed interface which is depicted by the sockets in Figure 1.3(b).

Perhaps the most outstanding advantage of Object Oriented styles is that they admit a one-code, one-executable model to delivering a range of possible simulator variations. The General Purpose Reservoir Simulation idea is a concrete example of this. With a single code-base leading to a single executable the GPRS [13] allows users to perform two point and multi point flux approximation simulations for compositional or black oil models. Figure 1.4 shows how this style facilitates this model of extendability. The difference between the Black Oil serial simulator depicted by Figure 1.4(a) and the compositional distributed memory simulator depicted by Figure 1.4(b) is in the choice of internal implementation of specific objects. Rather than re-write the whole simulator, an extension can be made possible by adding a specific internal implementation of one or more modules. The object model and connectors remain unchanged, while internally, objects use one of many possible variations. In the language of Object Oriented design, this is called polymorphism.



(a) A black oil serial simulator.

(b) A compositional distributed simulator.

Figure 1.4: The Data Abstracted and Object Oriented architectural style admits variations by selecting individual implementations of its components. The connectors must remain the same to allow plug-and-play flexibility. In this style variations are obtained by a one-code, one-executable system.

The advantageous aspect of a component object model is itself the source of a number of its shortcomings. The first shortcoming is concerned with medium to long term maintainability. In the short term, the intent of modular architectures is to enable extensions by modifications, substitutions, or additions to modules of interest, isolating cross-module dependencies. Over the course of several years however, as the problem base to be covered expands dramatically and in unanticipated ways, the object component philosophy lewads to the need for a complete re-write of the simulator. In order to maintain the isolation of items of functionality and their associated data from each other and to eliminate the need for intimate knowledge of every aspect of all modules, the developer is bound to run into new concepts that simply do not fit the imposed rigid mold. The second shortcoming is one of computational efficiency. Polymorphism inherently encourages indirection and substantial runtime context switching. That is, the means available to achieve polymorphic architectures often force the developer to make numerous decisions at runtime. These branches can have a significant impact on performance.

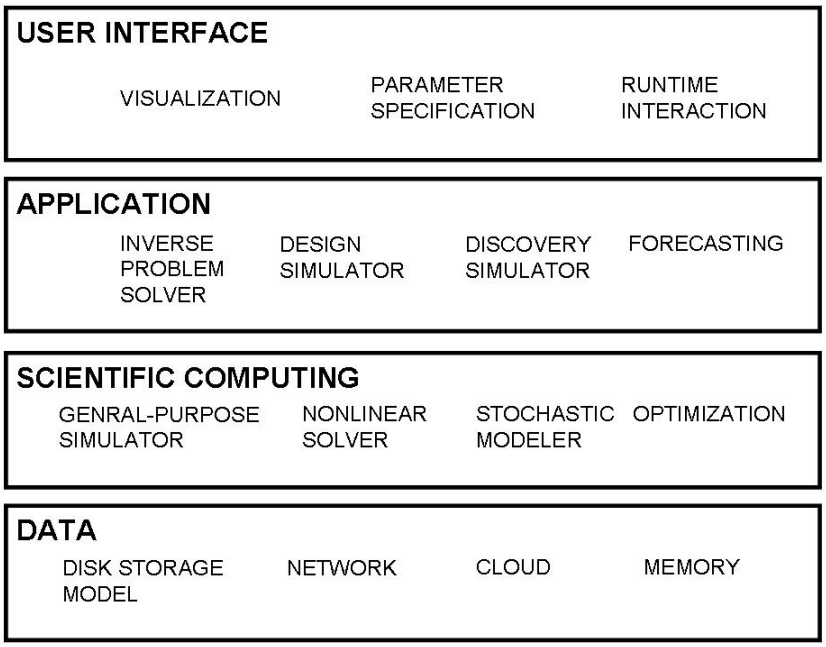


Figure 1.5: A schema of a multi tier architecture of a simulation application. Each layer is comprised of modules from the layer below it, and adds new, higher order concepts. Each layer can be used on its own to generate an application, or it can be used to build elements in a layer up.

Multi layered Architectures

A common simulation software architecture today is the Multi layered style. Simulators architected in this style are made up from modules that are pulled out of a multi layered software framework (for example [23, 5]). Each layer consists of a set of modules that implement a certain level of functionality and which can be used in a natural syntax suitable to their level. Higher-layer modules implement increasingly complex functionality by using lower-layer modules. Apparent advantages of taller hierarchies are greater flexibility, extendability, and reliability. An important practical advantage is that layering allows experts from different areas to contribute to the most suitable layer for their expertise. For example, at the lowest level, computer programming experts can tune the performance of an algorithm to the specific computer architecture, while scientists and engineers can use a mathematical syntax created by these modules to contribute their domain expertise without the burden of low-level details. Figure 1.5 depicts such an architecture.

Zooming into the scientific computing component, Figure 1.6 presents a possible multi layered hierarchy of successively abstracted scientific computing elements which can be used to put together a numerical simulator. Then lowest layer is concerned with memory and thread management which is a basic function of all codes. The usage syntax of modules within this layer deals with basic concepts such as memory addresses and locality of reference. The highest layer provides modules that implement discretizations of state vectors and partial differential operators. The syntax used in this layer is more natural to a simulation scientist whose level of granularity is on the level of putting a time-stepping module together with a solver to build a simulator. It is important to emphasize that as long as each added layer moving up the hierarchy is constructed with clear limits on the associated abstraction penalty, any combination would produce an efficient code.

A potential implication of taller multi-layer frameworks enabled by such abstractions is the emergence of fully-configurable and customized models on demand. As in Figure 1.6, domain experts can generate tailored purpose tools and to match them with interfaces suited to the purpose at hand. For example, using the same core software, on-site production monitoring could involve near-well specific numerics with an

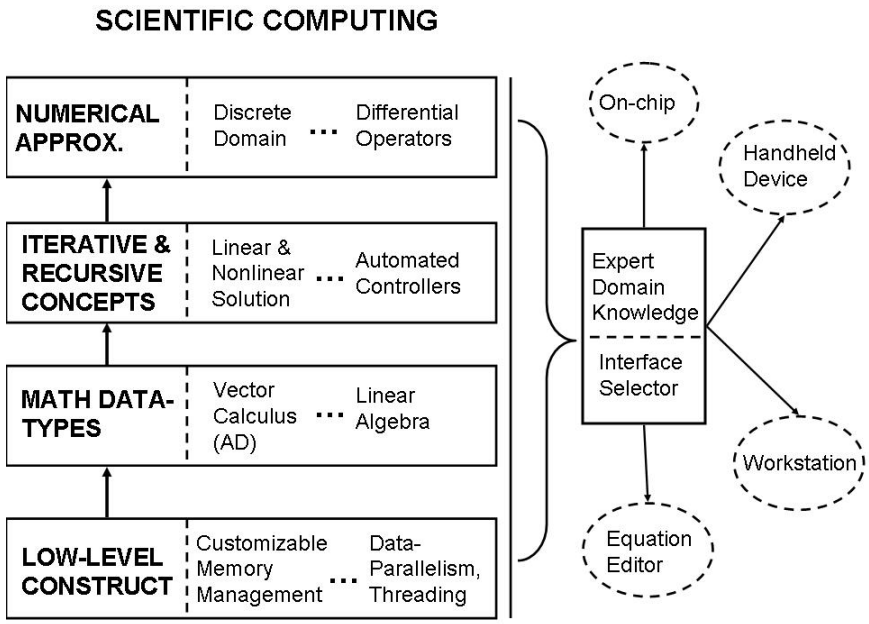


Figure 1.6: A Multilayer schema of a scientific computing module. An expert system applies domain knowledge, and along with an interface selector, puts together tailored applications on demand.

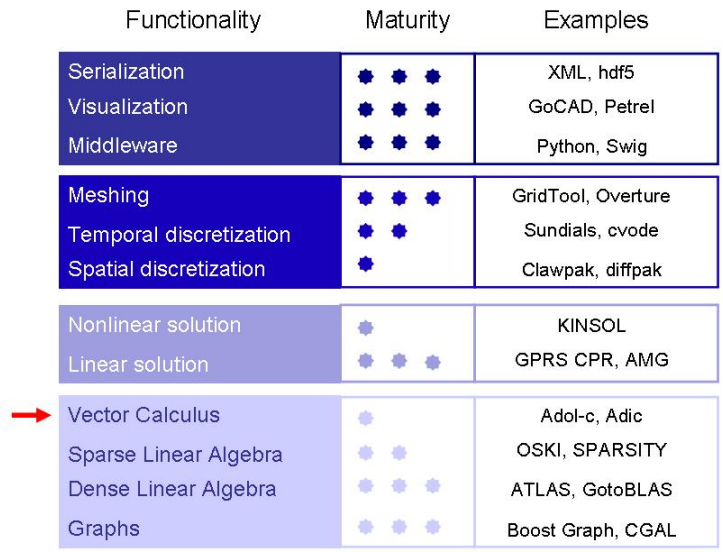


Figure 1.7: A survey of software in scientific computing.

interface through a hand-held device, whereas research into new Enhanced Oil Recovery (EOR) processes could involve rapid prototyping of new physics and formulations. This architecture leads to a one-code, multiple executable space of variations which ideally is a best of both worlds solution. The performance of runtime polymorphism is minimized while combinatorial flexibility is realistic.

Figure 1.7 surveys a set of modern scientific computing software, arranged in a multi layered view. Subjectively at least, mid-layer Scientific Computing software such as is in the areas of generic Nonlinear Solvers and Vector Calculus support are not considered mature. In the example of Vector Calculus there is little to no software that efficiently provides support to major simulation software. Linear Algebra packages seem to be the exception however. Numerous robust and popular Linear Algebra packages are available and they are typically built directly form low-level layers such as the Basic Linear Algebra Subroutines BLAS. Within this context, this work contributes to the mid-level layers with the goal of bringing simulation systems closer to the ideal of providing a single efficient and maintainable source base, and

multiple tailored executables on demand.

1.1.2 Programming Abstraction Paradigms

Programming abstraction paradigms are different styles of abstractions used to model components and connectors. Programming languages typically support more than one paradigm. Despite this, it may seem more natural to use one language over another when committed to one specific abstraction style. Often an application is written using more than one paradigm or language. With the hopes of meeting the goal of furthering the multi layered architecture, it is useful to consider three paradigms; the Object Oriented, Generic, and Metaprogramming paradigms.

Object Orientation in reservoir simulation

The *Object Oriented Programming* (OOP) paradigm provides the developer with language facilities and constructs with which to easily deliver an Object Oriented software architecture[63, 29]. For example, the notion of polymorphism from the architectural perspective is implemented using specific OOP constructs. Simulators developed entirely with this paradigm introduced more flexibility as far as an Object Oriented architecture admits. However, such simulators typically consist of at most two abstraction layers[72, 14]. The upper-layer is typically heavy, and consists of Object Oriented modules, internally implementing computational kernels in a structured paradigm. The lower-level is typically a collection of libraries which implement the most basic data-structures and associated algorithms.

In reservoir simulation, and in scientific computing in general, this top-heavy design choice was often made out of necessity. The issue was that the OOP paradigm offered higher degrees of abstraction at the cost of notable performance hits. The primary source of such degradation in computational performance is polymorphism. To understand this, we consider the example of implementing a well control object within a large OO simulator.

Figure 1.8 shows the system diagram for a well control object. The object can be implemented in one of four concrete ways; Bottom Hole Pressure (BHP), oil,

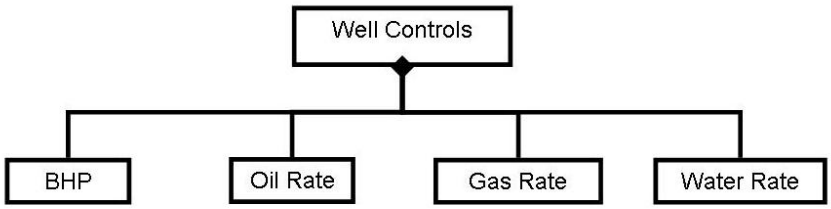


Figure 1.8: An inheritance diagram of a well control object. There are four concrete ways to implement a well control object. Any of the four can be referred to using a link to a well control object. The specific implementation choice is deduced at runtime based on the context.

gas, or water rate control. The OOP implements polymorphic behavior through a mechanism called inheritance. With this construct, BHP control objects are said to inherit common control features such as interface from the abstract well control type. The BHP control concrete type is said to extend the well control type with specifics. The desirable feature of this is that extending the well control variations of a simulator is simple in concept. A developer can add another concrete type that inherits the general interface of the well control type. There are however numerous albeit subtle potential pitfalls to the liberal use of this construct.

Firstly, writing a new concrete type may be a substantial endeavor in and of itself. The neatness of the existing inheritance relations and their common interface may need to be re-examined, and the object implementation itself is typically built using lower-level objects which themselves are more than likely to be polymorphic. This leads to a deepening of the inheritance tree. This is where the dreaded OOP performance hits can come into play. At runtime, an owner of a well control object must address it by an indirect reference. This is because the precise nature of the internal details of the object are not known to the owner. More precisely, when well controls are to be treated as just that, regardless of their internal implementation, the strongly typed nature of most compiled languages requires that one level of indirection be used. At runtime, depending on the context, the appropriate implementation is selected. This means that at every inheritance juncture in a possibly very tall hierarchy, a suitably large jump table is used. This sort of branching is not only costly

in and of itself. A more severe cost could be the disruption of inlining. Consider the consequences of a simple query to see if a certain well-control is satisfied. In this example design, this would be performed for each well, with every residual call in each nonlinear iteration of each timestep of a simulation. Each of these calls triggers a conditional branch which leads to a function call involving perhaps a couple of lines of code. This could be disastrous. We need a way to parameterize objects by the type of their inner workings so that we can avoid these runtime performance hits.

Generic Programming

The *Generic paradigm* introduces a way of programming algorithms generically, leaving it up to compilers to instantiate or adapt to specifics based on the context in which the algorithm is being used. The context dependencies are inferred at compile time, and so the generic paradigm introduces the possibility for a new kind of polymorphism. Compile-time polymorphism promises to rid designers and implementers from the runtime costs of OOP polymorphism while retaining some of its flexibility advantages. The idea is to offer Object-Oriented flexibility at little, or no, overhead compared to structured codes[39, 57, 68].

A principle philosophy behind the generic paradigm is a separation of datastructure and algorithm. The C++ Standard Template Library (STL) is a prime example of this. Various sorting routines are available and each one works with any datastructure that satisfies certain broad criteria. The specializations and specific accommodation choices within the algorithm are made at compile time. In that sense, the STL shows precisely how the generic paradigm introduces a one-code, many-executables scenario. The idea is to use the generic paradigm to introduce programming in the middle. Modules in such intermediate layers build onto lower-layer primitive or intrinsic libraries, and they implement enriched data-structures with enriched algorithmic capabilities. Higher layers use the OOP to deliver runtime flexibility without excessive abstraction performance penalties.

A notable generic programming technique that has impacted the scientific computing community is known as Expression Templates (ET). The ET technique was first described by Veldhuizen [70], and independently by Vandevoorde [67], and in

the scientific computing context by Haney [40] and Kirby [45]. It makes heavy use of the templates feature of the C++ language to eliminate the overhead of operator overloading on abstract data-types, and without obscuring notation. To date, the major scientific computing applications which capitalized on this technique are predominantly in the Linear Algebra and other mid-layer generic data-type settings. Production grade codes include the Parallel Object Oriented Methods and Applications (POOMA)[42] collection, the Matrix Template Library (MTL)[62], and the Blitz++[71, 69] vector arithmetic library. These efforts are often credited for demonstrating the emergence of the C++ programming language as a strong contender to Fortran in terms of performance in scientific codes [39, 57, 68]. At the time of development of the ET technique however, compiler support and debugging protocols for the generic metaprogramming paradigm were limited. Only a handful of compilers could pragmatically handle such code. In contrast, today, all of the major popular optimizing compilers list pragmatic optimizations to compile heavily generic code. Successful uses of generic techniques such as ET in scientific computing application areas, and the continually improving compiler support motivate us to explore applications to Reservoir Simulation.

Metaprogramming

More recently, several scientific computing initiatives have embraced the *Metaprogramming paradigm*. This paradigm makes further use of compilers to perform actual computations at compile-time [1]. This spurred a number of initiatives which combined the use of the Generic and Metaprogramming paradigms to create intermediate abstraction layers. We note that until very recently, compilers did not fully support the Metaprogramming paradigm. Wider compiler conformance to the language standard is encouraging mainstream applications, and some of the most recent reports of Next Generation Simulators take advantage of this[23].

Because of its very nature, metaprogramming lives at compile time only. Not only are the deductions and operations performed at compile time, but also the resultants exist at that point as well. A popular trend uses forms of metaprogramming within Just In Time (JIT) compiler settings to introduce some forms of runtime flexibility.

For example, an interpreted code may direct execution. As certain runtime quantities become known, a JIT compiler uses this information to compile an accordingly optimized module, which the interpreted runtime environment uses.

1.1.3 The Software Goals

The software objective of this work is to further the way we build reservoir simulation software for performance and maintainability in light of increasing complexity. We aim to do this by example. We design, develop, and demonstrate a mid-layer generic library that can be used with a OOP style or otherwise to build highly configurable simulation software. We aim to provide clients (other developers) working on top-layer development with efficient and flexible data-structures that can construct Jacobian matrices behind the scenes. The aim in terms of interface is a natural syntax for mathematical expressions. This encapsulation allows clients to avoid large, difficult to debug, and inflexible pieces of code related to the Jacobian matrix implementation without sacrificing efficiency.

1.2 Formulation and linearization

Reservoir simulation equations are nonlinear. To obtain discrete approximations with a certain level of implicitness, simultaneous systems of nonlinear equations need to be solved. Nonlinear solution methods employ linearization and solution of the resulting linear systems. For a nonlinear vector function $r : \mathbb{R}^n \rightarrow \mathbb{R}^m$, with n variables and m equations, linearization involves the computation of the typically sparse Jacobian matrix, $J_{i,j} = \frac{\partial r_i}{\partial x_j}$, $i = 1, \ldots, m$ and $j = 1, \ldots, n$. The least extensible most tedious and error prone part of any reservoir simulation software is that implementing this formulation and linearization component. The evaluation of the residual, r, and the Jacobian matrix, J, for a given set of unknowns is also a component of the code which relies heavily on expert domain knowledge of the physical models and their discrete approximation. Programmers involved in the development of formulation and linearization codes are typically less interested in, and capable of, hand-tuning

for performance. The orthogonality between hand optimization and code clarity from the domain expert's view exacerbates the challenge of producing simulators for emerging computer hardware. For these reasons, formulation and linearization are a prime opportunity for innovation in software architecture and abstractions.

1.2.1 Formulation concepts

Depending on the physics of a simulation, there is a minimum number of independent variables and equations that fix the thermodynamic state of the system. For example, in isothermal compositional systems, the thermodynamic state is fixed by as many independent variables as the number of components in the system. All remaining quantities can be obtained using the independent variables and constitutive relations or algebraic constraints. Often, the algebraic constraints can only be written as implicit nonlinear functions, and subsequently the inverse needs to be computed. A common tactic is to couple these auxiliary constraints to the minimal set of discrete equations as a system of Partial Differential Algebraic Equations (PDAE). The specific choice of independent variables, secondary variables and algebraic constraints, and the alignment of unknowns with equations is generally referred to as a *formulation*. While different formulations may have different numerical properties, all formulations lead to the same unique answer. There is considerable interest in studying the properties of various formulations. Moreover, there is a common belief that certain problems warrant the use of one formulation over another. Regardless of the reality behind that belief, it is important for modern simulation software to accommodate such interests.

The General Purpose Reservoir Simulator [13] proposes a certain general formulation approach. The GPRS approach to formulation is to use the natural variables as a universal formulation. Any other formulation is attained indirectly within the linearization process using the Jacobian matrix of the transform from one set of variables to the other. Variable ordering and alignment are treated using Jacobian matrix permutation operations. The GPRS approach starts with the full set of equations and variables, and undergoes a two stage reduction process. First, the system is reduced

to the primary set, and then the resultant is reduced to the minimal implicit variables. While the GPRS formulation approach introduced a far leap from the static formulation setup of its predecessors, these transformation operations are by their very nature hard coded into the guts of the formulation and linearization routines. Any subsequent developer that extends a capability within the general purpose umbrella needs to directly account for the extensions into every formulation transformation system. This easily turns certain extensions into endeavors that are error prone and almost impossible to debug by a unit testing methodology.

Phase transitions are a basic fundamental of flow through porous media. Such transitions are often modeled as instantaneous. Subsequently, the number of degrees of freedom within an element of a discrete domain is subject to change instantaneously. Along with the number of degrees of freedom, it is entirely possible that the set of independent unknowns and the equations that they are aligned with change as well. This introduces obvious challenges to producing maintainable code. The difficulties are compounded considering that the precise manner in which the switching between alternate variable sets is dependent on the formulation used.

One goal of this work is to introduce a general abstraction that provides a generic and universal model for every possible formulation. Through this abstraction, the formulation my be specified in a parameter specification type of syntax, and without any programming, the formulation is realized.

1.2.2 Linearization operations

Once a formulation and its associated concepts are chosen, there are three distinct methods that can be used to implement Jacobian evaluation routines;

- Manual differentiation involves explicit derivation and programming of all derivatives and Jacobian assembly routines. Most commercially and publicly available simulators adopt this approach [33, 17, 13]. This is a tedious and notoriously error prone process, but results in efficiently computed, exact Jacobian matrices up to rounding error. The chief drawback is the lack of flexibility. Changing

numerical formulations or physical models can require partial rewrites of substantial components spanning many modules from say, property packages, to equations, to solver setups.

- Numerical differentiation uses truncated expansions, possibly with divided differences, along with coloring and separation sparsity-graph algorithms to approximate the Jacobian [32]. While this approach is flexible, it has three drawbacks. First, conditional branches cannot be treated as desired (e.g. upwinding or variable switching). Second, it is not always possible to bound the truncation error *a priori* for arbitrary functions. Finally, the asymptotic complexity can limit the efficiency.

- Automatic Differentiation (AD) is a technique for augmenting computer programs with derivative computations. AD analyzes the expression parse-tree and performs a handful of mechanical rules to evaluate derivatives: summation and product rules, transcendental elementary function derivatives, and the chain rule. The approach offers flexibility and generality, and accuracy up to machine precision. However while the asymptotic complexity is comparable to that for analytical differentiation, the ability to develop optimally efficient implementations is problem specific and an ongoing question.

To meet our objectives for a mid-layer library, AD seems to be the most viable approach. AD combines flexibility and accuracy, as well as the potential for optimal efficiency provided implementations are tailored to the application at hand. In this work, we aim to design and tailor an AD approach to compute general reservoir simulation Jacobian matrices automatically in a fast, bug-free, and notationally convenient manner.

Today, a sizable research community continues to develop various aspects of AD including improved methods for higher-order derivatives, parallel schemes, and sparsity-aware methods. Several comprehensive introductions [56, 27, 34], and volumes of recent research activity [11, 19, 36] are available. Information about the growing software packages, literature, and research activities in the area is available on the World Wide Web at `www.autodiff.org`.

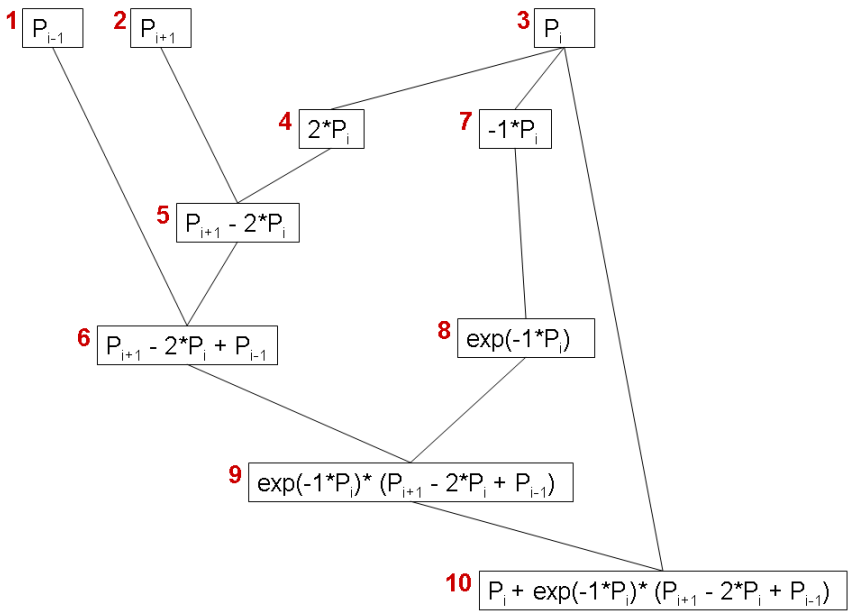


Figure 1.9: The ordered parse graph representation of the sequence of intermediate steps in evaluating Equation 1.2.1. The ordering is dependent on operator precedence rules.

Mathematics of AD

Given a general function of several variables, $f : \mathbb{R}^n \rightarrow \mathbb{R}$, in theory, there are two alternate algorithmic approaches to AD (and differentiation in general); the forward and reverse (adjoint) modes. For specified independent variables $(x_1, \ldots, x_n)$, both modes evaluate the function $f(x_1, \ldots, x_n)$, as well as the gradient vector $\nabla f \equiv \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)$. The modes differ in the manner in which these quantities are computed. In the forward mode, the chain rule is used to propagate derivatives starting from independent variables and progressing towards the dependent variable. On the other hand, the reverse mode propagates derivatives of the final dependent variable through a sequence of adjoint additions of intermediate dependent variables. To propagate adjoints, one must be able to parse an expression, evaluate it, and then reverse the parse-flow, while remembering or recomputing any intermediate values that nonlinearly impact the final result.

To illustrate the forward and reverse modes, we apply them to differentiate the

scalar multi-variate expression in Equation 1.2.1. Assume that there are three independent variables, ordered as (P_{i-1}, P_{i+1}, P_i). Following the basic arithmetic precedence rules, and proceeding from the right to the left, the nonlinear function in Equation 1.2.1 is said to generate a topologically sorted parse-graph as shown in Figure 1.9. The parse-graph represents the precise sequence of elemental operations (binary, unary, and transcendental) needed to evaluate the expression.

$$f^{(i)} = P_i + \exp(-P_i)\,(P_{i+1} - 2P_i + P_{i-1}) \tag{1.2.1}$$

Computer program compilers parse expressions and apply right to left operator precedence rules to generate parse-graphs. At run-time, the expressions are evaluated by a traversal of the graphs, accumulating the final result. Intermediate stages consist of temporary values for each node computed behind the scenes. We denote such values as $t_j, j = 1, \ldots, M_{nodes}$, and present their expressions for this example in the listing below. Here, $M_{node} = 10$, and in the way that Equation 1.2.1 is written, seven kernel operations are required.

$$\begin{array}{lll} t_1 = P_{i-1} & t_4 = 2t_3 & t_7 = -t_3 \\ t_2 = P_{i+1} & t_5 = t_2 - t_4 & t_8 = \exp(t_7) \\ t_3 = P_i & t_6 = t_1 + t_5 & t_9 = t_6 * t_8 \\ & & t_{10} = t_3 + t_9 \end{array}$$

The forward mode.

Given values for the three independent variables, the forward mode proceeds by sequentially computing intermediate-stage values as well as their gradients. This is a top-down traversal of the parse-graph in topological order. The sequence listed below, shows the gradient evaluations which occur simultaneously with the function evaluations. There are three initialization stages, followed by seven stages, each of which involves the differentiation of only a single unary, binary, or transcendental operation.

The first three initialization stages state that the gradient of an independent variable with respect to all three variables is simply the appropriate basis vector. Overall, this is a sequential process that can propagate the entire gradient along the traversal. Moreover, it is mostly a serial process unless the parse-graph has separable branches; for example in this case, steps 4,5, and 6 can proceed in parallel with steps 7 and 8. This kind of separability/parallelism on its own is of unlikely interest in reservoir simulation applications, since it is difficult to exploit efficiently. Rather, suppose we are considering the set of three equations $\left(f^{(i-1)}, f^{(i+1)}, f^{(i)}\right)$, aligned with the three unknowns, and augmented with appropriate boundary conditions. Then the evaluation of the three-by-three Jacobian matrix amounts to stacking the three gradients as rows. Each row computation is separable with little data overlap. This property can more readily be exploited for computational parallelism on shared memory settings.

$$
\begin{aligned}
\nabla t_1 &= [1,0,0] \\
\nabla t_2 &= [0,1,0] \\
\nabla t_3 &= [0,0,1] \\
\nabla t_4 = 2\nabla t_3 &= [0,0,2] \\
\nabla t_5 = \nabla t_2 - \nabla t_4 &= [0,1,-2] \\
\nabla t_6 = \nabla t_1 + \nabla t_5 &= [1,1,-2] \\
\nabla t_7 = -\nabla t_3 &= [0,0,-1] \\
\nabla t_8 = t_8 \nabla t_7 &= [0,0,-t_8] \\
\nabla t_9 = t_8 \nabla t_6 + t_6 \nabla t_8 &= t_8\left[1,1,-\left(2+t_6\right)\right] \\
\nabla t_{10} = \nabla t_3 + \nabla t_9 &= \left[t_8, t_8, 1 - t_8\left(2+t_6\right)\right]
\end{aligned}
$$

From the example it is easy to see that if there are N_v logically non-zero entries in the final gradient, then the worst-case complexity of the forward mode is $O\left(N_v\right)$ times the number of operations to compute a single function evaluation. In this example 14 additional kernel operations are required beyond the seven operations for

the function evaluation. This implies that if one were to apply the forward mode to a vector function, $f_i, i = 1, \ldots, N_b$, then the sparse Jacobian comes automatically at a price of $O(N_b N_v)$ scalar function evaluations. This, at least in the asymptotic sense, is comparable with computing the Jacobian by hand, assuming no exploitation of repetitive structures is made. Despite this, the complexity result is often considered too large since it scales with the number of independent variables, and in practice, expert developers will invariably take advantage of factoring short-cuts for specific problems. With a combined application of local node reduction graph algorithms and global separations on columns, it may be possible to further improve the complexity. Finally, we remark that conditional branches are treated nicely by the forward mode since only the instantiated branch is evaluated at runtime.

The reverse mode.

The reverse mode proceeds in bottom-up topological order along the parse-graph. In contrast to the forward mode, elements of the gradient are accumulated independently. First the final dependent variable is differentiated with respect to itself for initialization. Then recursively, the gradient element adjoints are accumulated as in Equation 1.2.2.

$$\frac{\partial t_n}{\partial t_n} = 1, \frac{\partial t_n}{\partial t_j} = \sum_{k=n-1}^{j} \frac{\partial t_n}{\partial t_k} \frac{\partial t_k}{\partial t_j}, \; j = n-1, \ldots, 1 \tag{1.2.2}$$

After one pass, all individual derivatives are propagated, and then they are assembled into a gradient vector. The sequence listing below correctly computes the elements of the gradient of Equation 1.2.1 by the reverse mode. We have introduced the shorthand notation $t_{j,k} \equiv \frac{\partial t_j}{\partial t_k}$.

$$
\begin{aligned}
t_{10,10} &= 1 \\
t_{10,9} &= 1 \\
t_{10,8} = t_{10,9}t_{9,8} &= t_6 \\
t_{10,7} = t_{10,8}t_{8,7} &= t_6 t_8 \\
t_{10,6} = t_{10,9}t_{9,6} &= t_8 \\
t_{10,5} = t_{10,6}t_{6,5} &= t_8 \\
t_{10,4} = t_{10,5}t_{5,4} &= -t_8 \\
t_{10,3} = 1 + t_{10,7}t_{7,3} + t_{10,4}t_{4,3} &= 1 - t_8\,(2 + t_6) \\
t_{10,2} = t_{10,5}t_{5,2} &= t_8 \\
t_{10,1} = t_{10,6}t_{6,1} &= t_8
\end{aligned}
$$

In this case, the reverse mode requires only 11 additional kernel operations beyond the seven required for the function evaluation. In fact, the reverse mode requires $O\,(1)$ times the number of operations for function evaluation. This constant scaling at first glance is appealing since it implies that a Jacobian can be computed in $O\,(N_b)$ scalar function evaluations. The favorable complexity scaling however can be easily overshadowed by necessary and expensive implementation constructs. It is precisely the point that the most efficient AD realizations must combine algorithmic and implementation properties along with tailored application to the problems at hand.

For arbitrary sparse Jacobian problems, there are no hard rules as to the most favorable approach since the underlying sparsity structure and how it is exploited can influence the scalings. We devise our library with the capability to perform in either mode, with our current emphasis on the forward mode. The motivation for this is that hybrid algorithms such as [28] may be explored for further performance optimization.

While there are numerous reports on the use of AD in numerical simulation and

optimization, it is not yet a mainstream approach in industrial-grade, large-scale simulators [44, 7, 20].

Software Implementations

As outlined above, the algorithmic complexity of AD can be comparable with that of hand differentiation. Rather, the performance challenge for AD software is in the lack of implementation techniques that are simultaneously efficient, transparent, and notationally clear. Modern AD codes are implemented with either Source Transformation or Operator Overloading approaches.

Source Transformation.

Source transformation involves the development of a pre-processing parser that receives client code for the residual as input and generates code for derivatives. This automatically generated code can then be compiled and run. While several highly efficient, optimizing source transformers are available [9, 8, 58], the work-flow makes this approach less desirable. Moreover, conditional branches are difficult to treat and since we are interested in simulating cases where the physics may not be known prior to run-time, source transformation is not flexible enough.

Operator Overloading.

Operator Overloading involves two steps. First, new data-types are defined to store both the values of the variables as well as their derivatives. Second, arithmetic and fundamental functions are re-defined, by augmenting them with information on how to differentiate themselves and/or recording the sequence of operations performed on a so-called tape. For example, the multiplication operator knows to multiply its operands $a * b$, and to apply the product rule for the derivatives $\nabla a * b + a * \nabla b$. If the program starts by initializing independent variables with the appropriate unit derivative with respect to themselves, they can be used as operands creating dependent variables. This approach allows client developers to write programs for expressions in a natural manner, and with the full confidence that the appropriate derivatives are made. A challenge to this approach is efficient implementation.

As is well documented [12], the cost of the excellent notational convenience provided by Operator Overloading is generally too high. Because of the necessary return-by-value feature, every operator call in an expression results in the allocation of a temporary object on the heap. In turn, this temporary is assigned to, deeply copied by the calling source, and finally destroyed on exit. Moreover, for collections, one loop is performed per operator. These two events can cause substantial performance deterioration of more than an order of magnitude. This is due to the fact that the associated memory calls are expensive, numerous wasted cycles are expended to repeatedly store and load data from temporaries, and cache lines are continually reloaded from one level of the expression parse-graph to the other (*cache thrashing*) [12]. Several Operator Overloading AD packages are publicly available [35, 60, 6].

1.3 Automatically Differentiable Expression Templates Library ADETL

By example, this work introduces a software design and implementation philosophy that addresses the extendability and platform adaptability challenges ahead of the simulation software development world. The example addresses the challenges in architecting and developing the formulation and linearization guts of any simulation code. In particular, the proposed solution consists of an algorithmic framework and library of automatically differentiable data-types written in the C++ programming language.

The library provides generic representations of variables and discretized expressions on a simulation grid, and the data-types provide algorithms employed behind the scenes to automatically compute the sparse analytical Jacobian. Using the library, reservoir simulators can be developed rapidly by simply writing the residual equations, and without any hand differentiation, hand crafted performance tuning loops, or any other low-level constructs. Rather, formulation and linearization code written using the library, is self-documenting, and reads like language written by a physics and discretization domain expert.

A key challenge that is addressed is in enabling this level of abstraction and programming ease while making it easy to develop code that runs fast. Faster than any of several existing automatic differentiation packages, faster than any purely Object Oriented implementation, and at least in the order of the execution speed of code delivered by a development team with hand-optimized residuals, analytical derivatives, and Jacobian assembly routines. A second challenge is in providing a generic multi layered software framework that incorporates plug-in low-level constructs tuned to emerging architectures without the need to revise any code written by simulator developers using the library.

Chapter 2

The AD Core; Its Design And Evolution

A primary design objective of the ADETL is to provide a generic library of datastructures and algorithms that allows its users to develop discrete residual equations using a natural and self-documenting syntax. This design objective naturally precludes the use of Source Transformation [9, 8, 58] as an implementation option. Despite its efficacy to produce highly efficient AD code, Source Transformation requires that users instrument simulator code with pre-processor annotations. The annotations are intertwined with the underlying computer programming language and must demark the various differentiation contexts. This annotation process is simply not mathematically natural, and moreover, it severely restricts usage choices by requiring that they be made at compile time.

To achieve its design objectives, the ADETL core uses an Operator Overloading (OO) approach [35, 60, 6]. The ADETL provides users with the natural syntax of mathematical operators. There are numerous specific approaches to creating a functional, generic OO implementation of AD. While these various implementation approaches provide equivalent semantics from the library user's standpoint, they differ dramatically in terms of their computational performance. A primary technical contribution brought by the ADETL is a novel approach to OO implementations of AD.

ADETL's AD core operates behind the scenes. Unlike existing OO packages and published methods [35, 60, 6], ADETL combines computational performance that is on par with hand-coded differentiation routines with generic datastructures and algorithms, a natural vector calculus syntax, and a combinatorial level of flexibility. We present the linear evolution of the various OO techniques contained in the ADETL, and we compare and contrast their merits in various contexts. Before that though, we describe the common underpinnings of all OO implementations; specially defined AD datatypes, and the operator overloading skeleton.

2.1 Common aspects of all OO implementations

All OO implementations of AD consist of two components. The first component is a set of customized data-types that are used to represent various variables within a simulator. Unlike typical datastructures used to represent non automatically differentiable quantities such as those in Linear Algebra packages, the custom AD datatypes augment both the value and derivative states of the variables which they are used to represent. Using upper case letters to denote AD variables, they are a mathematic pair; $F = \{f, f'\}$. Within a simulator, all possible types of variables, intermediate terms, and residual equations that need to be stored belong to one of the following categories,

Univariate Scalar	$f : \mathbb{R} \to \mathbb{R}$	$f' = \frac{df}{dx}$,	
Multivariate Scalar	$f : \mathbb{R}^N \to \mathbb{R}$	$f'_i = \frac{\partial f}{\partial x_i}$,	$i = 1 \dots N$,
Univariate System	$f : \mathbb{R} \to \mathbb{R}^M$	$f'_i = \frac{\partial f_i}{\partial x}$,	$i = 1 \dots M$,
Multivariate System	$f : \mathbb{R}^N \to \mathbb{R}^M$	$f'^j_i = \frac{\partial f_i}{\partial x_j}$,	$i = 1 \dots N$, $\quad j = 1 \dots M$.

A critical functionality of any custom AD datatype is to enable the user to explicitly control and choose constant, independent, and dependent states. A constant state is simply one in which a variable has a zero gradient. On the other hand, independent variables are the numerical unknowns in any sub-context of the residual computation flow. A characterization of an independent variable state is that its derivative

with respect to every other variable in its context is zero. For independent univariate scalars, the gradient is one. Similarly, for independent multivariate scalars and univariate systems, the gradient is an elementary unit vector, e_i. Finally for an independent variable represented as a multivariate system the gradient is the identity matrix.

The second component is a complete set of mathematical operators that are defined to perform the appropriate evaluation and differentiation, given AD datatype arguments. Using capital letters to denote augmented AD variables and operations on them, the binary mathematical operators that need to be defined are,

$$\begin{aligned} A \pm B &= \{a \pm b,\ a' \pm b'\}, \\ A \times B &= \{a \times b,\ a' \times b + a \times b'\}, \text{and} \\ A \div B &= \{\frac{a}{b},\ a' \times \frac{1}{b} - \frac{a}{b^2} \times b'\}. \end{aligned}$$

Moreover, a complete AD implementation requires that the most commonly used transcendental, exponential, and hyperbolic unary functions be overloaded. Moreover, overloaded definitions of all unary operators must assume that the chain rule is necessary since this is not generally known at compile-time. That is, let $\mathcal{F}$ denote a functional with Frechet derivative $\mathcal{F}'$, then,

$$\mathcal{F} \circ B = \{\mathcal{F} \circ b,\ (\mathcal{F}' \circ b) \times b'\}.$$

With these two components in place, OO AD implementations a simulator developer is free to set any differentiation context, such as a discrete residual evaluation routine, or a sub-context such as a flash calculation within a residual routine. The context is set by first declaring all variables using the custom AD datatypes. Next, the independent variables are specified. Within the AD community, this process is known as setting the *differentiation seed*. Finally, the user can simply write the desired evaluation equations using the custom datatypes. The differentiation occurs behind the scenes, and is performed automatically by the overloaded operators. The user does not need to write any differentiation formulas.

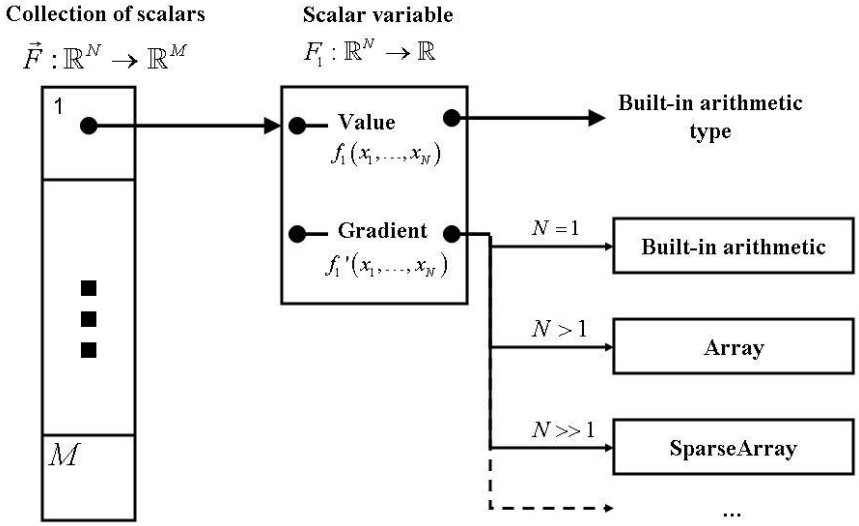


Figure 2.1: An illustration of the composite datastructures within ADETL. The outermost structure is a collection of differentiable scalars. Differentiable scalars store function values as well as gradients. The Jacobian matrix is formed by simply augmenting the set of gradients as its rows.

While we present the available options and specifics of ADETL in another section, we need to introduce a few specifics in order to develop the underlying algorithms and strategies behind the ADETL core.

2.1.1 Customized AD data-types

Figure 2.1 illustrates the design choice of datastructure composition within the ADETL. The ADETL uses collection datastructures for all systems. Examples of collections are arrays and dynamically re-sizable vectors or lists. Each element in a collection can be any type of ADETL scalar variables. The Jacobian of a system is formed by taking the union of gradients of the scalar elements. In essence, this allows the ADETL core to concentrate its AD operator functionality to scalar arguments. operators that are overloaded for systems simply delegate the AD functionality to scalar operations on an element by element basis. A basic ADETL collection datastructure is the `ADvector< T >` template, which uses the datatype identifier `T` to parameterize the vector's element datatype. The only restriction resulting from this design choice

is that one cannot create systems where one entry is a univariate variable, whereas in another entry it is a multivariate. Instead, the more general element type needs to be used throughout for all entries or equations in the vector.

Differentiable scalars are the atomic datastructure of the ADETL core. As depicted in Figure 2.1, `ADscalar< T >`s are comprised of a *value* and a *gradient*. The template datatype parameter, `T`, is used to specify the intended choice of datastructure to use for the `ADscalar`'s gradient. This design choice models a generic paradigm that facilitates the separation of datastructure and algorithm. This way, all scalars, regardless of whether they are multivariate or univariate, can be treated universally. Suppose that we have available to us a dense array datatype called `Array`, and a sparse alternative called `SparseArray`. Then, we can declare a univariate scalar as the datatype `ADscalar< double >`, a multivariate with a dense gradient as `ADscalar< Array >`, and finally, a multivariate with a sparse gradient as `ADscalar< SparseArray >`. Note that the choice of gradient type not only allows us to model different domain dimensions and sparsity choices, but also the underlying implementations may be different also; for example one can drop-in parallel arrays as a variation. Listing 1 provides a minimal declaration of an `ADscalar<>` template which we will use throughout this discussion to illustrate the underlying concepts.

The declaration provides facilities to construct variables using statements such as,

```
ADscalar< double > X(1.0,2.0);   // univariate
ADscalar< Array > Y(1.0, 10 );   // multivariate with 10 unknowns
ADvector< ADscalar< Array > > Z( 100, X ); // system of 100 equations
```

Aside from facilities to create and initialize scalars, the `ADscalar<>` datatype also provides access to its value and derivative, and defines the assignment operator, given a right hand side value, and allows one to set the activation status.

2.1.2 Overloading Operators in C++

The second component of any specific OO approach is to redefine the fundamental unary and binary mathematical operators so that they perform both their respective

Listing 1 A minimal implementation of an AD scalar variable data-type.

```
template< typename T >
class ADscalar
{
protected:
  T        m_Gradient;
  double   m_Value;

public:
  // Constructors that allocate and initialize ADscalars
  ADscalar( double inValue = 0.0, T inGradient = T() );
  ADscalar( ADscalar< T > & inRHS );

  // Set a variable's AD activation or seed
  void make_independent( int _ID );
  void make_constant( );

  // Access to the value and derivative information
  double &  value( );
  T &       gradient( );

  // Assignment
  void operator= (const ADscalar<T> & in_RHS);
  ...
};
```

value operations, as well as the derivative of the operation. With this infrastructure, users can write self-documenting automatically differentiation expressions. Consider as an example adding three univariate scalars using the statements,

```
ADscalar<double> X(1.0,2.0), Y(2.0,1.0), Z(0.0,1.0);
ADscalar<double> W = X + Y + Z;
W.value();              \\ returns 3.0
W.gradient();           \\ returns 4.0;
X.make_independent();
Y.make_constant();
W = X + Y + Z;
W.value();              \\ returns 3.0
W.gradient();           \\ returns 2.0;
```

Programming languages such as C++ offer specific facilities for the programmer to provide multiple definitions for any built-in operator or function. At compile time, the usage context is analyzed and the most appropriate operator definition is selected. This choice is then applied in-place for compilation. The context-specific choice depends on the datatypes of the call's arguments. OO is a convenient infrastructure with which to implement AD; AD can be implemented by simply providing overloaded operators to be used for all calls where one or more of the arguments is an AD custom datatype. All fundamental arithmetic unary and binary operators are redefined specifically to accept the custom AD datatypes as the arguments and to perform both the value operation as well as its derivative. This is the basis of all OO implementations of AD, including all of those in the ADETL. There are numerous specific ways to define the actions of the overloaded operators however, and these choices dictate critical performance issues. Before moving on to developing and exploring various strategies, we present the common aspects across them.

Binary operators such as the addition operator are overloaded using the general form,

```
T3 operator+ ( const T1 & _A, const T2 & _B )
{
```

```
  // create a temporary object of type T3
  // temporary object encodes or stores the result and its derivative
  // return a copy off the temporary object;
};
```

where `T1` and `T2` are the argument datatypes, the combination of which determines which definition to use in any given context. The datatype `T3` is the return type of the overloaded operator and it is the choice of this datatype that primarily makes one OO approach differ from another. For example, in the most basic approach, AD is evaluated directly, and the arguments as well as the return types are AD datastructures. On the other hand, more sophisticated approaches use operators to build lightweight representations of an expression's intended parse graph. In that case, the return type is the graph encoding datatype, and several operator definitions must be provided to support operations with graph encoding datatype arguments.

Similarly, unary operators such as the assignment operator are overloaded using the form,

```
T2 operator= ( const T1 & _RHS )
{
  // assign value and derivative from _RHS to self
  // return a copy of self;
};
```

where, once again, the argument type `T1` of the usage context determines which operator definition gets used. In the case of direct evaluation, this type is always an AD datatype. On the other hand, approaches that rely on building parse graph representations use a parse graph encoding argument datatype, and the assignment operator is itself responsible for coordinating the evaluation process.

Pairwise Evaluation refers to the specific mechanism with which compiled languages such as C++ operate arithmetic expressions. It is a direct consequence of this inherent feature that each operator instance, such as the addition operator above, produces some form of temporary storage to hold the resultant of the operation.

The scope of this temporary object is the duration of the expression itself. Pairwise Evaluation requires that expressions be executed one operator at a time using a well-defined order. The order of execution is determined by an *Operator Precedence* schedule. For example, since multiplication has a higher precedence than addition, the expression `x * y + z` is mathematically equivalent to `(x * y) + z` rather than to `x * ( y + z )`. Moreover, for operators with the same precedence, all operators besides unary and assignment operators are left-associative; for example `x + y + z` is equivalent to `( x + y ) + z` rather than `x + ( y + z )`. In order to produce binary instructions that execute a pairwise evaluation process, compilers must introduce temporary intermediate resultants for each operator. This has potentially serious performance implications, particularly for expressions that involve non-trivial user-defined arguments and overloaded operators. For example, the expression `w = x + y + z`, is executed as the three step sequence,

```
t1 = x + y
t2 = t1 + z
w  = t2
```

which involves two temporary objects, `t1` and `t2`, created by each of the two operator instances. If the arguments, `x`, `y`, and `z` are built-in types, such as floating point numbers, then the operators that are used are the built-in ones, and the execution sequence that is generated is close to optimal. The intermediate temporaries that are generated are floating point numbers and are placed on the stack. Depending on the precise target architecture and compiler, the execution results in approximately three to four read and write machine instructions, and two floating point operations. On the other hand, suppose that the arguments are objects of a user-defined datatype such as univariate AD variables, `ADscalar< double >`. In that case, at compile time, a search is conducted for the most appropriate operator definition to use. In this case, that turn out to be the template,

`T3 operator+ ( ADscalar<double> &_A, ADscalar<double> &_B)`.

The execution chain that is generated by the compiler in this case is necessarily more complex, and as we shall discuss, is often far from ideal. The temporary objects in

this case, are of whatever the return type of the overloaded operator happens to be.

A final piece of background regarding all OO implementations is that certain rules from mathematics cannot be assumed. For example, the commutativity of operators including addition should not be assumed. That is while the floating point expression `a + b` is equivalent to `b + a` barring rounding error, this is not a requirement on user defined operators. Because of this fact, compilers may not be able to optimize the sequence of overloaded operators as they would for built-in arithmetic. For example, if a variable `a` is of type `double`, modern optimizing compilers will typically replace the arithmetic sub-expression `a+a+a+a` with the less costly term `4*a`. On the other hand, in the same example, if the argument `a` happens to be a user defined datatype, compilers cannot make the same assumption. Another short-circuited optimization is *common sub-expression elimination*. As of the time of writing, there is no wide support for means to indicate to the optimizer which mathematical rules are safe to assume.

2.2 OOAD by Instantaneous Evaluation

The most basic OOAD approach is instantaneous or direct evaluation. When an arithmetic expression involving any AD datatype is parsed, the compiler chooses AD overloaded operators that evaluate the operation and its derivative directly, producing an AD resultant one operator at a time. The templates facility affords the possibility of writing one set of overloaded operators for all AD scalar types. This is accomplished by making the gradient datatype a template parameter. For the purposes of this discussion, we restrict attention to homogeneous expressions; that is expressions involving arguments with the same gradient datatype. The comprehensive OOAD approach within the ADETL uses a compile-time metaprogramming technique known as type traits promotion to provide the pre-processor with the most appropriate resultant datatype given two arguments with different gradient datatypes, e.g. adding two scalars, one with a dense gradient, and the other with a sparse one producing a dense resultant. In this scheme, binary operators such as multiplication are written as,

```
template< typename T >
ADscalar< T > operator* ( const ADscalar<T> & _A,
                          const ADscalar<T> & _B )
{
  ADscalar< T >  t1;
  t1.value()    = _A.value()    + _B.value();
  t1.gradient() = _A.value() * _B.gradient() +
                  _A.gradient() * _B.value();
  return t1;
};
```

The direct OOAD implementation is rather straight-forward. Unfortunately, a closer inspection of the execution chain that OOAD produces quickly reveals serious performance concerns for multivariate variables. Owing to the pairwise evaluation process, the execution chain typically results in a several-fold degradation in performance over hand coded operations. Next we walk through illustrations of these execution chains for univariate terms and for multivariates.

2.2.1 Univariate expressions

Within simulators, univariate variables may be declared frequently to store phase and component properties within individual simulation cells. For example, in an isothermal Black Oil setting, phase density and viscosity can be modeled to depend on pressure only. Subsequently such variables are most suitably stored in univariate scalar entries rather than say a sparse multivariate with a single nonzero element all the time. It is common to use nonlinear analytical expressions to compute these variables. Moreover, the call frequency of evaluating these quantities is substantial. For this reason, it is critical for an AD implementation to perform univariate computations efficiently. We compare the instructions generated by a hand written univariate expression with that generated by the OOAD approach.

Mathematically, the variable we hope to evaluate is,

$$\rho = \left\{ e^{(P-Pr)^2},\ 2\,(P-Pr)\,e^{(P-Pr)^2} \right\}.$$

First, a hand-crafted implementation may be,

```
const double dp        = p - pr;
const double rho       = exp( dp * dp );
const double d_rho_d_p = 2.0 * dp * rho;
```

It involves a single exponential evaluation and four simple operations. Only a couple of temporary variables are explicitly declared on the stack. Now consider the OOAD statement,

```
const ADscalar< double > RHO =  exp( (P - pr) * (P - pr) );
```

The statement evaluates the value and derivative of density while appearing more compact and self-contained compared to the hand-coded alternative. Proceeding by evaluating the expression one operator at a time, inlining the overloaded instructions, the simple OOAD statement generates the following instructions,

```
double t1, t2;
t1 = P.value() - Pr;
t2 = P.gradient() - Pr;
double t3, t4;
t3 = P.value() - Pr;
t4 = P.gradient() - Pr;
double t5, t6;
t5 = t1 * t3;
t6 = t1 * t4 + t2 * t3;
double t7, t8;
t7 = exp( t5 );
t8 = exp( t5 ) * t6;
RHO.value() = t7;
RHO.gradient() = t8;
```

Unlike the hand-coded alternative, the instruction pipeline length is significant, and the amount of stack space explicitly declared is almost three times larger. This may seem like a serious issue at first. However, realizing that on most architectures, there is ample stack space, and floating point operations are not a bottleneck compared to memory bandwidth, this is actually not a terrible idea. The performance penalty of the notational convenience is minimal. The situation is far more bleak however for multivariates.

2.2.2 Multivariate expressions

Compositional models typically require the evaluation and differentiation of intermediate quantities that depend on several independent unknowns. For example, the density of a phase in a simulation grid block depends on component fractions as well as pressure. The number of components in such models typically vary from a handful and up to $O(10^2)$. Dense multivariate variables are the most suitable datastructure for such variables. Similar to the use of univariate scalars, the call frequency for these operations is substantial, and efficiency is important. Since the root of the performance issues concerning the use of OOAD are similar for both sparse and dense multivariates, we focus this discussion on dense problems.

Listing 2 provides a bare-bones definition of a dense `Array` data-type which is used to represent the gradient. The constructor, which is invoked whenever an array object is declared, dynamically allocates memory for, and initializes the array. In this example, for the sake of simplicity, the memory allocation is achieved by direct system calls which are costly operations. In the ADETL, all dynamic memory management is provided by fit-for-purpose, hand-crafted memory allocators which typically amortize allocation costs. Nevertheless, constructor calls are relatively costly operations especially considering that they also involve an initialization loop. The destructor frees memory back to the system and is evoked when

To examine the performance of OOAD, consider the simple dense multivariate AD expression,

$$W = \{f * g + h,\ f' * g + f * g' + h'\},$$

Listing 2 A bare-bones implementation of an Array data-type.

```
class Array {
protected:
  const int N;
  double* p_data;
public:
// Constructor allocates memory dynamically
  Array( int _N, double _val = 0.0 ) : N(_N)
  {
    p_data = new double [ N ];
    for ( int i=0; i < N; ++i ) p_data[ i ] = _val;
  }
  // Destructor returns memory
  ~Array( ) { delete [] p_data; }
  // Overloaded assignment operator
  void operator= ( const Array & _RHS)
  {
    for( int i = 0; i < N; ++i ) p_data[ i ] = _RHS[ i ];
  }
  ...
};
```

where the gradients, f', g', and h', are N-dimensional vectors. A hand coded implementation of a routine to evaluate and differentiate this expression would be,

```
w = f * g + h;
for ( int i=0; i < N; ++i ) {
   Dw[ i ] = g * Df[ i ] + f * Dg[ i ] + Dh[ i ];
}
```

The statements produce efficient instructions involving one loop with a decent floating point operation density over the required memory access demands and loop overhead. Compare the semantics of using an OOAD expression to yield the same result; `W = F * G + H` where the arguments are all of type `ADscalar< Array >`. Unfortunately, the overhead of this syntactic convenience is substantial, and the deceivingly simple OOAD statement generates code similar to,

```
// multiplication; F * G
double t1;
double *d_t1 = new double [ N ];
for ( int i=0; i < N; ++i ) d_t1[ i ] = 0.0;
t1 = F.value() * G.value();
for ( int i=0; i < N; ++i )
   d_t1[ i ] = G.value() * F.gradient[ i ] +
               G.gradient[ i ] * F.value();

// addition; F * G + H
double t2;
double *d_t2 = new double [ N ];
for ( int i=0; i < N; ++i ) d_t2[ i ] = 0.0;
t2 = t1 + H.value();
for ( int i=0; i < N; ++i )
   d_t2[ i ] = d_t1[ i ] + H.gradient[ i ];

// Assignment operation
```

```
W.value() = t2;
for ( int i=0; i < N; ++i ) W.gradient[ i ] = d_t2[ i ];

// Clean-up temporaries
delete [ ] d_t2;
delete [ ] d_t1;
```

The example shows that the use of OOAD introduces severe inefficiencies in this case. First, two temporary arrays need to be allocated, initialized, and de-allocated. Second, a total of five loops are executed compared to just one. Moreover, the floating point operation density within each loop is small. Finally, the number of indirect addressing operations is almost double what it could be. Aside from the obvious performance implications of the additional dynamic memory calls, Operator Overloading can cause memory to go out of cache sooner than it would otherwise. It can also cause register spills due to the excessive counters and temporary resultants that are required. It can also cause performance degradation by disrupting function inlining since the instruction density is extended.

In order to overcome these performance issues, a completely different strategy is required. Rather than directly evaluate expressions resulting in numerous loops, the main idea is to instead build up a parse graph of the expression and to delay its evaluation until the entire graph is available. At that point there is at least some hope that the execution can occur in one fused-loop.

2.3 Lazy AD with Expression Templates

Direct evaluation leads to acceptable performance penalties for univariate problems. The situation is far worse though for multivariate expressions. In such cases, the pairwise evaluation process leads direct evaluation to generate excessive temporary gradients and loops. The fundamental essence of the problem is that the derivative evaluation process occurs before the whole context of the expression has been processed. As a first advancement from direct evaluation, we develop a framework

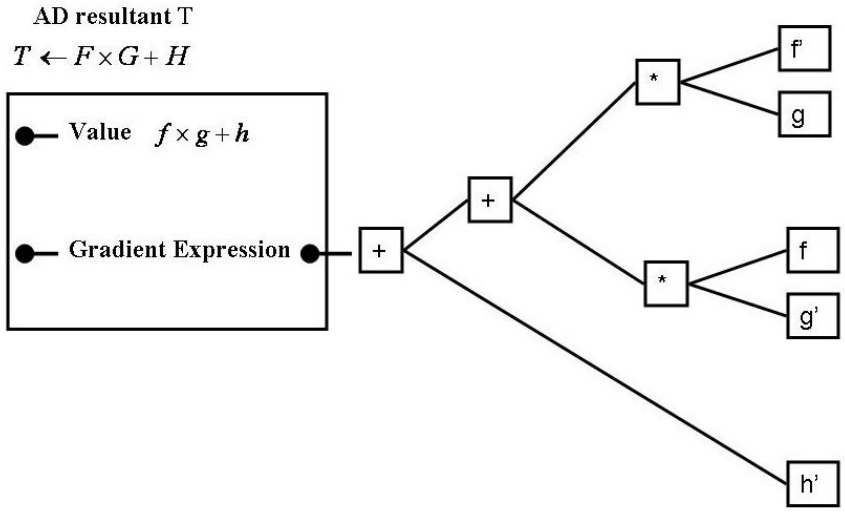


Figure 2.2: The basic Expression Templates resultant, T, contains the expression's value and the parse graph representation of its derivative.

that exploits the convenience of Operator Overloading while overcoming this inefficiency. This is accomplished by the use and extension of a generic metaprogramming technique known as Expression Templates (ET) [45, 70, 42].

The basic idea is to design an OO implementation that generates graph datastructures which encode the parse graph of the derivative, while directly evaluating the value of the expression at hand. For example, consider the statement $F * G + H$ once again. Upon execution, we hope to generate the datastructure depicted in Figure 2.2. This implies that the derivative is not evaluated yet since it is not needed so far. The latest point in the execution flow at which the derivative is required is upon assignment of the expression to an `ADscalar`; $W \leftarrow T$. With ET, the derivative evaluation is contained within the assignment operator of the gradient, and since the entire parse graph is available, it is conceivable at least, that the computation can be executed as efficiently as a hand-coded alternative.

As a design choice, the ADETL introduces another *ADscalar* type that has a parse graph as its gradient. For example, the overloaded multiplication operator becomes,

```
ET3 operator* ( const ADscalar< ET1 > _A,  const ADscalar< ET2 > _B )
```

where `ET1` and `ET2` are the sub-graph datastructures on the arguments, and `ET3` is the datatype of the product of the two sub-graphs. Note that it is the actual datatype that encodes the parse graph. Next, we discuss the internal workings of how such gradient expressions are built, followed by a presentation of how they are evaluated for dense gradients and for sparse problems.

2.3.1 Building Expression Templates at compile time

The ET technique was first described by [70], and independently by [67], and in the scientific computing context by [40] and [45]. To date, the major scientific computing applications which capitalized on this technique are predominantly in the Linear Algebra and other mid-layer generic data-type settings. It makes heavy use of the templates feature of the C++ language in order to eliminate the overhead of operator overloading on abstract data-types, and without obscuring notation. Production grade codes include the Parallel Object Oriented Methods and Applications (POOMA)[42] collection, the Matrix Template Library (MTL)[62], and the Blitz++[71, 69] vector arithmetic library. These efforts are often credited for demonstrating the emergence of the C++ programming language as a strong contender to Fortran in terms of performance in scientific codes [39, 57, 68].

At the time of development of the ET technique however, compiler support and debugging protocols for generic- and metaprogramming were limited. Only a handful of compilers could pragmatically handle such code. In contrast, today, all of the major popular optimizing compilers list pragmatic optimizations to compile heavily generic code. Given the success of ET in these application areas, and the continually improving compiler support, we seek to integrate the technique within an Operator Overloading AD framework.

Perhaps the most performance critical aspect of the ET approach is to ensure that the graph representation is lightweight enough to allow compilers to easily replace them with code that is as close as possible to a single hand written fused loop. Subsequently, the graph cannot be built dynamically since the required additional memory management and indirect addressing costs would be overwhelming. Rather,

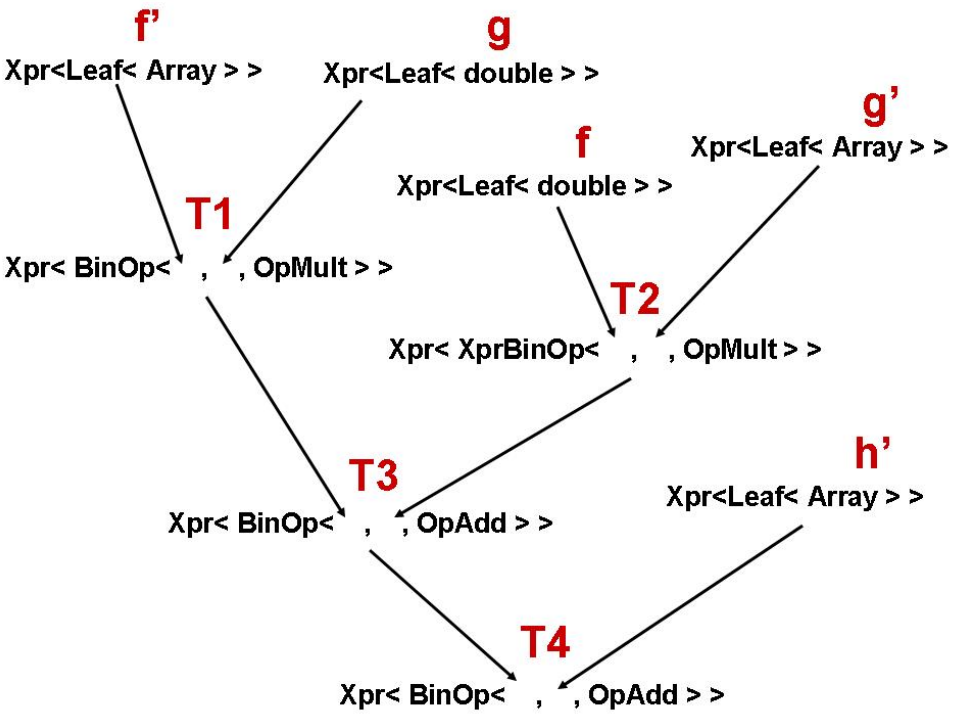


Figure 2.3: The parse graph template type composition generated by an ET evaluation of $f' \times g + f \times g' + h'$.

the parse graph representation would ideally be encoded into a datatype that is deduced by the compiler. The ET technique is in essence a metaprogramming approach that encodes the parse graph node by node into a datatype. To make the approach more apparent, we walk-through an example.

Recall the AD expression,

$$F \times G + H = \{f \times g + h,\ f' \times g + f \times g' + h'\},$$

where the gradients, f', g', and h', are N-dimensional vectors. Using `Array` datastructures to represent the gradient vectors, the gradient portion of the expression could be implemented using the statement;
`F.gradient() * G.value() + F.value() * G.gradient() + H.gradient()`.
In the ET approach, the operators in the above statement are overloaded to generate

a parse graph datatype that is built-up at compile-time along the pairwise evaluation process. This is achieved by making the temporary storage resultants of each operator to be a node in the graph. Figure 2.3 illustrates this process. In the figure, the nodes `T1`, `T2`, `T3`, and `T4` are the datatypes of the intermediate temporary resultants. In terms of a sequential chain of events, the statement produces the following instructions,

```
T1 t1( F.gradient(), G.value() ); // f' * g
T2 t2( F.value(), G.gradient() ); // f * g'
T3 t3( t1, t2 );                  // f'*g + f*g'
T4 t4( t3, H.gradient() );        // f'*g + f*g' + h'
```

where the lower case temporaries are the actual objects of each of the datatypes in Figure 2.3. Notice, no evaluation of derivatives is performed yet. All that is executed is a compile time deduction and runtime execution of a parse graph. Next, we walk through the ET engine that generates ET parse graphs.

Each node in an ET graph is wrapped by a universal intermediary datatype `Xpr< N >`. The use of this wrapper allows all nodes to be referred to generically by parameterizing the internal details of the node with the template parameter `N`. The `Xpr< N >` simply delegates all data query, access, and arithmetic operations to the specific node datatype `N` which it wraps. A sample declaration of such a wrapper node is,

```
template< typename N >
struct Xpr{
  const N &m_node;
  Xpr( const N & _node );
};
```

The declaration reveals a performance critical detail; the wrapper, and all other ET graph datatypes store references or memory addresses of child nodes rather than copies. This is a critical optimization since otherwise, the sequence of temporaries allocated on the stack by an expression would grow in size recursively without an

upper bound. That could lead to disastrous bottlenecks and no compiler would have any hope of optimizing away a bottomless recursion stack. The trade-off in this choice is rather subtle. The ET graphs produced in this manner cannot be explicitly stored for later use. This is because according to the standard, the lifetime of intermediate node objects is the duration of the statement. After that point in the execution chain, the addresses in memory become invalid and attempts to access that memory leads to undefined behavior.

There are generally three types of internal detail node wrapped by an `Xpr< N >` object:

1. Leaf nodes store a reference to one argument or operand within an arithmetic expression. As the name suggests, such nodes are terminal leaves of an ET parse graph. For the example depicted in Figure 2.3, all gradients and values are represented by `\Leaf< Array >` and `Leaf< double >` types respectively.

2. Binary operators are overloaded to generate a `BinOp< Left, Right, Op >` node. This node is a generic binary operation node, where the `Left` and `Right` template parameters are the datatypes of the left and right arguments of the operator, and the `Op` is a functor that applies a certain operation such as addition.

3. Similarly, unary operators are overloaded to generate a `UnryOp< Arg, Op >` node, where `Arg` is the datatype of the operators argument, and `Op` is an encoding of the actual unary operation to be performed.

While the applicative functors that appear in binary and unary operation nodes are datatypes, they never actually get instantiated, i.e., no stack memory is required for such variables. Rather, they provide a generic service through a static routine that has the appearance of a global function and the behavior of an inlined statement as far as efficiency. As an example, consider an applicative addition functor;

```
template< typename T >
```

```
struct OpAdd{
  static inline T apply( const T & _l, const T & _r )
    { return _l + _r; }
};
```

The final piece of a minimal implementation is the overloaded operators themselves which kick-off the sequential construction of the expression templates. There are numerous technicalities involved in their definition, including the need for *template partial-specialization* to account for all possible combinations of operand types; an expression and an array, two arrays, etc. Pre-processor macros maybe applied to automatically generate such templates prior to compile-time. For example, an overloaded addition operator that handles addition of a sub-expression node to a terminal array, is,

```
template< typename L >
  Xpr< XprBinOp< Xpr<L>, Leaf<Array>, OpAdd > >
  operator+( const Xpr<L> &_l, const Array &_r )
  {
    return Xpr( XprBinOp( _l, _r, OpAdd ) );
  }
};
```

With all the pieces in place, arithmetic expressions involving multivariate AD scalars produce a temporary AD scalar that has an ET graph for a gradient. No evaluation takes place until it is needed; whenever an assignment operator is encountered. At that point, the goal is to perform the gradient computation in one fused loop. The technique to achieve this for dense multivariates follows directly from the standard ET technique. On the other hand, sparse multivariates cannot operate in the same way, and ADETL introduces a novel graph traversal algorithm to perform this. We discuss the ET evaluation process for dense and sparse gradients independently.

2.3.2 ET Evaluation; Dense Multivariates

ET graphs are built along the pairwise evaluation process of expressions involving multivariate gradient datatypes. The evaluation of the expression is triggered by an overloaded assignment operator. The goal of the evaluation process is to perform the vector expression using one fused loop. For dense vector representations, this is turns out to be both straightforward and a rather effective strategy.

At each iteration, $0 \leq i < N$, the overloaded version of the `Array` assignment operator queries the root ET node for its i^{th} entry and assigns it in place. Recall that there are three types of nodes that make up an ET graph; Leaf nodes, binary operators, and unary operator nodes. In the dense ET set-up, the actions which occur when a node is queried for an entry depends on its type as follows.

1. Terminal leaf nodes referring to a gradient array return the i^{th} component of the gradient. On the other hand, terminal leaf nodes referring to a scalar return the scalar itself regardless of the index i.

2. Binary operator nodes query the left and right child nodes for their i^{th} entry values, and then they perform the binary arithmetic operation and finally, they return the result.

3. Unary operator nodes query the child node for its entry, perform the unary operation on the result, and return to the caller.

The result is that each iteration of the fused assignment loop results in a reverse traversal of the ET graph. We return to the example statement,

```
w' = f' * g + f * g' + h'
```

Provided that an optimizing compiler is capable of eliminating the overhead of function calls, the instruction sequence generated by the iterative graph traversal process is almost identical to that of a hand-crafted alternative. Assuming that binary operation nodes always query the left child first, it is straightforward to trace the execution

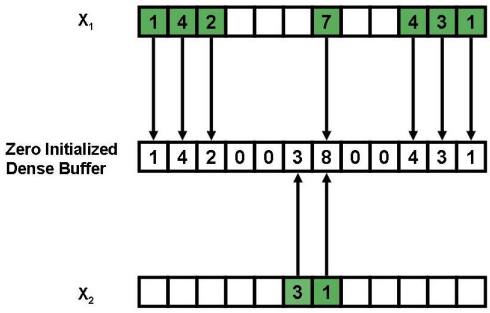
pipeline for the example at hand using Figure 2.3. In this case, each iteration of the loop embedded in the assignment operator would perform the following instructions,

```
t1 = f'[ i ] * g
t2 = f * g'[ i ]
t3 = t1 + 2
t4 = t3 + h'[ i ]
w'[ i ] = t4
```

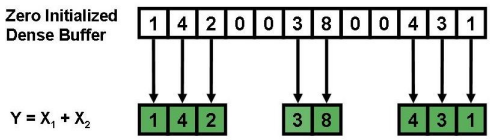
The graph construction phase of the ET framework is oblivious of the underlying gradient datastructure. The evaluation process on the other hand inherently relies on the underlying datastructure. In the example above, it is apparent that the dense evaluation procedure relies on the indirect accessing property of dense arrays. The i^{th} entry is defined for leaf nodes, and subsequently, for all internal nodes as well. Moreover, the length of the iteration is known prior to the evaluation of the expression. Sparse array data-types cannot be queried sequentially for elemental entries since otherwise, all evaluation processes equate to dense ones involving logical zeros. For these reasons, a further development is required in order to realize a sparse ET evaluation framework.

2.3.3 ET Evaluation; Sparse Multivariates

Sparse gradients datastructures form the backbone of Reservoir Simulation Jacobian evaluation routines. The ADETL provides a generic `SparseVector< T >` template which is parameterized by its nonzero element type, `T`. The element type could be a scalar, producing a point sparse vector, or a dense block, producing a block sparse datastructure. The ET operator overloads described for dense arrays can also be used for expression involving sparse gradients. Subsequently the build-up of ET graphs with sparse leaves occurs in very much the same way as it does with dense expressions. The exact function and contents of the three types of expression nodes however are different to accommodate the sparse evaluation process.



(a) Prolong Phase; nonzero entries of each argument are added into a zero initialized dense buffer sequentially.



(b) Restrict Phase; the resultant dense buffer turned into a sparse array.

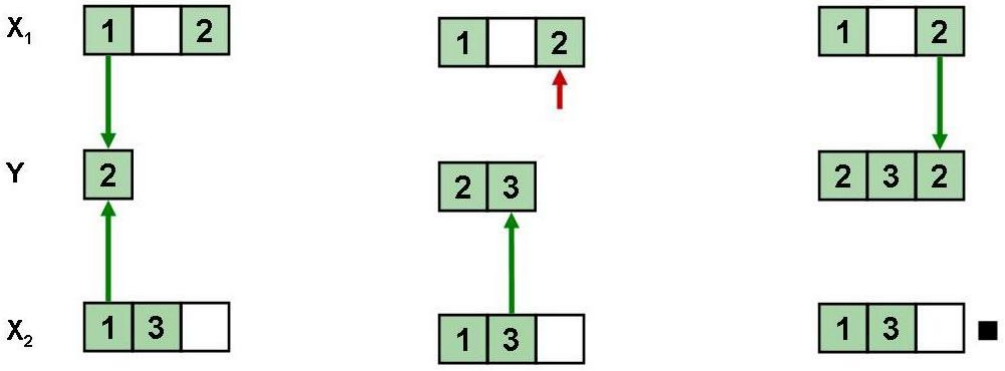
Figure 2.4: An illustration of the two phases of an axpy routine to evaluate the sparse vector expression $y = X_1 + X_2$.

The ET graphs provide a complete and simultaneous access to all arguments. The possibility exists to compute the result of the expression using a one pass iteration that minimizes the number of memory accesses, and branch logic evaluations, and that introduces no dynamic memory needs.

Without the ET facility, traditional sparse computations involve at most two sparse vector arguments. Multi-way computations (those involving multiple arguments) are executed by repeating the two-way process several times over intermediate resultants in essentially a pairwise operator evaluation process. For example, the `axpy` routine of the standard sparse Basic Linear Algebra Subprograms (BLAS)[26] executes the mathematical expression $ax + y$ where x and y are arrays and a is a scalar. By using the routine repeatedly with y as the target, we can compute any sparse vector linear combination $y = a_1x_1 + \ldots + a_kx_k$. To motivate the multi-way sparse evaluation algorithm developed for the ADETL, we first recall the two standard two-way combination algorithms.

Standard two-way sparse vector algorithms

The first algorithm is modeled after the most common sparse BLAS `axpy` implementation. The algorithm is a two stage process. In the first stage, sequentially, each sparse array argument is added into a zero-initialized dense buffer. In the second



(a) Coinciding entries are added. Both running pointers are incremented.

(b) Entry with smaller column index is inserted and its running pointer is incremented (green) while the other is not (red).

(c) Since one argument is completely traversed, the remaining entries in the other argument are just inserted.

Figure 2.5: An illustration of a one-pass routine to evaluate the sparse vector expression $y = X_1 + X_2$.

stage, the entire dense buffer is traversed to deduce non-zero entries and to thereby produce a sparse resultant. This process is illustrated in Figure 2.4 for the sparse expression $y = X_1 + X_2$. It is easy to see that this algorithm performs poorly whenever the dimension of the required enclosing dense buffer is very large compared to the number of non-zero entries in the resultant. On the other hand, when that is not the case, this algorithm is very effective as it uses heavier indirect addressing instead of an embedded branch within the iteration.

The second algorithm is a one-pass approach that uses a running pointer to each argument. Initially each of the two pointers is bound to the first non-zero entry of its respective sparse vector. While both running pointers have not traversed the entire vector, the following sequence of operations is performed. The column indices of the two running pointers are compared. If the nonzero entry column indices are equal, the two entries are operated on and inserted into the resultant. On the other hand, if they are not equal, then the entry with the smaller column index is inserted, and its running pointer only is advanced. At the end of the iteration, if one of the two sparse arrays involves any remaining untraversed entries, they are simply inserted into the

resultant. Figure 2.5 illustrates this process for the problem of adding two sparse arrays using this process.

With an ET graph on hand, all sparse vector arguments, and the arithmetic operations to be performed on them are parsed and available simultaneously at runtime. This introduces the possibility to extend two-way sparse vector algorithms onto multi-way versions which can reduce the asymptotic number of branch comparisons and associated memory reads. Such an extension would necessarily require the ET nodes to now maintain more information regarding state. Moreover, since the `axpy` approach is inherently serial, its fundamentally orthogonal to use with an ET graph. The sparse ET algorithm developed for the ADETL extends the one-pass process to a multi-way version.

ET multi-way sparse vector algorithms

The ADETL sparse ET evaluation algorithm exploits the availability of the parse-graph at runtime in order to asymptotically reduce memory bandwidth requirements. In particular, the ET graph generated by any expression that involves one or more `SparseVector< T >`s maintains state information that is cached at every node. This state information is initialized upon construction of the ET graph through the pairwise evaluation process; a forward topological sweep. The ET graph nodes need to also include additional logic as compared to the dense ET graph alternative. In particular, the three types of nodes that make-up ET graphs are designed as follows.

1. Terminal leaf nodes of sparse vectors must maintain a running pointer the is initialized to the sparse vectors first non-zero entry. Terminal leaves must also determine whether they are active. Leaves are active provided there are any remaining unprocessed non-zero entries.

2. Binary operator nodes maintain a current entry state. Such nodes can be thought of as intermediate resultants of non-zero entries. Moreover binary nodes need to be able deduce and set the activation of the two child nodes. A child node is active if the column index of its current non-zero entry is the leading

column of the two.

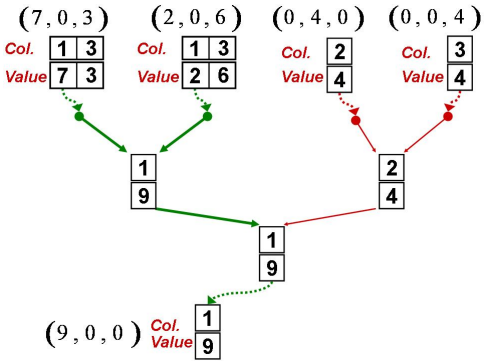
3. Unary operator nodes maintain an intermediate temporary non-zero entry as well as a active state label that is simply delegated to its child node.

Upon assignment, a single pass iteration is performed. At each iteration, two reverse topological sweeps are executed. The first is an *Analyze Phase* that labels the nodes in the graph as active or inactive. The goal of this labeling is to truncate any branch or sub-graph that ultimately results in a trailing nonzero entry. This is essentially the mechanism by which the number of index comparison operations is reduced over a direct pairwise evaluation. The second sweep is an *Advance Phase* whereby all active nodes are visited to evaluate their nonzero entry value, and to update the running pointers of the active leaf nodes. This is a single pass process much like the two-way one pass algorithm. The iteration continues so long as the parent node is active.

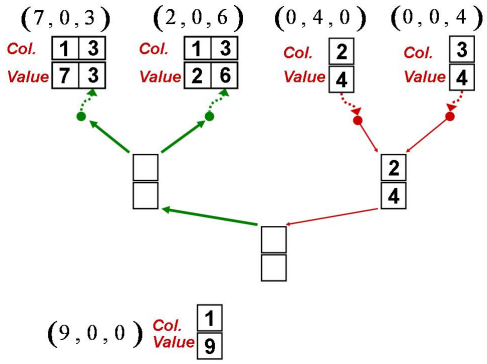
Illustrative example

To illustrate the solution process conceptually, we proceed by tracing through the ET evaluation process of a simple example. Suppose that a sparse AD scalar expression requires the evaluation of the sparse gradient statement $w = a + b + c + d$. The four sparse gradient operands are $a = (7, 0, 3)$, $b = (2, 0, 6)$, $c = (0, 4, 0)$, and $d = (0, 0, 4)$. The operands are represented by ADETL `SparseVector< double >` objects each of which consists of two arrays of equal size. One array contains the column indices of each non-zero entry, and the other array stores the corresponding values of the entries. For example, the representation of a has a column index array $(1, 3)$ and a value array $(7, 3)$.

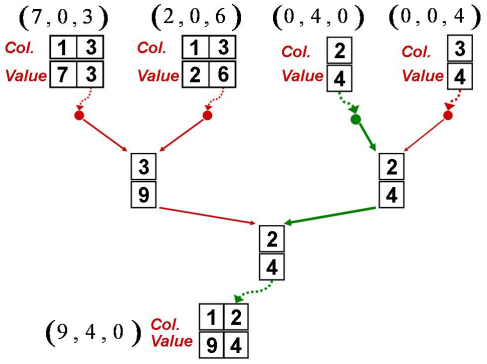
Similar to the dense ET graph construction process, the parser processes the sparse statement at compile-time and generates a topological sequence of the operations to be performed. When the `SparseVector` overloaded assignment operator is encountered, three iterations are performed in this case. Each iteration executes an



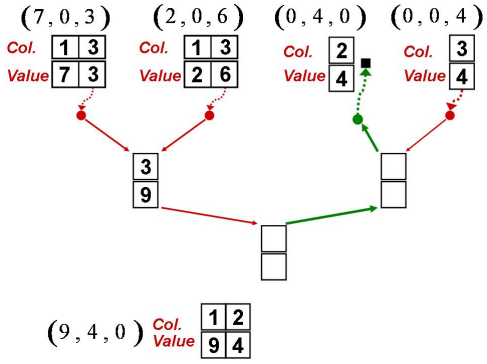
(a) Advance 1: three index comparisons



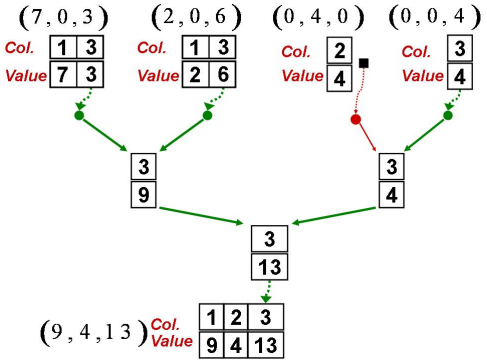
(b) Analyze 1: two leaves are incremented



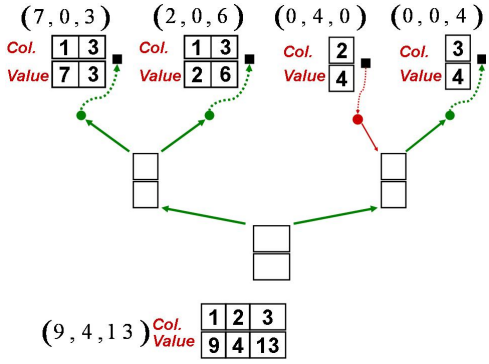
(c) Advance 2: two index comparisons



(d) Analyze 2: one leaf is incremented



(e) Advance 3: two index comparisons



(f) Analyze 3: three leaves are incremented

Figure 2.6: Evaluation sequence of a hypothetical sparse ET graph with 4 arguments, and a total of 3 logical nonzero entries. Red, thin arrows indicate branches of the graph that are inactive. Green, thick arrows indicate active branches involved in the current nonzero entry.

Advance and an Analyze phase. The sequence of state snapshots along this process are depicted by Figure 2.6. In the figure, the dotted curved arrows represent the four running pointers to the expression operands. The solid arrows represent connections amongst ET nodes. Red, thin arrows represent inactive connections while green, thick arrows represent active ones. In the ET graph, there are three binary nodes, all of which are addition nodes. The internal binary nodes maintain intermediate temporary states. That is, each node stores a column index and value of an intermediate non-zero entry. Figures 2.6(a) and Figure 2.6(b) show the graph states during the Advance and Analyze sweeps of the first iteration. Similarly, Figures 2.6(c) and Figures 2.6(d) correspond to the two sweeps of the second iteration, and Figures 2.6(e) and Figures 2.6(f) to the third.

2.3.4 Summary

The compile time ET approach is applied to dense gradients within AD scalar expressions. For such dense problems, the ET approach eliminates all inefficiencies of the DE OOAD alternative. One fused loop is executed without additional memory transfer or floating point operation requirements. Moreover, the approach eliminates all dynamic memory management needs for temporary intermediate results. The cost of graph construction may be significant for deep expressions with short gradients. Improved compiler optimizations, inlining in particular, may completely eliminate this cost.

The ET approach to building expressions directly carries over to sparse problems. The evaluation process however requires that nodes maintain additional storage and perform branches with conditional logic on column indices. A practical implication of this requirement is that optimizing compilers are far less likely to optimize away intermediate quantities and function call costs. Moreover, owing to the parse graph structure of ET graphs, `axpy` like algorithms are not readily applicable. Finally, unlike the situation with medium to large sized dense arrays, it is more likely than not that sparse arrays have few nonzero entries. Subsequently the relative cost of ET graph construction becomes significant. These factors lead us to develop the final evolution

of ADETL's AD core tailored to sparse problems.

2.4 Sparse Linear Combination Expressions

Despite its unique suitability for dense problems, two inherent aspects of compile time ETs make their use for general sparse problems undesirable. First, in order to accommodate a one-pass sparse algorithm, sparse ET nodes need to maintain state. Subsequently, they can no longer be considered lightweight objects to be optimized away by a compiler. This added cost suggests that there is little to no utility in compile time expression building on the stack. Perhaps we can devise dynamic datastructures that are not costly to build at runtime, and yet which can provide more flexibility. In particular, the second inherent issue is that the ET graph is unstructured and heterogeneous, making it difficult to devise adapted BLAS-like `axpy` algorithms without additional computation. A more flexible runtime alternative could afford a polymorphic environment in which expressions are built, analyzed, and the most appropriate algorithm applied.

It is the pairwise evaluation process of AD scalar expressions that lead to the unstructured, heterogeneous structure of ET graphs. Further insight into the resulting gradient expressions is in order to realize alternate encodings. A basic result from mathematics is that all unary operators including general composition result in gradients of the form $c\mathbf{f}$, where c is a scalar constant, and $\mathbf{f}$ is a gradient vector. Moreover all binary arithmetic operators result in gradients of the form $c_1\mathbf{f_1} + c_2\mathbf{f_2}$. Since these two forms are linear, and their composition preserves linearity, every AD gradient expression can be reduced to a linear combination of gradients $c_1\mathbf{f_1} + \ldots + c_k\mathbf{f_k}$. Specifically, this is achieved by distribution; multiplying scalars through the entire expression. As an example of this, consider the AD scalar expression $W = exp(F * G + H)$. The gradient portion of this expression is,

$$\begin{aligned} w' &= e^{(fg+h)}\,(gf' + fg' + h') \\ &= c_1.f' + c_2.g' + c_3.h', \end{aligned}$$

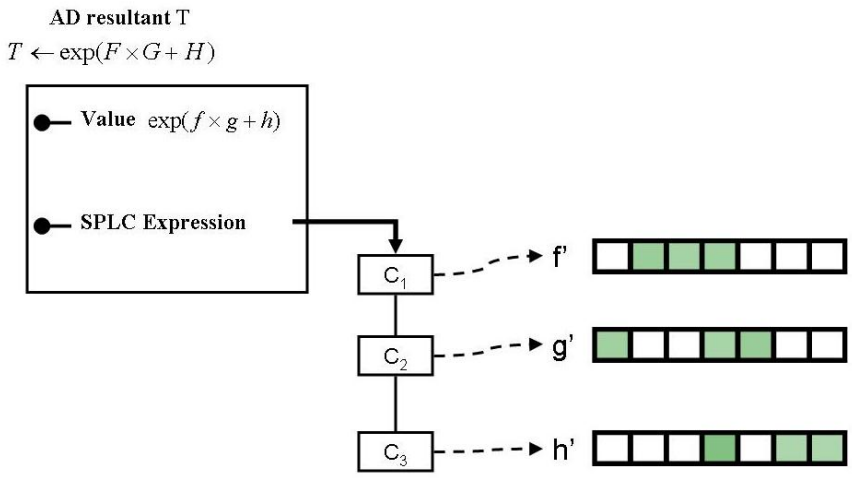


Figure 2.7: An illustration of an SPLC expression attached to the result of an AD scalar expression.

where,

$$c_3 = e^{(fg+h)},$$
$$c_2 = c_3 * f, \text{and},$$
$$c_1 = c_3 * g.$$

The fact that multiplying out gradient expressions always leads to SParse Linear Combinations (SPLC) is a useful property. This is because SPLC expressions always produce balanced binary parse graphs and they can also always be represented by a linear list of coefficients and corresponding gradients. ADETL uses a runtime SPLC expression mechanism as a default for all sparse gradient operations. Returning to the example expression $exp(F * G + H)$, suppose that all three arguments are of type `ADscalar< T >` where `T` is some sparse vector datastructure. As illustrated in Figure 2.7, the result of the statement is an AD scalar whose gradient is a `SPLC_Xpr< T >` datatype. This datatype is no more than a list of coefficient and running pointer pairs that encodes the SPLC expression to be performed upon assignment. Note that unlike ET expressions, the runtime SPLC expression is not embedded within the datatype itself. Instead, SPLC expressions are always of type `SPLC_Xpr< T >`, regardless of

the operations to be performed or the number of arguments involved.

One positive consequence of this is that the overloaded AD scalar operators do not require metaprogramming type traits promotion engines. For example, to overload the multiplication operator for two sparse AD scalar arguments, the header is simply,

```
template< typename T >
ADscalar< SPLC_Xpr< T > > operator* ( const &ADscalar< T > _A,
                                      const &ADscalar< T > _B )
```

On the other, hand a potentially disastrous drawback to the runtime nature of SPLC expressions is construction cost. While each element in the list is small; a floating point number and a memory address, during the course of a single residual evaluation routine, millions of such objects may be created and destroyed. Next we develop the strategies brought by the ADETL in order to avoid the potential pitfalls of computationally expensive expression creation.

2.4.1 Building and destroying SPLC expressions dynamically

An important consideration is the choice of datastructure that is used to store and represent runtime SPLC expressions. Owing to their efficiency of concatenation ADETL uses a singly linked list datastructure for SPLC expressions. Each node in the list represents one coefficient and vector pair within the SPLC. The evaluation of the expression consists of adding up these scalar-vector products. Note that the coefficients are not directly multiplied into the arguments as the expression is built since that would require that each node maintain a new copy of the sparse vector which is obviously very costly.

Each node in the linked list stores a floating point coefficient, a pointer to a sparse vector argument, and a pointer to the next node in the list. The pointer in the terminal node of the list is made to point to the null memory address. There are three fundamental building blocks to any SPLC expression; the multiplication of a scalar and a sparse vector, the multiplication of a scalar and a sub-expression, and the addition of two sub-expression each containing one or more terms. Only in the first situation is it ever necessary to allocate memory dynamically for a new node.

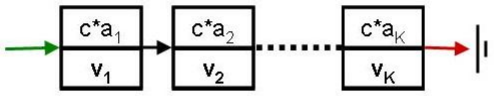
Since the elements of SPLC expressions are allocated dynamically, their lifespan needs to be controlled explicitly, and nodes can be made to persist beyond a statement's scope. For this reason, after evaluation, SPLC expressions need to be destroyed and allocated memory returned to its origin.

Figures 2.8(a) through 2.8(c) illustrate the three overloaded operations which constitute the building blocks of any SPLC. The scalar-expression multiply (Figure 2.8(b)) and expression-expression addition (Figure 2.8(c)) operations require very little overhead compared to a hand crafted evaluation routine. In particular, on top of floating point multiplications which are a part of the evaluation, the multiplication requires the overhead of a number of indirect references to traverse the SPLC list from head to tail. The addition process depicted in Figure 2.8(c), requires the overhead of two pointer operations. On the other hand, the expression node allocation process depicted in Figure 2.8(a) can cause excessive performance degradation depending on what memory allocator is used. The number of such operations in a given expression is equal to the number of sparse gradient arguments. This means that for the average simulator code, the memory allocator can be required to provide nodes several orders of magnitude times the number of equations or indpendent unknowns.

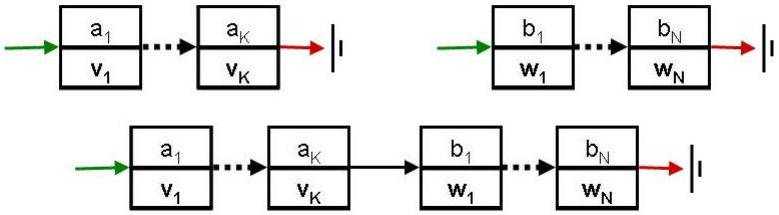
SPLC Memory Allocators

Allocators are abstractions that manage requests for dynamic allocation and deallocation of memory throughout the life of a program. By providing a universal interface, the inner workings of allocators are abstracted, allowing programmers to use them interchangeably. There are numerous generic memory models that range from wrappers around direct system calls to thread-safe shared memory models. As a general rule of thumb, whenever expert knowledge on a particular application's memory needs and access patterns is available, the use of a specialized or custom allocator may substantially improve the performance or memory usage, or both, of the program. The ADETL SPLC expression building process requires the allocation of as many expression nodes as there are arguments in gradient expression. Beyond the scope of a specific statement in which SPLC expressions are built, the small expression node objects are no longer required.

(a) The only overloaded SPLC operator that requires dynamic memory allocation is a constant times a sparse vector, $c.v$



(b) Multiplication of a constant and an splc sub-expression results in a single traversal of the list, multiplying the constant out into the expression weights.



(c) Adding two sub-expressions with one or more entries requires the connection of one sub-expression's tail to the other head.

Figure 2.8: SPLC expressions are represented by a one-directional linked list. There are three fundamental operations the form an SPLC expression.

The ADETL includes its own custom allocator component. While the C++ Standard Template Library provides general-purpose allocators there are many scenarios using the ADETL including SPLC expression building where customized allocators are crucial. These scenarios include improving performance of allocations and deallocations by using memory pools, and encapsulating access to different types of memory, like shared memory or garbage-collected memory. Specifically with regards to SPLC expression construction, many frequent allocations of small amounts of memory are required, and the use of tailor-purpose memory pools is in order. As a consequence, higher level ADETL memory allocators are non-portable; that is, they maintain state, and are therefore not to be used interchangeably with STL allocators for example.

Memory pool models amortize the cost of memory allocation by pre-allocating a cache of memory at the start of a simulation. When some memory is needed, it is handed out by the pool, and if the pool happens to be exhausted, then it is expanded.

Chapter 3

Using and extending the ADETL

The ADETL is a generic extensible library written in the C++ language. At the time of writing, the library requires 6 MB of disk space. Figure 3.1 depicts a conceptual layout of the library and its layered architecture. Software within each layer implements concepts, objects, or algorithms at a certain level of abstractness and complexity. Programming with higher levels of the ADETL requires less computer programming literacy and more Reservoir Simulation domain skills. On the other hand, programming with lower layers requires more awareness of the target hardware platform features and low-level programming detail.

The ADETL is designed to support a combinatorial level of flexibility and extensibility. Within each layer, objects may have one or more template parameters. By specifying a combination of non-default choices for any or all of these parameters, the user can toggle the make-up of the code that is compiled. Extending the library can be achieved by implementing additional variations of the various parameter options so long as the additions satisfy the parameter's required concept criteria. The ADETL uses this form of compile-time polymorphism to allow users of higher levels to produce platform specific, performance-tuned code with little or no knowledge of hardware-specific coding optimizations. Rather, the user can simply select what is best and does not have to worry about how it is implemented or how it evolves.

Figure 3.1 illustrates the multi-layered structure of the library. We introduce the usage semantics and the concept parameter option spaces within each layer proceeding

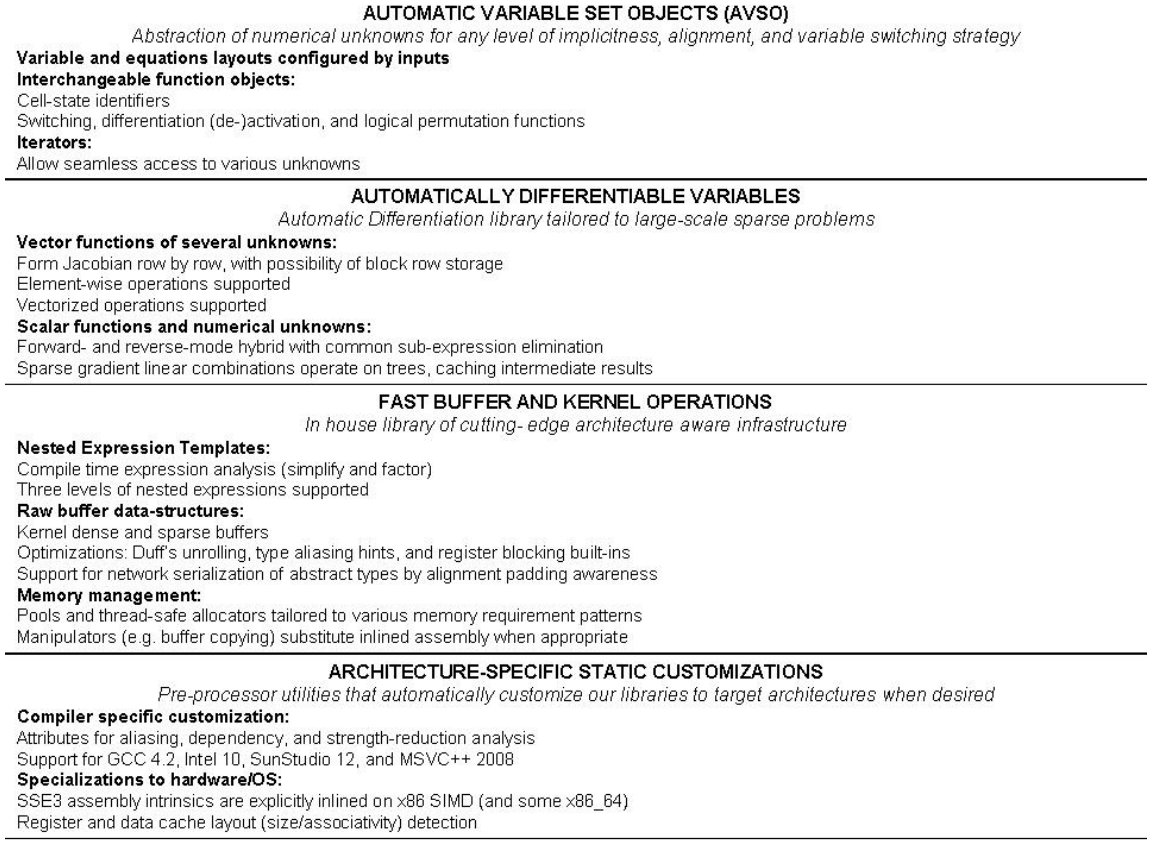


Figure 3.1: Overview of the four layer architecture of the ADETL.

in a top-down order.

3.1 Automatic Variable Set Objects AVSO.

In reservoir simulator design, two important considerations are the choice of independent variables and the way that they are ordered. There is interest in the ability to rapidly implement simulators using different variable sets and orderings and to use them interchangeably. Moreover, different physical models typically involve different variables. Subsequently there is also interest in utilities that facilitate the incorporation of new variables and new models. Without programming abstractions to generically address all aspects of simulation variable sets, it is unlikely to achieve the kind of design flexibility and extendability that is saught today. Moreover, without AD, the process of achieving a suitably high-level abstraction is also intractable; the notions of ordering and their impact on sparsity are inherently very much coupled. This layer of the ADETL builds on top of its sparsity detecting AD capability in order to provide a total abstraction of the notion of independent variables. The Automatic Variable Set Object (AVSO) layer of the ADETL provides programming semantics and functionality that allow users to configure, declare, operate, and manipulate numerical unknowns without the need for any low-level programming and with complete plug-and-play flexibility.

To appreciate some of the challenging design problems solved by the AVSO layer, consider the task of adding thermal effects into an object-oriented compositional simulator such as GPRS [13]. The first step could be to allocate sufficient working space to hold temperature and the other new thermal intermediate variables as well as their derivatives. In GPRS, this can be done by modifying the `DataPool` class. Immediately, the first bottleneck arises; the mathematical ordering of variables needs to be explicitly mapped on to the new layout of memory in the `DataPool`, and all indirect addressing code needs to be altered in order to accommodate. These changes can be extensive. Moreover, since the changes proliferate throughout the gut of the discretization component with a fine level of granularity, there is ample potential for subtle computer programming logic errors. After space is allocated for the new

variables and all indexing and indirect referencing issues are resolved, the next change is to modify the memory layout of the residual vector and the Jacobian matrix to accommodate the temperature equation. This introduces further indirect addressing issues which need to be addressed. The situation is further exacerbated when the variable switching logic needs to be modified accoridingly in order to evaluate a consistent residual vector and Jacobian matrix. Without AD, it is necessary for the programmer to explicitly write out partial derivatives and insert them within the appropriate Jacobian entries. Once again, it becomes unavoidable to have to worry about granular context specific modifications, and to sift out any programming logic errors.

The kind of flexibility that the ADETL delivers extends beyond making it almost effortless to introduce new variables, equations, or alignments. Using a declarative syntax, the user can also be oblivious of any indirect mapping changes and the underlying memory layout details. Using the AVSO layer, any variable switching strategy can used simply by declaring it as a simulation parameter. No indexing changes to the size or structure of the Jacobian need to be implemented. They are taken care of automatically behind the scenes.

The main construct in the AVSO collection is the template,
`VariableSet<` **ADCollection, VariableOrdering, VariableActivation** `>`,
which requires concrete types for three concepts; **ADCollection** is the concept for an underlying AD data structure and including its memory layout, **VariableOrdering** is a record of the mathematical ordering of unknowns in a given context, and **VariableActivation** is a specification of when and which variables should be active (independent) or inactive (dependent). To instantiate a `VariableSet` object, a concrete type needs to be selected for each of these concepts. There are a number of different concrete types that are currently available for use. Moreover, users who need new functionality can simply extend the library by implementing the desired concrete type so long as the implementation satisfies the set concept criteria. The concept criteria ensure plug-and-play integration and compatibility with other concepts. For example, one can choose independently choose to use a distributed memory model, an ordering such as the one in GPRS, and a custom variable switching schedule,

without having to worry about the dependencies across concepts. Before presenting usage examples, we discuss the three concepts, their criteria for extensions, and the currently existing options.

3.1.1 ADCollection concept and types.

The motivation behind this concept is to abstract the physical memory layout, its management, and its indirect addressing from the mathematical ordering of variables and equation alignment. This design choice is made recognizing today's rapidly changing computer hardware scene. Memory management and access are two critical performance concerns, and optimal implementation techniques vary dramatically across architectures. For example, memory optimization for serial or multi-core cache-oriented machines is not suitable for massively parallel distributed architectures.

Concept requirements. Suitable types must, from the perspective of an interface, consist of a collection of contiguously enumerable and accessible AD scalar types. This requirement does not preclude types that store variables in non-coherent buffers, so long as they provide an interface that makes it look like they do. Such types are implemented in the second layer of ADETL under collections. The concept requirements for the scalar entry data types are that they support ADETL value and gradient access interfaces. No other criteria on memory allocation or coherency are required, and such things are all left to the will of the user.

Currently available examples. The second layer of the ADETL implements policy-based collections with the `ADvector` template. The default is to use dynamic memory allocators for coherent buffers ordered in a stride-blocking fashion. This object can be declared by leaving its template parameters unspecified; For example,

```
ADvector<> X_unknowns( );
```

declares a dynamically allocated array of AD scalars, using coherent memory, and a stride oriented ordering.

3.1.2 VariableOrdering concept and types.

Formally, we define a set of independent variables as,

- a contiguously enumerable collection of scalar unknowns, and,
- a collection with the square identity matrix as its Jacobian.

In Reservoir Simulation, there are numerous options for the choice of variables, as well as for how they are ordered. The ordering of the scalars determines the column permutation structure of the Jacobian matrix of any multivariate function including the residual. The specific choice of variables and their ordering heavily influences the computational efficiency of the linear solution process. Moreover, many formulations involve dynamic variable switching, where the Jacobian of the identity operation changes size. This further adds to the complexity of implementing flexible simulation codes. With the introduction of AD and an ability to specify the differentiation seed (the partial derivative of an unknown with respect to itself), it is paramount for the ADETL to provide high-level facilities that automatically manage variable sets behind the scenes. The **VariableOrdering** concept abstracts the notion of automatic variable ordering and enforces a uniform indirect addressing map, thereby eliminating the burden of all low-level programming semantics. The **VariableOrdering** concept sets a fixed standard for the automatic activation and enumeration of the derivative seed of any declared dynamic variable set. One can set-up and instantiate any variable set with any variable partitioning or ordering. The activation and deactivation of variables for differentiation are handled automatically.

Concept requirements. As discussed, ADETL expresses any Reservoir Simulation choice of variable and equation alignment using an unstructured three-dimensional indexing system. The first index enumerates levels; A level is defined as a contiguously enumerated collection of tiles. Each tile is a collection of one or more scalar variables. The **VariableOrdering** concept requirements are twofold. The first requirement is that objects provide an interface for the client to specify any three-dimensional unstructured ordering. The second requirement is that objects provide an interface that responds to queries on structure. These requirements leave the **VariableOrdering**

concept rather extensible; in one implementation, access can be organized to visit a space-filling curve, and in the next, it might be organized by visiting active variables only.

Currently available examples. The ADETL classes to instantiate in order to specify and record a specific ordering is `MultiLevelRecord`. Additionally, in order to facilitate working with single level variable sets, the class `LevelRecord` may also be used, eliminating the overhead of accessing levels in a single level setting.

Example 1. *Black oil variables A typical choice and ordering of numerical unknowns for a two-phase oil and water model is to enumerate primary variables first, followed by well-pressure variables listed for each segment, one well at a time. The primary variables are listed one grid block at a time, and for each block, the typical ordering is pressure followed by saturation. For example for a problem with N_b grid blocks, and two wells with N_s segments each, this ordering becomes,*

$$P_1, S_1, \ldots, P_{N_b}, S_{N_b}, Pw_1^1, \ldots, Pw_{N_s}^1, Pw_1^2, \ldots, Pw_{N_s}^2. \tag{3.1.1}$$

To model this ordering using the three-dimensional indexing concept, we need to specify:

1. *That there are three levels; Primary variables, well pressures for the first well, and those for the second.*

2. *The number of tiles in each level, which for the concrete example above is N_b for the first level, N_s for the second, and N_s for the third as well.*

3. *The number of variables in each tile in each level, which for this example is two, one, and one.*

The corresponding `MultiLevelRecord` *object can be declared by,*

```
MultiLevelRecord  BlackOilSetup( 3, [ Nb, Ns, Ns ], [2,1,1,] );
```

3.1.3 VariableActivation concept and types.

Variable switching is the process of selecting the appropriate minimal set of unknowns such that a physical model is consistent and uniquely determinate. In Reservoir Simulation, variable switching is used in various formulations of multiphase flow models in which phases may appear or disappear. Moreover, it often occurs that the independent variable sets for different states have a differing dimensionality. Subsequently, the dimensionality of the residual vector and the Jacobian matrix can change also. A similar process is observed in dynamically implicit discretizations such the Adaptive Implicit Method (AIM). With approaches such as AIM, the independent variable set in a block within a level may change to reflect different implicitness treatments of the individual variables. When a variable is required to be implicit, it becomes active from the AD perspective, and *vice versa*. The **VariableActivation** concept abstracts this process of enacting and querying the activation of individual variables independently tile by tile, and level by level. In so doing, the concept provides concrete implementations that automatically take care of rearranging column structure and enumeration of variables.

Concept requirements. According to the **VariableOrdering** unstructured three-dimensional structure, each tile in a given level lists a contiguous set of unknowns. The concept requires that this be a list of the union of all sets of possibly active unknowns across all states without duplication. The phase and/or implicitness states need to be associated with unique tags which may be of any data type. The last requirement is that implementations support a table that indicates which unknowns are active for each state tag.

Currently available examples. The only currently available object is of the `2DStatusTable` class. This implementation provides an interface to query structural information regarding a status tag associated with any level. This structural information is used by the `VariableSet` object to automatically effect status changes. The information consists of a boolean list of active variables for each tag, as well as the number of active and inactive unknowns. Inversely, the object may also be queried for a list of the state tags for which a specific variable is active.

Example 2. *Three-phase compositional Natural Variables Consider a three-phase four-component isothermal problem that is to be modeled using the Natural Variables. In the formulation, the primary variables in each grid-block depend on the phase-state of the block. Supposing that this setup is described by Table 3.1 below.*

State	P	So	Sg	X_g^{C1}
Oil, Water	x	x		
Oil, Gas	x	x		x
Three Phase	x	x	x	x

Table 3.1: A hypothetical set of primary variables for a three-phase problem. A check (x) marks an active independent unknown.

A `2DStatusTable` *object supporting this setup can be declared using code Listing 3. The last two lines of the listing illustrate some query functionality.*

Listing 3 Declaring a Natural Variable activation setup using the ADETL.

```
int  NumberOfVariables  = 4;
int  NumberOfStates     = 3;
int  StateTags[]        = { 1, 2, 3 }; \\ OW, OG, OWG
bool StateActivations[] = { 1, 1, 0, 0,
                            1, 1, 0, 1,
                            1, 1, 1, 1 };
2DStatusTable<int>  NaturalVariables( NumberOfVariables,
                                      NumberOfStates,
                                      StateTags,
                                      StateActivations );

\\ Query number of active variables for OG state
NaturalVariables.NumberActive( 2 ); \\ returns 3

\\ Query whether Sg is active for the OG state
NaturalVariables.IsActive( 2, 3 );  \\ returns false
```

3.1.4 A complete AVSO `VariableSet` usage example.

In this example, we walk through the process of developing a simulation residual and Jacobian code to model a three phase compositional process. The focus is on first showing how to use the AVSO layer, and then to show how one can change the formulation.

For the sake of clarity, we restrict attention to the three-phase, four component model described in Example 2. We assume there is one injection well operating under a rate constraint. We aim to use the Natural Variables initially, and we choose the primary set as P, So, Sg, and X_g^{C1}. The secondary set consists of the variables X_o^{C2}, Sw, X_g^{C2}, X_o^{C1}, and the well, which is completed in one grid-block, is associated with a single well-pressure unknown, Pw.

The first step is to declare an AD `VariableSet`. To do this, we need to select concrete types for three concepts. For the **ADCollection** concept, we choose the ADETL default which is the coherent dynamically allocated AD array, `ADvector< >`. Next, we specify the logical ordering of unknowns using the following declaration for a `MultiLevelRecord`,

```
MultiLevelRecord  NaturalOrderingSetup( 3, [ Nb, Nb, 1 ], [4,4,1,] );
```

The declaration parameters in this statement specify that there are three levels. In both the primary and secondary levels, there are Nb tiles, and in the well completion level, there is only one tile. The third parameter is a list of the number of variables to be stored in the tiles of each level.

Next, we need to specify a formulation strategy for each level. For the primary variables, we use the Natural Variables strategy described in Listing 3. The secondary variables in this example require no variable switching and are always treated implicitly, and so it is appropriate to use the pre-built AVSO element `NoSwitchingPolicy` as the concrete type. The activation of the well variable depends on the well-constraint, and may be switched from a variable with fixed flow rate, to a constant with a fixed BHP. To model this, we declare a `2DStatusTable` using Listing 4.

Listing 4 Declaring an well-constraint variable activation setup.

```
int  NumberOfVariables  = 1;
int  NumberOfStates     = 2;
int  StateTags[]        = { 1, 2 }; \\ RATE, BHP
bool StateActivations[] = { 1,   0 };
2DStatusTable<int>  WellVariables( NumberOfVariables, NumberOfStates,
                                   StateTags, StateActivations );
```

Listing 5 provides an implementation that declares an automatic variable set. The Listing also illustrates several functionalities such as variable access and modification and cell status changes.

Listing 5 Declaring an independent set.

```
VariableSet< ADvector<>,
             MultiLevelRecord,
             2DStatusTable<int> > X( NaturalOrderingSetup,
                                     [NaturalVariables,
                                      NoSwitchingPolicy,
                                      WellVariables ] );

int block_number = 100;
int primary   = 1;
int secondary = 2;
int pressure  = 1;

// can access and modify variables using natural semantics
X[ primary ][ block_number ][ pressure ] = 100.0;

// automatically affect appropriate AD variable activations
X[ secondary ][ block_number ].set_status( 2 );
```

Perhaps the most important functionality of the `VariableSet` construct is its interface for setting the variable activation state of any given simulation grid block and any variable level. Changes to the status of a cell can have repercussions on all subsequent entries in the independent set. To see this, consider the scenario illustrated in Figure 3.2. In this scenario, the water phase in the first cell of the problem

disappears. To affect this change, it is necessary to set the variable state in the first tile of the first level. Since now there is one less independent variable, the ordering of unknowns changes in every following entry. Figure 3.3 illustrates this process in terms of the Jacobian matrix of the unknowns. As introduced, AD augments all data with its derivative, and so, variable set changes need to be reflected in the seed Jacobian matrix of the unknowns. The actual implementation for achieving this is left to the implementer of the `VariableSet` concept. In the current implementation, state changes do not involve any dynamic memory management needs. The implementation however does require one indirect addressing and assignment for every effected entry. For this reason, it may be necessary for computational efficiency to devise other implementations that scale in a sub-linear manner. This will be particularly important for large-scale problems on cache limited machines where the number of register loads and stores are critical.

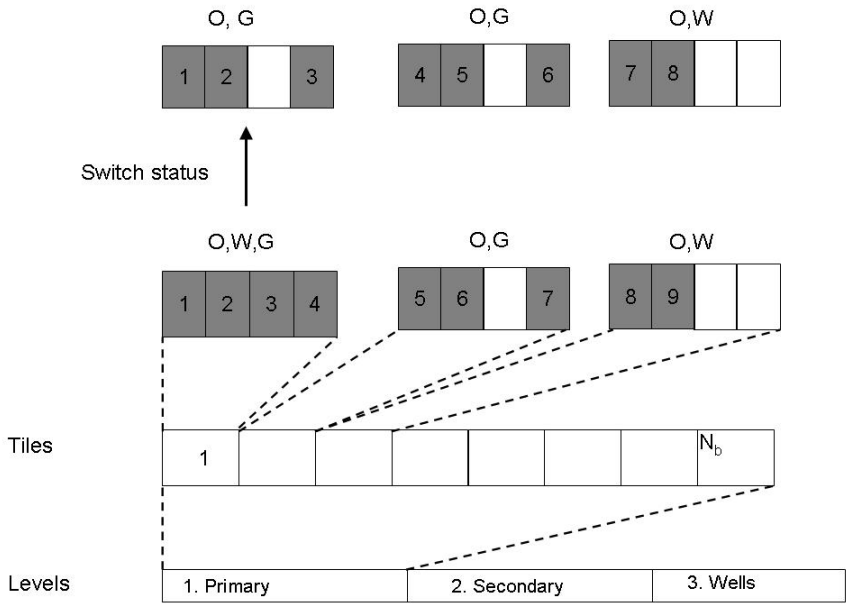


Figure 3.2: A conceptual view of the way ADETL treats variable switching.

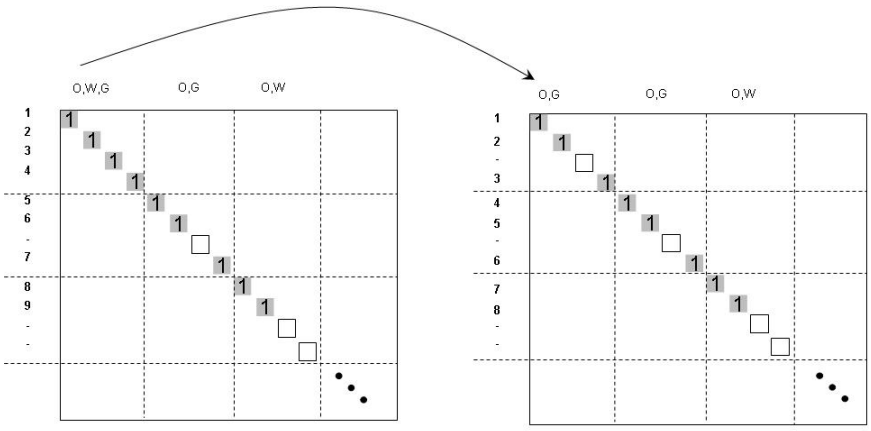


Figure 3.3: ADETL treats variable switching by modifying the sparse Jacobian matrix of the unknowns.

3.2 Automatically differentiable variable data types

The second layer of the ADETL provides various Automatically Differentiable data-types which can be used to evaluate and store nonlinear terms and equations. Mathematically, any generally nonlinear mapping is

The data-types are organized into two concepts. The first is the concept of an AD scalar variable and the second is an AD collection. This choice of distinctly separating collections from scalars is common to generic programming philosophies primarily because it allows for an abstraction of buffer memory management from scalar algebra and in this case, from AD functionality as well. A scalar can be used to represent anything from a real number to the result of a multivariate map with a target in the real numbers. Collections are sets of scalars and a collection can be used to represent vector quantities as well as sets of nonlinear multivariate equations or vector functions.

3.2.1 The ADCollection concept.

In Generic Programming, a collection is defined as a data-structure of one or more scalars. Each scalar entry in a collection can only be referred to *via* a reference to the collection itself. Examples of collections are arrays, queues, and associative maps such as dictionaries. The C++ Standard Template Library (STL) provides implementations of several collection containers, giving the developer access to fast and efficient collections that are type safe and easy to use. Moreover, the STL collections are generic implementations; They abstract both memory management and indirect addressing, and separate these from the specifics of the scalar element data type. Moreover, in the case of associative maps, they also abstract the associativity relation. For the purpose of constructing discretization codes, the most important collections are sequential random access containers, and more specifically arrays and vectors. The STL design choice for implementing sequential collections is to use templates with two parameters. For example, the popular STL vector container, `std::vector< T, A >`, requires two template parameters. The first template parameter, **T**, is the concrete type of scalar element to be stored, and the second parameter, **A**, is the type of memory allocator to be used.

The collections offered by the STL have withstood the test of time, and have proven their worth through pervasive use by C++ programmers. Nevertheless, for the purpose of AD collections, the STL containers leave two properties to be desired. Firstly, while the dimension of an AD vector dictates the row dimension of its Jacobian matrix, the number of unknowns or independent variables in the differentiation context dictates the column dimension of the Jacobian. In order to allow ADETL users to program with the more natural two-dimensional semantics of Linear Algebra, `ADCollection`s must implement AD specific behavior pertaining to the collection as a whole. In a sense, this amounts to designing collections so that they can act and appear as both the residual vector and the Jacobian matrix depending on the user's interests. The second feature, is an efficiency issue. The STL collections are generic, and they apply the most general, yet least-efficient variable instantiation and initialization mechanisms. The TR1 of the C++ language standards committee has long recognized this aspect of STL collection as a potential drawback to using them in

scientific computing. ADETL collections address this by introducing an additional template parameter to allow the user to fine tune memory manipulator choices.

Example 3. *Using STL collections.*
To appreciate the importance of the `ADCollection`*s, consider the alternative of using a STL vector to store the AD scalar types representing the discrete residual and its Jacobian. We walk through the process of declaring such a structure, performing a discretization operation, and finally, accessing information on the Jacobian matrix. Listing 6 shows the semantics involved in doing this. The first statement declares the data-structure, allocates memory for it, and initializes all entries. In the declaration, the allocator template parameter,* ***A****, is not explicitly specified, signaling the compiler to use the STL default memory allocation policy. Moreover, the programmer has no control over how the variables are initialized. After performing a standard connection-list pressure differencing, the programmer needs to involve herself with low-level, vector related semantics, in order to determine the column dimension of the Jacobian matrix. Using ADETL collections, we provide interfaces such as the last statement in the listing.*

Figure 3.4 shows the taxonomy chart of `ADCollection`s and the currently implemented types. The collections are divided into two broad categories. The first, is collections that are to be allocated on the stack memory, requiring a fixed length that is specified at compile-time. The second category are collections that are allocated dynamically on the heap. Dynamic collections are subdivided into those that are fixed in length albeit at runtime, and those that may be re-sized.

The stack-based container `ADStackArray< int LENGTH,` **ElemType** `>` is a wrapper around a built-in C++ array. The length of the array is specified as the first template parameter, and the concrete ADETL AD scalar element type as the second parameter. Stack arrays are designed for use when small arrays are required.

The `ADarray<` **ElemType, Allocator, Manipulator** `>` template class is a dynamically allocated, fixed size array. The length of the array may be specified at runtime or compile-time, but once it is specified, it cannot be changed. The implications of this on AD is that the row dimension of the Jacobian is guaranteed to be fixed,

Listing 6 Using ADETL with STL collection containers requires low-level awkward semantics.

```
// Declare an STL vector of 100 AD variables
// The STL will run 100 constructor calls even though
//      ADscalars need not be initialized yet
std::vector< ADETL::ADscalar< > > discrete_problem(100);

...

// Perform some computations
for ( cell_id = 0; cell_id < 100; ++cell_id )
{
   int          left_id = connection_list[ cell_id ].left;
   int         right_id = connection_list[ cell_id ].right;
   ADETL::ADscalar< > K = connection_list[ cell_id ].transmissibility;
   discrete_problem[ cell_id ] = K * ( pressure[ right_id ] -
                                       pressure[ left_id ] );
}

// Awkward low-level semantics to query the
// column dimension of the Jacobian matrix
int jacobian_col_dim = -1;
for (int i=0; i < 100; ++i )
{
    int row_col_d = discrete_problem[i].gradient().back().col_index();
    if ( row_col_d > jacobian_col_dim )
        jacobian_column_dim = row_col_d;
}

// We would prefer semantics that are natural
// when talking about Jacobians
int jacobian_col_dim = discrete_problem.column_dim();
```

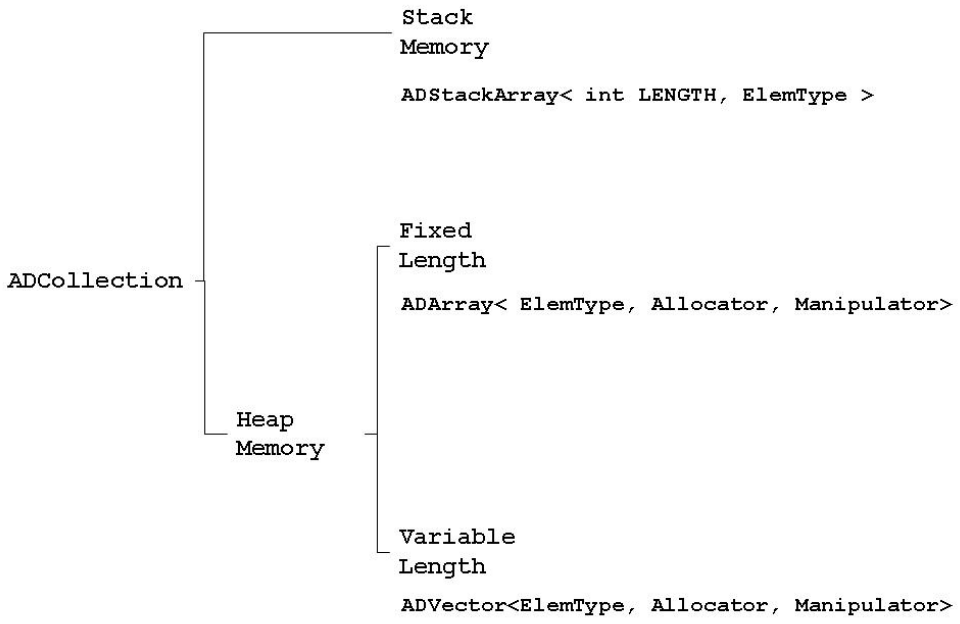


Figure 3.4: Taxonomy chart and examples of collection datastructures that are available through the ADETL.

and the implementation of this container can take advantage of this to attain better efficiency.

The most flexible collection data structure is the `ADvector< ` **ElemType, Allocator, Manipulator** `>` template class. It is dynamically allocated and re-sizable at runtime.

Chapter 4

ADETL Simulators and computational examples

The inception of the ADETL spurred an initiative to develop a general-purpose large-scale reservoir simulator; ADGPRS. The simulator is to use the ADETL as its exclusive source of datastructures, active and otherwise. Empowered with the ADETL as its building block for all formulation, discretization, and linearization models, the simulator promises to deliver new kinds of flexibility and productivity to the fingertips of the simulation research community. With the introduction of the ADETL, ADGPRS departs from the model of abstraction as a means to divide and conquer, and arrives at the model of abstraction from within. The ADETL provides a midlayer generic library that implements a general purpose variable set model. ADGPRS uses the ADETL to abstract completely the notions of formulation and variable switching strategies. Moreover, all sparse analytical Jacobian matrices are evaluated automatically behind the scenes. Object Oriented runtime polymorphism is factored away from the inner trenches of formulation and linearization routines in favor of ADETL's more efficient and symantically natural generic alternatives.

Researchers can work with ADGPRS using domain specific natural language semantics to play with new physical models, formulations, and discretization techniques all the while having the benefits of a large scale robust simulator. Moreover since the ADETL provides all datastructures for ADGPRS, the simulator is instantaneously

generic with regards to hardware architecture specializations. Primitive ADETL constructs provide iterators and memory models which can be switched out at will for various alternate implementations without needing to alter any higher level code. Moreover ADGPRS is truly generic with respect to formulation. Variable switching, ordering, and alignments can be specified in a parameter file. There is no need for any manual analysis and implementation of permutation or transformation routines. Finally, the ADGPRS is well on its way to providing the first reported simulator suite that operates under a one-code, many executable system.

4.1 So, what is the overhead of this abstraction?

In general, it is difficult to accurately forecast the anticipated computational overhead of using the ADETL. This is simply because an objective answer is bound to be application and context specific. Moreover, across a wide spectrum of situations or applications, the overhead can vary dramatically. For these reasons, we answer the question at hand by providing an empirical characterization of the overhead specifically for the use of the ADETL to develop the large scale compositional simulator ADGPRS.

We assess the computational overhead of using the ADETL by comparing empirical runtime data generated by ADGPRS and GPRS for a suite of benchmark problems. The original GPRS code [14, 13] is an Object Oriented simulator written in the C++ language. Its formulation and linearization core is based on hand-coded derivatives and sparse Jacobian matrix construction routines. The variable switching, implicitness level, and alternate formulation capabilities are all based on a hand-coded system that uses a base model along with permutation, combination, and elimination algebraic routines to arrive at the target alternate systems. Since its first release in 1999, the GPRS has been expanded and maintained without fundamental changes to the discretization skeletal infrastructure. The GPRS project led to numerous well received advances in the areas of advanced process modelling such as chemically reactive compositional flows, smart well and facilities modelling, and in optmization and control.

The objectivity of such comparisons is subject to numerous factors. Amongst the more significant are the following:

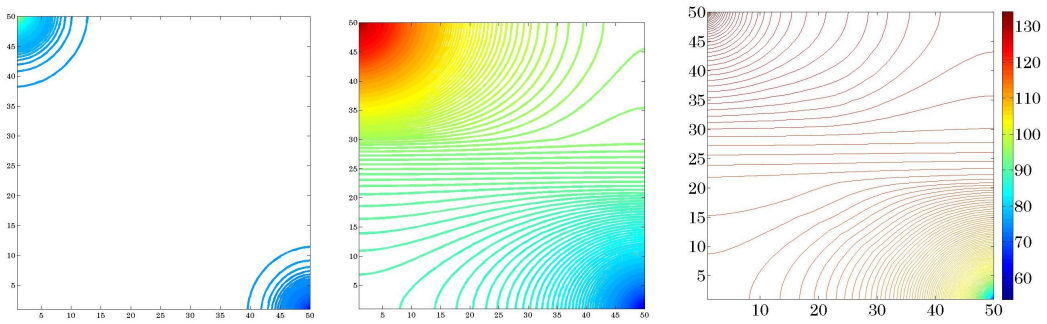
- ADGPRS is developed by a team that is familiar with the internal workings of GPRS. While the developer of the ADETL provides input in the form of advice, there was no direct involvement in decisions such as the choice of datastructures or memory management. Subsequently, it is fair to assume that anyone without expert knowledge of the ADETL and with a fair amount of domain expertise can expect to see similar results.

- ADGPRS uses one AD datatype. That is AD-types with block sparse gradients and SPLC expressions are employed regardless of the underlying derivative structure or call frequency. As will be more clear from the test suite results, this implies that with a more judicial use of the ADETL, the performance of ADGPRS can be improved relative to GPRS.

- The templates facility is used to a large degree in order to allow routine calls that involve no derivatives to be executed using built-in floating point datatypes rather than ADETL types. This means that we are not highlighting the overhead of using the ADETL for computations where no derivatives are necessary.

- In the examples presented, only 35% of the ADETL was used in the execution chain. Subsequently, this implies that the comparison results in these examples do not characterize the entire library. Rather, they provide empirical results for the use of certain portions of the library.

- The same compiler and build environment were used to obtain executables of either simulator. The empirical data used in the comparisons were obtained on a dedicated four core workstation.



(a) 0.1 days (b) 260 days (c) 980 days

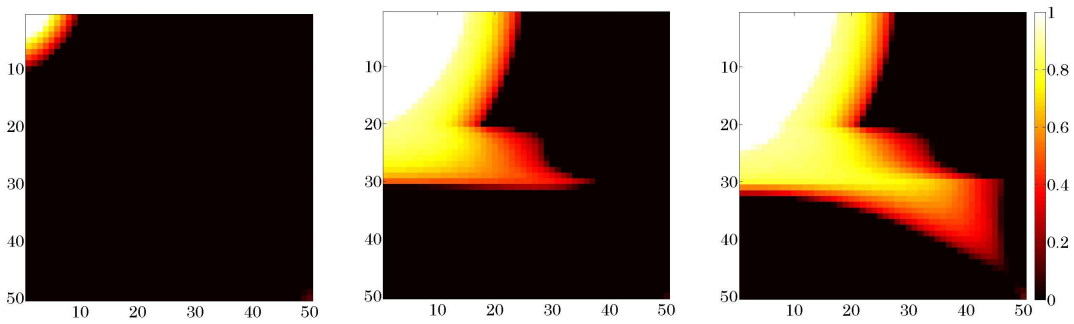
Figure 4.1: Pressure (bar) contour plots for three time snapshots during the course of a simulation. The reservoir is initially at 75 bar. An injection well is located at the top left corner and is operated at a BHP of 140 bar. A production well located at the lower right corner is operated at 30 bar.

The test-suite consists of four simulation examples. All of the examples involve compositional models of multiphase flow that are driven by wells. The examples differ in terms of the structure and level of heterogeneity of the field, the problem dimensionality, and the number of chemical components and phases involved.

4.1.1 Example 1: Two dimensional, nine component model

The first benchmark problem involves a horizontal two-dimensional square field that is homogeneous except for the presence of a hundred-fold permeability anisotropy strip running along the middle. The field is discretized using a 50 by 50 Cartesian mesh. The model is of nine chemical components which may partition into two phases; oil and gas. At one corner of the field, an injection well is operated under a Bottom Hole Pressure (BHP) control introducing a gas mixture of Methane and Carbon Dioxide into the reservoir. At the diagonally opposite corner of the field, a production well is operated under a fixed BHP control as well.

Figure 4.1 shows a sequence of three contour plot time snapshots of the pressure field. The evolution of the pressure field shows the effects of the considerable compressibility within the system which introduces a transient stage of flow development. Moreover, due to both, the heterogeneity and the differing phase mobilities,



(a) 30 days (b) 500 days (c) 750 days

Figure 4.2: Gas saturation time snapshots of a simulation of a reservoir that is initially oil saturated. An injection well located at the top left corner introduces a gaseous mixture of Methane and Carbon Dioxide while a production well located at the lower right corner produces a mixture of gas and oil.

the pressure field is not radially symmetric.

Figure 4.2 shows a sequence of three time snapshots of the gas saturation. Initially, the reservoir is pressurized and the fluids within it are within the oil phase. As gas is injected, the production well draws down on the reservoir thereby lowering the pressure within the vicinity of the production well. This local de-pressurization causes some phase change to occur at the production well liberating some gas. This example includes phase appearance due to both injection effects and thermodynamic effects. Both situations result in variable switching within the affected cells.

Table 4.1 presents the timestepping and solver statistics over the course of the simulations using GPRS and ADGPRS. While the two simulators are as similar as possible as far as modelling and alogrithmic content, there are some differences that lead to the count discrepancies. For instance, various heuristics are used within the nonlinear solution loops of both simulators, and at the time of writing, these differences may lead to a alight variation in the number of nonlinear iterations to convergence.

Table 4.2 presents the timing results obtained. In addition to total simulation time, the table lists the timings of three parts of the simulation process; the time spent on computing stencil or other spatial discretization operations, time spent in computing physical and thermodynamic properties, and time spent on linear solution.

	GPRS	ADGPRS
Time Steps	61	61
Newton Iterations	271	262
Linear Iterations	1021	1012

Table 4.1: Time-stepping and solver statistics for a simulation using GPRS and ADGPRS.

Factoring in the number of iterations taken, the overhead of the ADETL in this case is clearly negligible.

	GPRS	ADGPRS	
Discretization (sec)	20	15	75%
Properties (sec)	87	67	77%
Linear Solver (sec)	31	36	116%
Simulation Total (sec)	139	123	88.5%

Table 4.2: Computational time

4.1.2 Example 2: A five-spot pattern in a two-dimensional heterogeneous domain

The second benchmark problem involves a horizontal two-dimensional rectangular field with structured heterogeneity. The permeability, and the corollated porosity, were taken from the tenth layer of the SPE 10 Comparative Study Model [16]. Figure 4.3 shows a plot of the logarithm of permeability in the field. The field is discretized using a 60 by 220 Cartesian mesh. The model involves four chemical components which may partition into the oil and gas phases. A five spot pattern of wells is

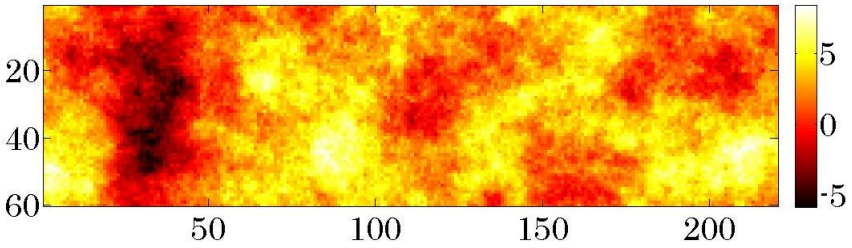


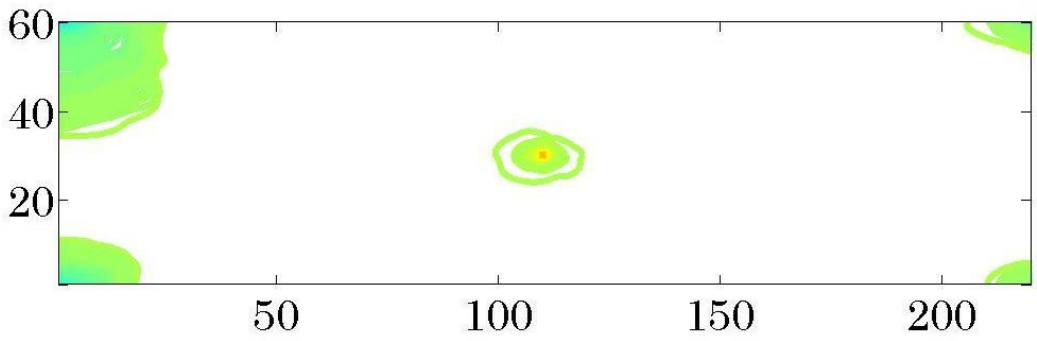
Figure 4.3: A plot of the logarithm of permeability taken from the tenth layer of the SPE 10 model [16]. The porosity used in this example is correlated with permeability.

formed by operating four production wells on each of the corners, and one injection well in the center of the field. The injection well is operated under a Bottom Hole Pressure (BHP) control and introduces a gas mixture of Methane and Carbon Dioxide into the reservoir. The production wells are also operated under a fixed BHP control.

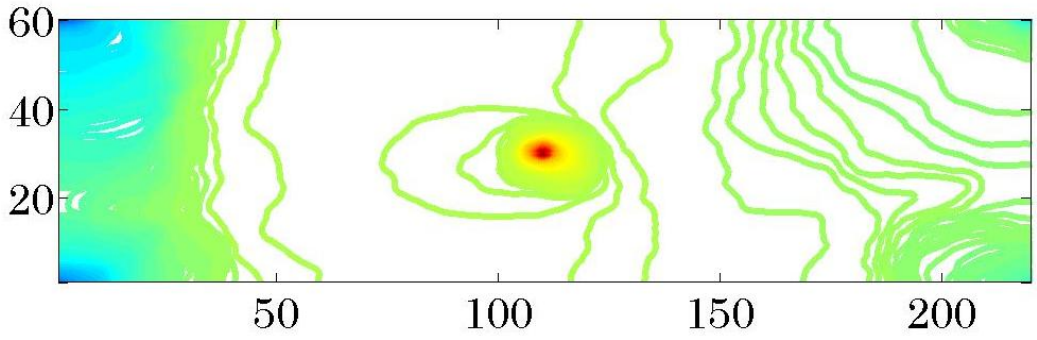
Similar to the situation in the previous benchmark example, the model involves significant compressibility. Figure 4.4 shows a sequence of three pressure field contour plots at three times during the course of a simulation. The evolution of the pressure field shows transient effects, and to a larger degree than in the previous case, the field heterogeneity forces an asymmetric pressure field.

Figure 4.5 shows a sequence of three time snapshots of the gas saturation in the field. As was the case in the previous example, the reservoir is initially pressurized such that the original fluid within it is in the oil phase exclusively. As gas is injected and the production wells come online, a pressure gradient develops. Because of the structure of the permeability in this example, the two production wells on the left side of the domain draw down a steeper gradient. This introduces a rather substantial phase change from oil to gas due to thermodynamic stability. Forced gas phase introduction occurs due to direct injection, and in this case, the injected gas produces an asymmetric front.

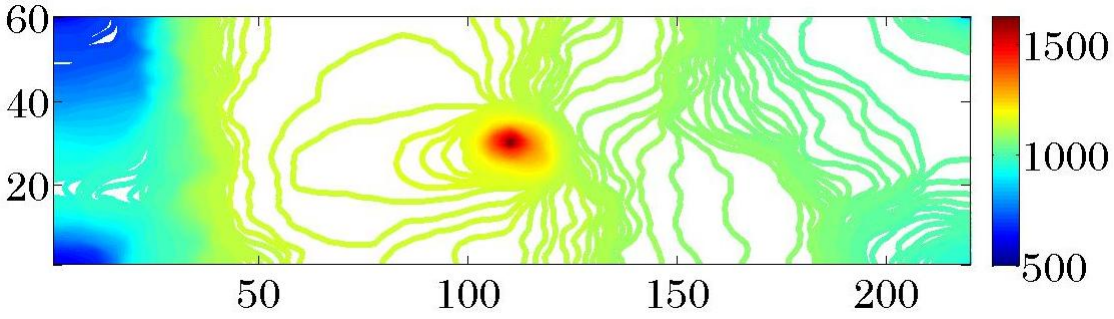
Table 4.3 presents the timestepping and solver statistics over the course of the simulations using GPRS and ADGPRS. While the two simulators are as similar as possible as far as modelling and alogrithmic content, there are some differences that lead to the count discrepancies. For instance, various heuristics are used within



(a) 0.03 days

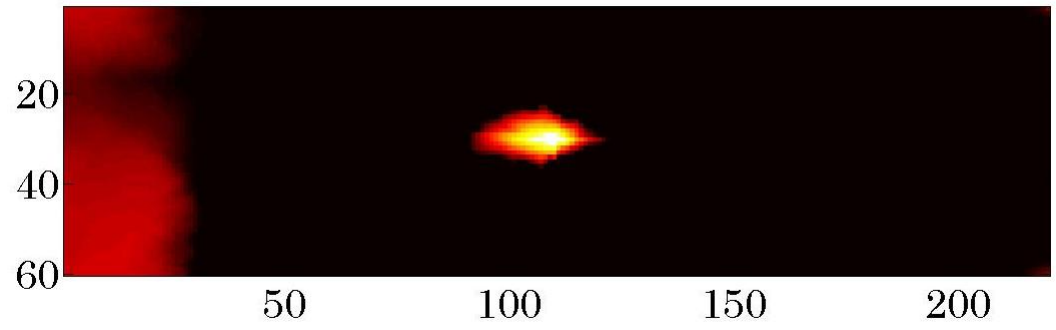


(b) 1.25 days

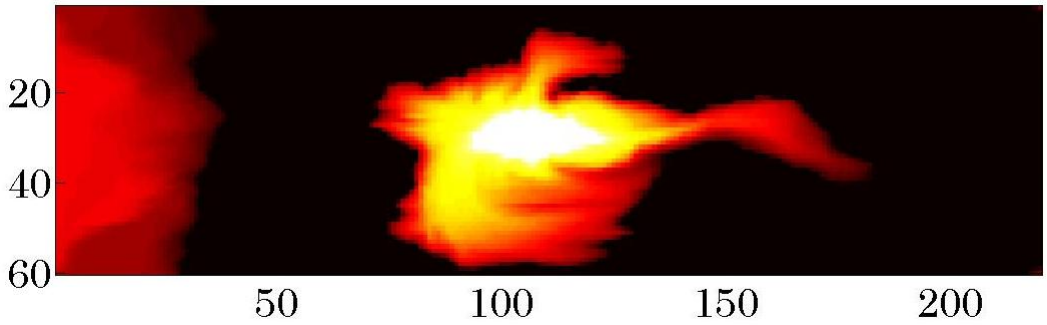


(c) 7.5 days

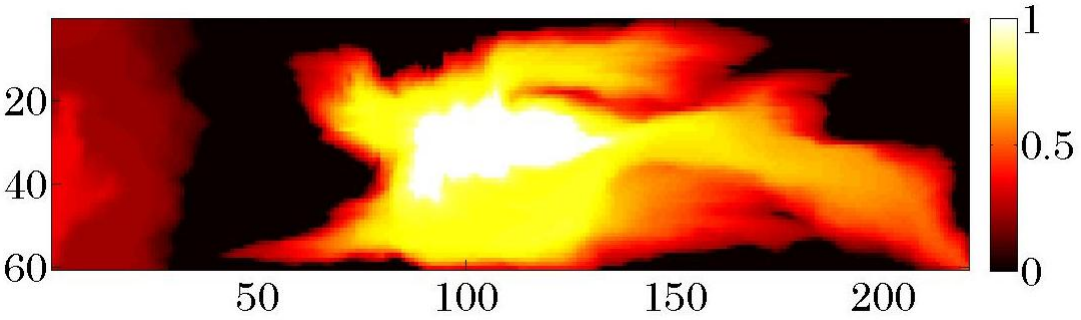
Figure 4.4: Pressure (psi) contour plots for three time snapshots during the course of a simulation. The reservoir is initially at 1100 psi. An injection well is located in the center of the model and it is operated under a BHP control 1828 psi. Four productions wells are located at the corners of the model, and are operated at 435 psi.



(a) 25 days



(b) 414 days



(c) 1314 days

Figure 4.5: Gas saturation time snapshots of a simulation of a reservoir that is initially oil saturated. An injection well located at the center introduces a gaseous mixture into the reservoir.

	GPRS	ADGPRS
Time Steps	164	164
Newton Iterations	735	640
Linear Iterations	4062	3641

Table 4.3: Time-stepping and solver statistics for a simulation using GPRS and ADGPRS.

the nonlinear solution loops of both simulators, and at the time of writing, these differences may lead to a alight variation in the number of nonlinear iterations to convergence.

	GPRS	ADGPRS	
Discretization (sec)	108	97	0.9%
Properties (sec)	141	214	1.5%
Linear Solver (sec)	176	171	0.98%
Simulation Total (sec)	426	494	1.16%

Table 4.4: Computational time

Table 4.4 presents the timing results obtained. In addition to total simulation time, the table lists the timings of three parts of the simulation process; the time spent on computing stencil or other spatial discretization operations, time spent in computing physical and thermodynamic properties, and time spent on linear solution. Factoring in the number of iterations taken, the overhead of the ADETL in this case is clearly negligible.

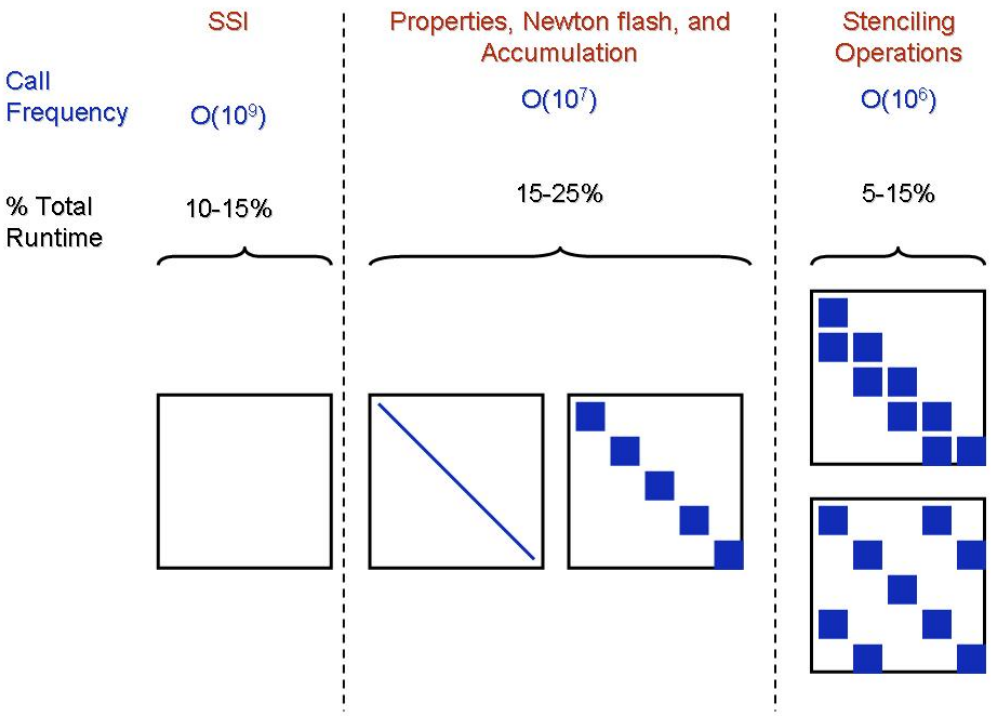


Figure 4.6: Derivative sparsity structures, call frequencies, and typical execution times of three major components of formulation and linearization routines; thermodynamic stability calculations, property and accumulation calculations, and stenciling routines.

4.2 Summary

To date, the general experience with ADGPRS is positive. Empirical results, including those presented in this discussion, show little computational overhead due to the use of the ADETL in constructing robust large scale general purpose simulators. Moreover, this is attained with little to no tuning or active optimizations using customized components heterogeneously as apropriate. At the time of writing, ADGPRS was implemented using block sparse gradients and SPLC expression building and evaluation routines exclusively. The results in this section reveal a glaring potential target area for optimization; the property calculation and phase behavior routines.

Figure 4.6 shows a survey of resultant Jacobian sparsity structures, call frequencies, and percent execution time for three parts of a residual call. These are thermodynamic flash calculation routines using the Sccessive Substitution Iteration (SSI)), accumulation, block property, and block Newton flash routines, and finally, operations invovlving indexing across blocks. The sparsity structures show the optimization opportunity rather clearly. The SSI routines are better off using floating point datatypes. The accumulation terms can be better computed using dense multivariate scalars with no assumption on sparsity. Finally cross-block terms seem to be optimal with the current choice; block-sparse multivariate gradients.

Chapter 5

Possibilities for the future of the ADETL

The ADETL is a modern extensible library that is large and continuously growing. The ADETL is architected in a multilayer style. The library provides generic datastructures and algorithms for the rapid development of flexible formulation and linearization modules. The highest layer of the library provides constructs that allow the user to define and operate with rather general concepts of formulations, unknowns, and state. One layer below this, the ADETL provides constructs that can be used to write any nonlinear system. The sparse analytical Jacobian is evaluated automatically behind the scenes. The derivative computations and Jacobian sparsity analysis routines are not only automatic and flexible, but they also result in very low computational overhead. The AD core of the library is built from the ground up using lower layer ADETL constructs. Such constructs provide encapsulated generic implementations of memory management, indirection, and iterative concepts. With compile-time switches, the programmer can essentially instantaneously produce simulation code tailored to various platforms.

The ADETL has enabled the development of ADGPRS, which makes the rapid modification of the choice of numerical variables a reality. Multi-user experience with ADGPRS is mounting and over three years since its inception, the developer and user base has been expanded significantly. Empirical results demonstrate that the

computational efficiency of ADGPRS is comparable to other large-scale hand-written codes.

The future of the ADETL is promising. In order to capitalize on this infrastructure, it is critical for the ADETL to continuously evolve. It would be unwise to neglect advances in language standards, compilers, and hardware architectures. Moreover, as the user base of the library continues to expand, and as all corners of the ADETL undergo general user stress testing, it is more than likely that unanticipated needs will arise. While it is an ambitious principle to uphold, the directions in which the ADETL grows are best set by the collective experience of its users. Moreover, the most useful feedback should come from users of the higher layers of the library, since they are the true clients. Exclusively using the input of the type of user that is completely proficient with and knowledgeable of the internal workings of the ADETL is like stamping the old brochures of a legacy code with the words, "and now, user friendly". Given this cautionary opinion, and based on current trends in the use of the ADETL in building ADGPRS, a few key advances are foreseeable. These directions fit nicely into a dichotomy. The first category of directions are those that build on content within the bounds of established ADETL concepts. The second is that of synthesizing new concepts to meet new kinds of needs.

5.1 Old concepts, new directions

The ADGPRS experience teaches by example a number of lessons on the practical use of AD in modern large scale simulation. Of these lessons, three stand out as first-order research directions that could lead to game changing contributions as far as the ADGPRS user base is concerned.

5.1.1 More datastructures and heterogeneous expressions across them

A lesson learned from building ADGPRS is that locally specialized datastructures, particularly for the gradient components, are paramount. The mathematical intermediary terms in a realistic sophisticated model involve constants, univariates, block computations, and stenciling and other inter-block operations. The ADETL provides total extendability to deliver all of these datastructures. This is something that should be taken advantage of. While the ADETL today does provide a wide range of datastructures, there has been little experience with cross-datastructure interactions. This will be an important upcoming challenge for ADETL developers. Cross-type expressions require some serious design choices to be made. While in mathematics, arithmetic operations are closed, the ADETL analogues are not necessarily so. The introduction of the generic type parameterizations require the designer and developer to agree on a system for closure; for example, a sparse multivariate expression, plus a dense multivariate expression yields a block dense resultant regardless. Once these type closure decisions are made, they need to be communicated in such a way as to avoid subtle logic errors.

5.1.2 What is a variable set today and what will it be tomorrow?

The ADGPRS development team experienced ADETL's ability to abstract formulation concepts comprehensively delivering automatic management behind the scenes. The initial experience raised a few brows. From the users' standpoint, the variable set concept and the notions of local differentiation contexts and state seemed unnatural. From the ADETL developer's perspective, this new territory seemed dense with mines. The variable set layer of the library when used incorrectly could lead to logical programming errors that are more subtle perhaps than any issue with hand coding a Jacobian matrix. The fact that the underlying notions of the layer are new ups the ante. It is anticipated that the upper-most layer of the ADETL will move towards one of two directions. The first direction is towards capping off the power of

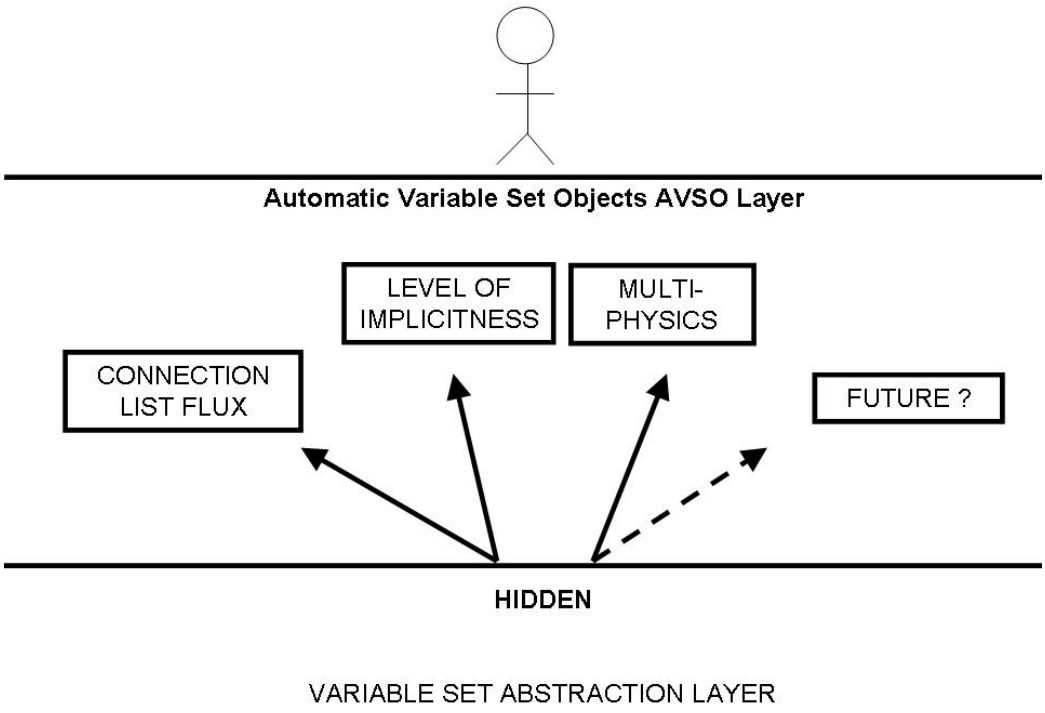


Figure 5.1: A schematic of a potential direction for the Automatic Variable Set Object AVSO layer of the ADETL. To reduce the potential for subtle logic errors, the AVSO provides a limited set of contexts to manage variables in a simulation code.

the layer by enforcing a strict and succinct list of ways in which it can be used. The other is more ambitious, and aims to re-enforce the abstract notions of variables and unknowns while providing domain specific concept checkers to validate code and help comb out any bugs in the ointment.

Figure 5.1 illustrates a mechanism by which to limit the range of use and mis-use of automatic variable sets. Essentially a library of a handful of specific contexts are available to the user. These contexts are familiar and concrete to domain experts. The advantage of such an approach is tighter concept and error checking. The potentially grave disadvantage is that the power to accommodate unexplored contexts of variable sets and activation systems is essentially limited. New concepts must be accommodated with an explicit horizontal integration which may not scale well.

The second possible direction is towards a standard specification and workflow check system. A debugging tool can be developed to outline to the user all differentiation contexts throughout the life of a simulation. The tool will need to "speak" the domain experts' language.

5.1.3 Thread safety first

An obvious direction is towards stretching the boundaries of the platform specific layers of the ADETL. It is anticipated that the primary contribution to this area will be thread safety in and of itself. That is, dynamic memory pools and expression object construction and destruction routines need to manage their own thread safety concerns. The idea of implementing parallel variations of the AD evaluation core is far less interesting. This is due to the ultra-fine level of granularity of such operations in the context of ADGPRS as a whole.

5.2 Synthesized Concepts For The ADETL

Beyond working within the realm of current ADETL concepts, the potential for advancement is bright. Two specific advancements are anticipated to significantly impact the way the ADETL is used. The first is the growing use of pre-compile time analysis and the hooks that such analysis could create when combined with the template metaprogramming style and even the generic parameterizations of the ADETL. The second exploits the fact that the expression parse graphs are available to the processor.

5.2.1 Automated performance tuning

In simulation, sparsity is a permanent reality. The primary tried and tested means to achieving acceptable computational efficiency is to recognize and the heavy register demands of sparse operations. The optimal parameters for features such as the dense block size and loop unrolling depths are both platform and context specific. The performance impacts of tuning such parameters can be on several orders of magnitude.

Despite this significant impact, such tuning is rarely conducted by say a specific user of ADGPRS or the ADETL. That is expected since the tuning process requires substantial expertise, and guiding it is very tedious. Experience proves that if the optimization is not automatic or inherent then people will avoid it.

Automatic tuning is a form of pre-compilation analysis. In fact it is often performed by install-time scripts. One successful example of an automated tuning system in use today is the ATLAS which is a performance tuned BLAS. Another is the sparse linear algebra package OSKI. The OSKI system optimizes its register blocking components for the actual machine the library is installed on. Optimal block sizes are determined for various datastructures and linear algebra operations. A potential direction to expand the ADETL is to introduce an install time optimization system. The system would run a few residual formulation and linearization calls so that a high code coverage is achieved. The optimization parameters are not only sparsity related, but also related to selecting amongst various ADETL datastructures and algorithms locally throughout the application.

5.2.2 Code verification, generation, and evolution

The ADETL introduces the idea of caching partial or complete parse graphs. This ability introduces considerable potential. Automatic code verification, generation, and documentation can become a reality. The formulation and linearization graphs can be serialized into any mathematical language specification to be verified or documented in a visual mathematical language. Another application would be to serialize to symbolic manipulation systems allowing two-way plug-ins to various problem solving environments.

With a longer term outlook, the ability to automatically generate variations on a simulator can lead to higher order applications. Once such potential is in situations where a physical phenomenon is observed and yet the governing physical model is unknown. In such cases, it is conceivable that an inverse problem may be solved where the control parameter is the governing model itself or some constitutive relation component. The target could be the observation itself, or asymptotic characteristics

over a space of governing parameters.

Part II

The Nature Of Nonlinearities In Simulation And Solution Methods That Understand Them

Chapter 6

Introduction

Growing interest in understanding, predicting, and controlling advanced oil recovery methods emphasizes the importance of numerical methods that exploit the nature of the underlying physics. Multi phase, multi component, flows through subsurface porous media couple several physical phenomena with vastly differing characteristic scales. Moreover, scale separation is not always apparent. The fastest processes such as component phase equilibria occur instantaneously and subsequently they are modeled using nonlinear algebraic constraints. Mass conservation laws govern the transport of chemical species propagating through an underlying flow field. These transport phenomena are near-hyperbolic. In the limit of low capillary numbers, the transport equations are purely advective and they give rise to an evolution with a finite domain of dependence. In the other limit, the transport problem is diffusive. Moreover, the underlying flow field itself is also transient, and it evolves with parabolic character. In the limit of no total compressibility, the flow field reaches instantaneous equilibrium, and is governed by an elliptic equation. Constitutive relations such as that for the velocity of a phase couple the variables across governing equations in a strongly nonlinear manner. Consequently, one challenge in modeling large scale flows through general porous media is in resolving this coupling without sacrificing stability. Additional sources of complexity include the heterogeneity of the underlying porous media, body forces, and the presence of wells which are typically operated using busy, and often discrete, control schedules.

In practice, a rich collection of numerical treatments is used to model porous media applications. For a classic review, see [4]. Broadly, these numerical methods can be partitioned into two categories based on the tactics used to deal with nonlinear coupling. One approach is to seek a collection of methods that are tailored to resolving the physics within specific distinct regimes. It is natural that within such regimes, the physics are dominated by one type of model over another. With *a priori* knowledge of how these regimes interact and of when transitions occur, tailored methods can be combined and applied adaptively as appropriate through the course of a simulation. Examples of methods in this class are Operator Split schemes [75, 49], and semi-implicit methods [4]. A second approach is to seek a class of methods that resolve a wide range of processes in a fully coupled manner. Rather than deal with stiff nonlinear coupling in the construction of the numerical approximation, such methods delegate that responsibility onto the underlying nonlinear solver and timestep controllers. Robust nonlinear solvers are particularly important when physical regime transitions occur frequently and in a complex manner. Overarching both philosophies is the fact that strong nonlinear physical coupling across a wide range of timescales poses challenges. In the split-and-couple, semi-implicit approach, severe restrictions on the timestep size usually arise, and timestep control for stability requires deep insight into the underlying physics. In the fully-coupled, fully-implicit approach, while there are no stability restrictions in the sense of the discrete approximations, the resulting algebraic nonlinear systems are difficult to solve ([4, 37, 43]).

This work focuses on exploiting fundamental understanding of implicit methods for oil recovery problems in order to devise nonlinear solution strategies that converge efficiently all the time. In this introduction we study the nature of nonlinearities that are typical of porous media flows, and we analyze the performance of the current state-of-the-art in nonlinear solvers.

6.1 Timestepping and nonlinearity in implicit models

In fully-implicit, fully-coupled methods, all unknowns are treated implicitly. This gives rise to a fully coupled nonlinear system of discrete equations that must be solved for at each timestep. One attractive aspect of this approach is its unconditional stability which is obtained at the cost of tight nonlinear coupling between fundamentally differing phenomena; e.g. parabolic and hyperbolic components. Two practical shortcomings of this approach continue to receive attention from the research community (see for example [37, 43]). The first is that available solution methods for the discrete nonlinear systems may themselves not be unconditionally convergent. The second aspect is that regardless of the technical details of the particular solution method, the computational effort required to solve large coupled systems can be significantly larger than that for de-coupled, localized computations, such as in a convergent step of a semi-implicit method. The practical implication of these two shortcomings is the use of *time-step chops*. With a try-adapt-try strategy, an attempt to solve for a time-step is made. If that fails within a specified finite amount of time, the time-step is adapted heuristically, and the previous effort is wasted.

Current simulators rely on a fixed-point iteration, such as a variant of Newton's method in order to solve these problems (see for example [4, 52, 25]). For general problems, Newton's method is not guaranteed to converge, and it is known to be sensitive to the initial guess, which must be supplied somehow. In most reservoir simulators, the initial guess to the iteration is the old state. For small timestep sizes, this is a good approximation to the new state, and is therefore likely to be a good starting point for the Newton iteration. For larger timesteps, however, this is less likely to be the case, and the iteration may converge too slowly, or even diverge. To motivate why Newton's method may fail in practice, we consider its origin and some examples.

6.1.1 The Newton Flow and Newton-like methods

At each time-step of an implicit simulation, given the current state, $U^n \in \mathbb{R}^N$, and a fixed time-step size, $\Delta t > 0$, we seek to obtain the new state, $U^{n+1} \in \mathbb{R}^N$, by solving a nonlinear residual system, $R : \mathbb{R}^N \longrightarrow \mathbb{R}^N$ as written in Equation 6.1.1.

$$R\left(U^{n+1}; \Delta t, U^n\right) = 0 \tag{6.1.1}$$

Newton's method generates a sequence of iterates, $[U^{n+1}]^\nu$, $\nu = 0, 1, \ldots$, that hopefully converges to the new state, U^{n+1}. Denoting the Jacobian matrix of the residual with respect to the new state as $\mathbf{J}$, this sequence is generated starting from the old state according to Equation 6.1.2 below.

$$\begin{aligned} \left[U^{n+1}\right]^0 &= U^n \\ \left[U^{n+1}\right]^{\nu+1} &= \left[U^{n+1}\right]^\nu - \mathbf{J^{-1}} R\left(\left[U^{n+1}\right]^\nu; \Delta t, U^n\right), \quad \nu = 0, 1, \ldots \end{aligned} \tag{6.1.2}$$

Newton's iteration itself can be regarded as an explicit, first-order time-stepping scheme for a particular dynamical system. To see this, suppose that the Newton iteration index, ν, is actually a continuous quantity, and consider the dynamical system in Equation 6.1.3 below.

$$\begin{aligned} U^{n+1} &= U^n && \nu = 0 \\ \tfrac{dU^{n+1}}{d\nu} + \mathbf{J^{-1}} R\left(U^{n+1}; \Delta t, U^n\right) &= 0, && \nu > 0 \end{aligned} \tag{6.1.3}$$

Comparing Newton's iteration (Equation 6.1.2) to this dynamical system, it is clear that Newton's method approximates the derivative of the new state with respect to the embedded time, ν, using a first order finite difference with a unit step size, $\Delta\nu = 1$. The forcing function of the dynamical system is the inverse of the Jacobian acting on the residual evaluated at the old embedded step, ν. Note that the embedded time, ν, is unrelated to the physical time step, Δt, which is fixed throughout Newton's

iteration. Since the forcing function is evaluated at the old embedded step, Newton's iteration is an explicit first-order discretization of the system in Equation 6.1.3. The embedded time-step, $\Delta\nu$, is a step along the mathematical field defined by the system in Equation 6.1.3, which we refer to as the *Newton Flow*. At the solution of the time-step, the forcing function is zero, and so, the solution is a *stable fixed-point* of the Newton flow.

Since explicit first-order time-stepping may be unstable (has a time-step restriction), Newton's iteration may not converge, even though the continuous Newton Flow may be well-conditioned. On the other hand, for general problems, when Newton's method does converge, there are no guarantees on how fast it will do so.

In practice, a convergent iteration for a given time-step size may be too slow, and it becomes necessary to stop if a prescribed maximum number of iterations is reached before convergence is achieved. In such cases, the iterates are discarded, and this process is repeated using a smaller time-step. It is not known rigorously which smaller time-step could be used with success. Several heuristics are often employed to attempt this; nevertheless, all such methods fall into a try, adapt, and try-again strategy. It is difficult to derive rigorously a relation between time-step size and convergence rate, since that would ultimately involve a stability analysis of the system in Equation 6.1.3, which is both highly problem dependent, as well as analytically intractable for realistic problems. Moreover, such an analysis is more likely to generate iterations that needlessly follow the Newton flow field too closely, resulting in a highly inefficient iteration.

6.1.2 An example to illustrate challenges encountered by Newton's method

To illustrate the pathologies that Newton's method may run into in practice, we consider examples of incompressible two phase flow under gravity effects. Before presenting these results, we introduce the details of the Buckley-Leverett model.

The Buckley-Leverett model

We consider a two-phase Buckley-Leverett problem in one-dimension with gravity effects. Flow is through a domain with unit length, $x \in [0,1]$, and the unknown is the water saturation, $S(x,t)$. The governing equation in conservation form is written in Equation 6.1.4.

$$\begin{aligned} S_t + f(S)_x &= 0 \\ S(x, t=0) &= S_{init} \\ S(x=0, t) &= S_{inj} \end{aligned} \tag{6.1.4}$$

In Equation 6.1.4 above, the initial saturation is denoted S_{init}, and the left boundary condition, S_{inj}, is fixed. The fractional flow function,$f(S)$, is defined in Equation 6.1.5 below, where the viscosity ratio, M^0, and the Gravity Number, Ng, are constant parameters.

$$f(S) = Kr_w \left(\frac{1 - NgKr_o(S)}{Kr_w(S) + M^0 Kr_o(S)} \right) \tag{6.1.5}$$

In this problem, we assume Corey relations [18] for relative permeability (Equations 6.1.6 and 6.1.7).

$$K_{rw} = K_{rw0} \left(\frac{S - S_{wr}}{1 - S_{wr} - S_{or}} \right)^{n_w} \tag{6.1.6}$$

$$K_{ro} = K_{ro0} \left(\frac{1 - S - S_{or}}{1 - S_{wr} - S_{or}} \right)^{n_o} \tag{6.1.7}$$

Note that for an up-dip case, $Ng > 0$, or a down-dip case, $Ng < 0$, the fractional flow function may exhibit a local minima or maxima respectively. This leads to the possibility of counter-current flow. In such cases, we refer to the extremum as the *sonic point*; a local maximum or minimum in the fractional flow, denoted f^*, and occurring at a saturation S^*.

We apply a fully-implicit discretization in time, and a first-order upwind discretization in space. On a uniform mesh with N cells, the numerical saturation unknown at the n^{th} timestep and the i^{th} cell is denoted as S_i^n. The numerical scheme is written as in Equation6.1.8 below.

$$S_i^{n+1} - S_i^n + \frac{\Delta t}{\Delta x}\left[F\left(S_i^{n+1}, S_{i+1}^{n+1}\right) - F\left(S_{i-1}^{n+1}, S_i^{n+1}\right)\right] = 0, i = 1, \ldots, N \tag{6.1.8}$$

In Equation 6.1.8, the timestep size is denoted as Δt, the mesh spacing as $\Delta x = \frac{1}{N}$, and the numerical flux as F. We apply a Dirichlet boundary condition on the left of the domain, and a second order treatment of a free boundary condition on the right. These are numerically prescribed by Equations 6.1.9 and 6.1.10 below.

$$S_0^n = S_{inj} \tag{6.1.9}$$

$$S_{N+1}^n = 2S_N^n - S_{N-1}^n \tag{6.1.10}$$

For general fractional flow functions it is necessary to apply an entropy satisfying upwind condition for the numerical flux. This condition corresponds to the analytical solution of cell-face Riemann problems. The condition applied is described by Equation 6.1.11 below.

$$F(a,b) = \begin{cases} \min\limits_{a \le s \le b} f(s) & a \le b \\ \max\limits_{b \le s \le a} f(s) & \text{otherwise} \end{cases} \tag{6.1.11}$$

Note that for fractional flows with sonic points, the numerical flux at a cell interface may be independent of both the left and right cell saturations when it is evaluated at the sonic point.

We illustrate fully-implicit solutions for two cases of Problem 1. The first case is a horizontal piston-like displacement, and the second includes gravity effects and

exhibits counter-current flow. In both cases, the injection saturation is one, $S_{inj} = 1$, and the relative permeability functions are quadratic with end-points of zero and one.

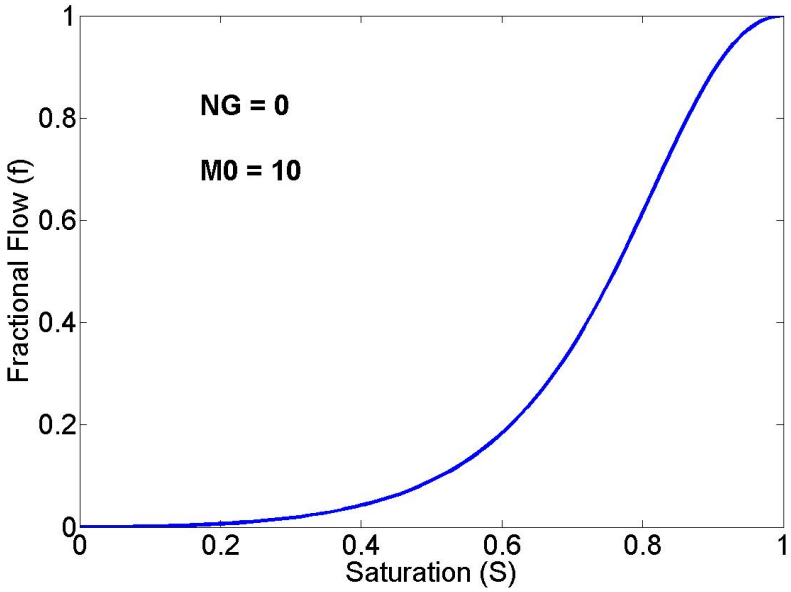
In the horizontal case, the end-point mobility ratio M^0 is chosen as 10, the Gravity Number Ng is chosen as 0, the initial saturation, S_{init}, is zero, and $N = 150$. Figures 6.1(a) and 6.1(b) show the fractional flow curve and solution profiles for various timesteps, respectively.

In the down-dip case, the end-point mobility ratio, M^0, is chosen as 0.5, and the Gravity Number, Ng, is chosen as -5 for a down-dip problem. The number of grid blocks used is $N = 150$. The initial condition has a saturation of one up to a distance of 0.34, and a saturation of zero elsewhere. Figures 6.2(a) and 6.2(b) show the fractional flow curve and solution profiles for various times.

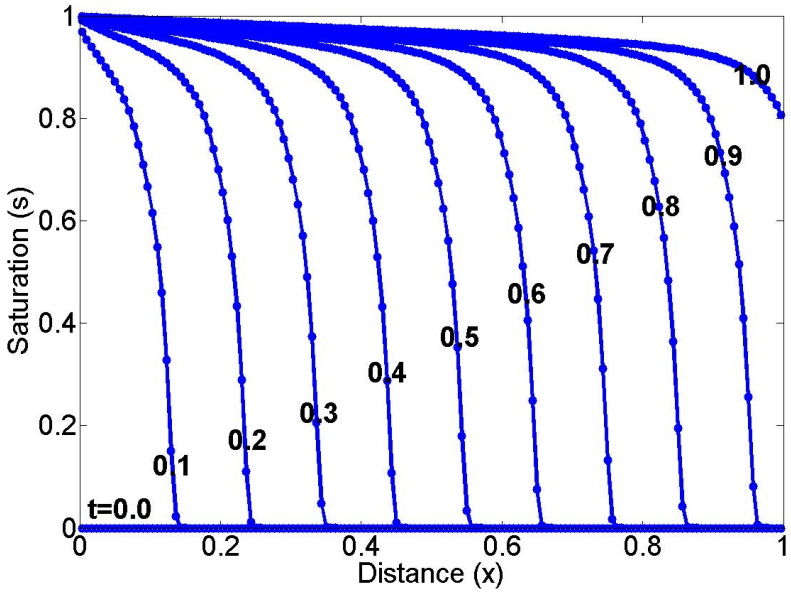
A two cell Newton Flow example

We apply a fully-implicit discretization on a discrete domain with two-cells; $N = 2$. The corresponding residual is regarded as a function in two-dimensional space. The first dimension is the range of possible saturations in the first cell $S_1 \in [0, 1]$, and second dimension is the range in the second cell $S_2 \in [0, 1]$. We compare the Newton iterations to the Newton flow in this two-dimensional space for two physical cases. The first is a horizontal case ($Ng = 0$), and the second is a down-dip problem ($Ng = -3$). In both cases, we use a unit injection saturation, $S_{inj} = 1$, and a uniform zero initial saturation, $S_{init} = 0$. The corresponding fractional flow curves to the two cases are illustrated in Figures 6.3(a) and 6.3(b) respectively.

Figures 6.4(a) and 6.4(b) illustrate a Newton process (series of straight thick blue arrows) for both physical cases starting from the old state $S^n = (0, 0)$ for a time-step of one. The contour lines on the figures are those of the norm of the residual, $\|R\left(S; \Delta t_D, S^n\right)\|$ over $S \in [0, 1] \times [0, 1]$. The thick curved black lines are integral numerical solutions of the Newton flow of Equation 6.1.3 emanating from various starting points on the boundary of the two-dimensional space. These integral curves are obtained using a high fidelity numerical integrator (variable-order Runge-Kutta with automatic step control), and they illustrate the continuous paths that Newton's method attempts to approximate, should it have been started from these various

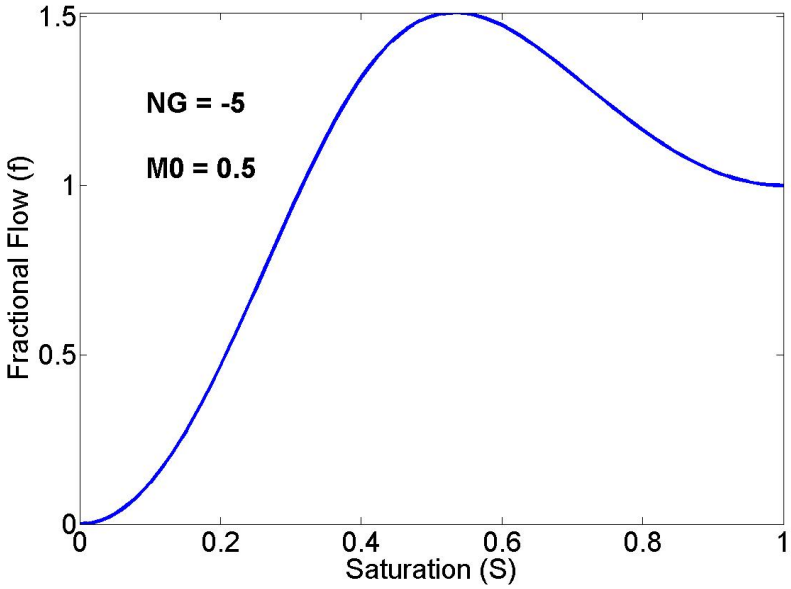


(a) Fractional flow curve

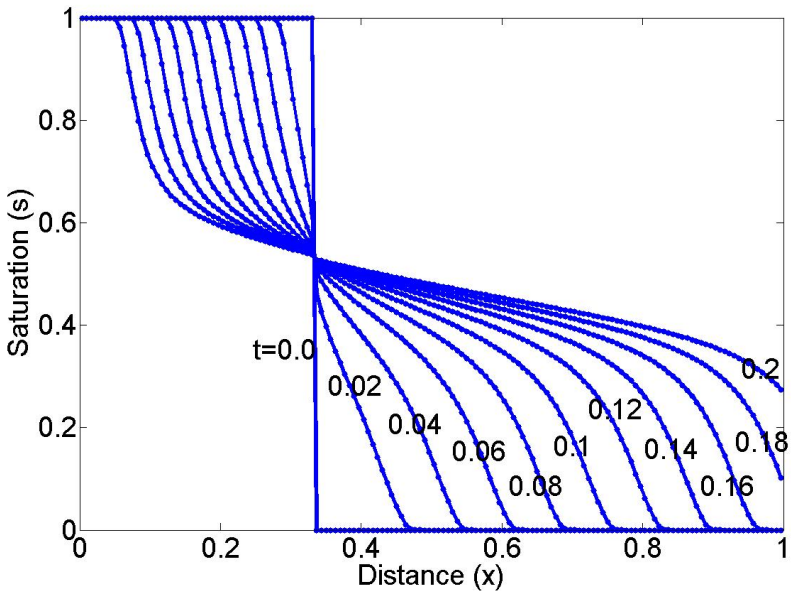


(b) Solution profiles

Figure 6.1: Fractional Flow curve and saturation solution profiles for a 1-dimensional Buckley-Leverett problem without gravity.

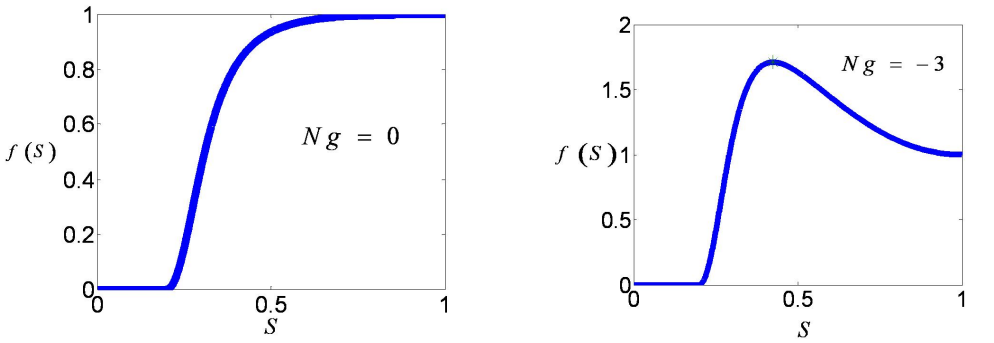


(a) Fractional flow curve



(b) Solution profiles

Figure 6.2: Fractional Flow curve and saturation solution profiles for a down-dip 1-dimensional Buckley-Leverett problem.



(a) Horizontal displacement $Ng = 0$ (b) Down-dip displacement $Ng = -3$

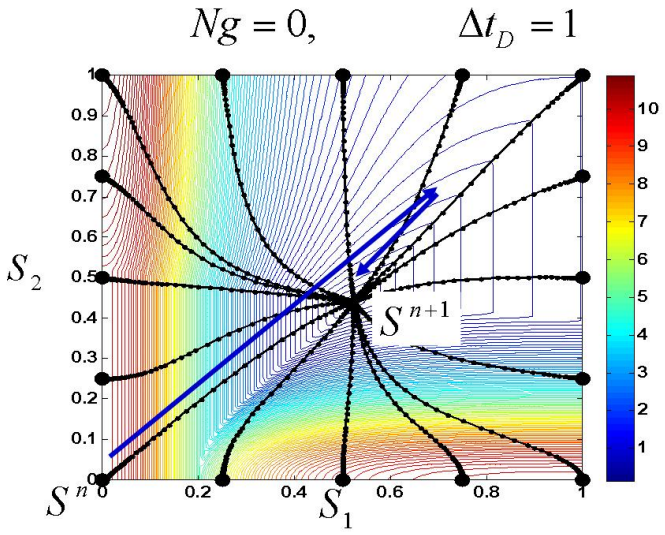
Figure 6.3: Fractional Flow curves for a 1-dimensional Buckley-Leverett problem.

initial guesses.

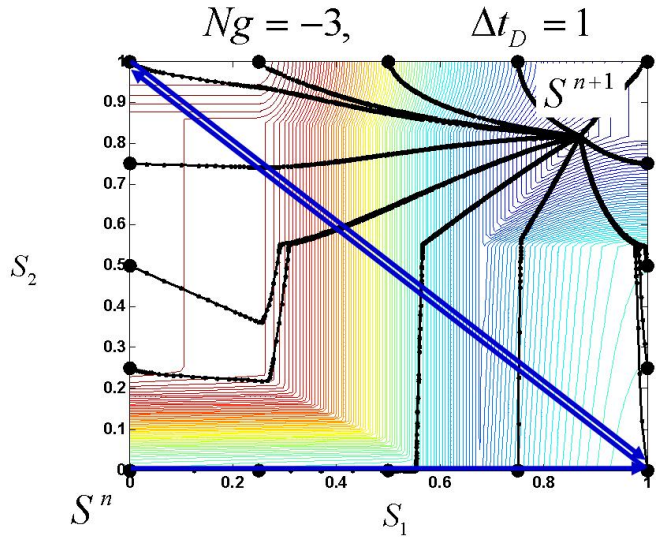
For the horizontal case, Figure 6.4(a) shows that Newton's method converges to the solution $S^{n+1} = (0.5, 0.45)$ in two iterations. Moreover, the Newton flow integral curves are generally smooth curves, all of which flow towards the solution. On the other hand, Figure 6.4(b) shows the iterates obtained for the down-dip case. Here, Newton's method diverges, with the iterates alternating between the points $(1, 0)$ and $(0, 1)$. For this case, the Newton flow integral curves are not smooth, and over saturation loci where the upwind directions change, there are clear kinks along the Newton integral curves. The smooth region around the solution is considerably smaller than that for the horizontal case in Figure 6.4(a). In theory, while following the Newton flow more accurately can lead to an increased robustness, the computational efficiency of doing so may be unjustifiable. In order to manage this trade-off, several variants of Newton's method have been devised, and are said to *safeguard* the iteration by improving it's ability to follow the Newton flow in some fashion.

6.2 Safeguarded Newton-Like Methods In Reservoir Simulation

Given the possibility of divergence of Newton's method for general problems, a number of variants have been devised to dampen the Newton updates. All of these



(a) Horizontal displacement $Ng = 0$



(b) Down-dip displacement $Ng = -3$

Figure 6.4: Residual norm contour lines, high-fidelity Newton flow integral paths (dotted), and Newton iterations (arrows) for two cases of Problem 1.

methods, which are said to *safeguard* the iteration, can be viewed as different ways to specify a diagonal matrix $\mathbf{\Lambda} = \text{diag}\left(\mathbf{\Delta\nu_1}, \ldots, \mathbf{\Delta\nu_N}\right)$ in a safeguarded Newton iteration written in the general form given by Equation 6.2.1.

$$\left[U^{n+1}\right]^{\nu+1} = \left[U^{n+1}\right]^{\nu} - \mathbf{\Lambda J^{-1}} R\left(\left[U^{n+1}\right]^{\nu}; \Delta t, U^n\right) \tag{6.2.1}$$

These diagonal weights can be interpreted as local time-steps in the explicit integration of the Newton flow introduced in the previous section. The standard Newton's method selects all of these weights to be unity, and subsequently, in cases where the underlying Newton flow changes too rapidly, the iteration may not converge.

There are several classic approaches to safe-guarding Newton's method, such as the line-search and trust-region algorithms described in [52] and [25], for example. In these methods, all entries of the diagonal are identical, implying that the Newton direction is simply scaled by a constant factor. In these methods, the choice of this scale is dictated by the rate of change in the residual norm along the Newton direction or within a neighborhood about the current iterate.

In commercial reservoir simulators, heuristic strategies have also been devised to safe-guard Newton's method. Such strategies select the diagonal scaling entries on a cell-by-cell basis using physical arguments. In practice, these heuristics are known to improve convergence (larger time-step sizes can be converged from the same initial state). There is a common underlying hypothesis shared by the most commonly employed heuristics. The hypothesis may be motivated by the residual contours in Figure 6.4, and by the shape of common fractional flow curves (e.g. Figure 6.3). The hypothesis is that the nonlinearity of the residual is in some sense dictated by the structure of the flux function in each cell. That is, suppose that a nonlinear Gauss-Seidel iteration is applied to the residual system as a whole. Then for each Gauss-Seidel iteration, the residual in each cell must be solved sequentially. To solve each of these single cell nonlinear residuals, we may apply a scalar Newton process, obtaining the saturation in the current single cell, assuming all other cell saturations are known. These scalar Newton iterations need to be safe-guarded by some scalar damping protocol ([46]). If this protocol assures cell-wise convergence of a Newton

iteration, then the outer Gauss-Seidel iteration at least has a hope of converging. The hypothesis is that if the same **scalar** damping protocols are also applied on saturation variables, cell-by-cell, in the context of **system** Newton updates, then that too is more likely to converge. Inspecting Figures 6.3 and 6.4, we can deduce that saturations around end-points and inflection points produce particularly sensitive Newton directions. The hypothesis is to apply a cell-by-cell (diagonal) damping factor to limit large changes around such physically sensitive boundaries.

The following is a sample list of heuristic scalar damping protocols for Newton methods in Black-Oil simulation ([33, 51]):

1. **Eclipse Appleyard**: for saturations only, cell-by-cell, scale back changes from immobile to mobile so that they are barely mobile. For changes from mobile to immobile, scale back to barely mobile. Ensure saturations are between 0 and 1. Algorithm 1 performs this update for a cell-variable.

2. **Modified Appleyard**: do the same as method 1, and in addition require that no saturation change in a cell is greater than some small amount in magnitude, which is usually chosen as 0.2. This cell-wise update is described by Algorithm 2.

3. **Geometric Penalty**: do the same as method 1, and in addition require that no saturation change in a cell is greater than 20 percent of the original saturation.

The focus of these cell-wise strategies is to avoid large Newton corrections to saturation variables, as may arise when crossing phase boundaries or other pathological properties of the fractional flow curve. Next, we examine practical examples of the behavior of these methods.

6.2.1 Examples of the performance of state-of-the-art solvers.

We observe empirical evidence over a suite of numerical experiments conducted on cases of Problem 2. For various cases of imposed well conditions, and initial saturation distributions, the following experiment is performed. A sequence of time-step sizes is fixed, and for each time-step size, the residual is solved using each of four protocols; Standard Newton (SN), Eclipse Appleyard (EA), Modified Appleyard (MA), and a

Algorithm 1 APPLEYARD-SAFE-UPDATE(u, δ)

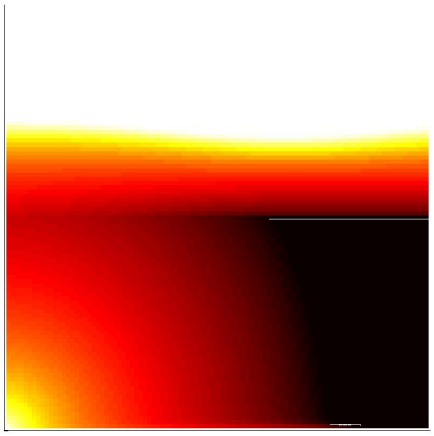
Require: u is a cell variable and δ is an update to it.
Ensure: The updated variable $u^* = u + \delta$ satisfies the Appleyard heuristic

```
u* ← u + δ
if u is a saturation variable then
  Set the irreducible and residual saturations, S_I and S_r, such that 0 ≤ S_I ≤ u ≤ S_r ≤ 1.
  Fix a small positive constant 0 < ϵ << 1. This can be related to machine precision; √ϵ_machine
  if u* > S_r then
    if u < S_r − ϵ then
      u* ← S_r − ϵ
    else
      u* ← S_r
    end if
  end if
  if u* < S_I then
    if u > S_I + ϵ then
      u* ← S_I + ϵ
    else
      u* ← S_I
    end if
  end if
end if
return u*
```

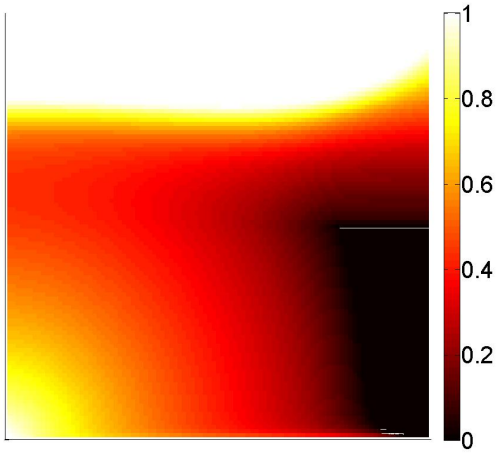
Algorithm 2 MODIFIED-APPLEYARD-SAFE-UPDATE(u, δ)

Require: u is a cell variable and δ is an update to it.
Ensure: The updated variable $u^* = u + \delta$ satisfies the Modified Appleyard heuristic

```
if u is a saturation variable then
  Fix a positive constant 0 < C_mac < 1. This is typically 0.2
  δ ← SGN(δ) min (C_mac, |δ|)
end if
u* ← APPLEYARD-SAFE-UPDATE(u, δ)
return u*
```



(a) 0.05 PVI



(b) 0.1 PVI

Figure 6.5: Oil saturation snapshots over a simulation of a model with oil on the bottom initially, a water injector in the lower right corner, and a pressure controlled producer in the upper-right.

Algorithm 3 MODIFIED-APPLEYARD-NEWTON-SOLVER($U^n, \Delta t$)

Require: $\Delta t \geq 0$ and $U^n \in \mathbb{R}^N$
Ensure: $R\left(U; \Delta t, U^n\right) = 0$, and niter $< MAXITER$
  niter $\leftarrow 0$
  $U \leftarrow U^n$
  while niter $<$ MAXITER, and, $\|R\left(U; \Delta t, U^n\right)\| >$ RTOL **do**
    $\delta \leftarrow -J\left(U, \Delta t; U^n\right)^{-1} R\left(U, \Delta t; U^n\right)$
    for $i = 0$ to $i = N$ **do**
      $U\left(i\right) \leftarrow$ MODIFIED-APPLEYARD-SAFE-UPDATE($U\left(i\right), \delta\left(i\right)$)
    end for
    niter $\leftarrow$ niter $+1$
  end while
  return U, niter

Geometric Penalty (GP). Each target time-step is solved using the old state as a starting guess. The number of iterations is noted, as are the convergence characteristics. In a representative case, the initial condition has oil on the upper half of the domain, and water below. A rate controlled water injector is applied in the lower-left corner, and a pressure control producer is placed in the upper right corner. Figures 6.5(a) to 6.5(b) show saturation solutions obtained for independent time-steps of sizes 0.05 and 0.2. Figure 6.6 shows the iteration count profiles.

As is empirically typical of problems involving two or more physical driving forces, SN and EA diverge (do not converge no matter how many iterations are spent) for time-steps larger than 0.005. The MA and GP strategies are convergent over the entire tested range of step sizes. What is key to this discussion is that while strategies such as MA and GP lead to convergent iterations, the number of iterations required grows with the requested time-step size. In practice, a maximum number of iterations is imposed by the user for efficiency considerations. So, while MA may well converge in 150 iterations, if the maximum allowable is 10, then the iteration is halted. The 10 iterations are wasted, and a heuristic must be used to scale back the time-step size to one for which MA may converge in less than 10 iterations. It is this try, adapt, and try again strategy that often brings a simulation of a complex model with several physical transitions to a grinding halt.

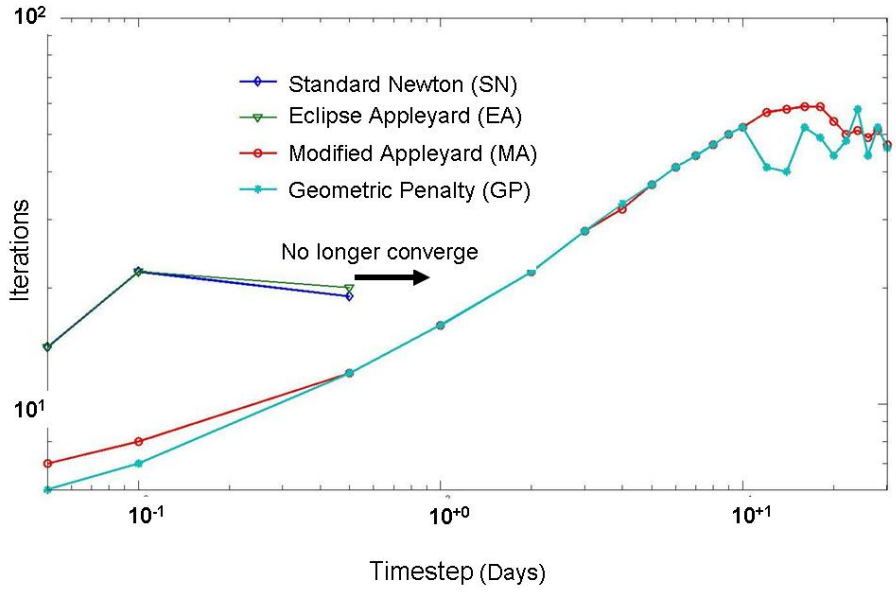


Figure 6.6: The number of iterations required to solve a sample time-step size using Standard Newton (SN), Eclipse Appleyard (EA), Modified Appleyard (MA), and a Geometric Penalty (GP).

Chapter 7

Adaptive Localization

Broadly, models of multi-phase flow in porous media are of mixed parabolic-hyperbolic character. They often include stiff nonlinear source terms and as well as additional equations in the form of nonlinear algebraic constraints. In practice, such models are numerically approximated using coupled implicit time-stepping strategies as the method of choice. Implicit coupled methods require little in the way of timestep control for numerical stability. A frequently cited limitation of implicit methods however is that they require the solution of large coupled nonlinear systems. The use of iterative solution approaches is therefore inevitable. Furthermore, the more popular iterative methods employed to solve such systems are of the quasi-linearized type. Each iteration within such approaches requires the solution of a large linear system. The challenges of designing large scale variants of nonlinear solution methods continues to be the topic of numerous research efforts. All large scale variants generally fall into one of two categories; methods that solve the original nonlinear system using linearizations of a related less computationally expensive problem, and those that solve a more tractable approximate nonlinear system using exact linearizations.

Several methods have been designed to reduce the computational cost associated with solving a timestep by substituting the original large problem with a smaller one that is in some sense representative of the original one ([15, 54, 50]). All of these methods attempt to exploit the property of local domain of dependence which is characteristic of traveling wave problems. The approximation methods choose a

coarser proxy representation by focusing the modeling detail around critical features and locales within the domain at a given point in time. The result is that the proxy model is typically an order of magnitude smaller than the original problem which it hopefully approximates. In such approaches, it is often challenging to guarantee that solving the coarse model results in an adequately representative solution to the original nonlinear problem. This is a potentially severe drawback, particularly in situations where accuracy is of any concern.

Another approach towards developing adaptive methods is to stick to a standard scheme and spatial mesh, and to then build adaptivity into the underlying solver. Because neither the mesh nor the scheme are modified, there will be no introduction of accuracy or stability issues. Large-scale variants of Newton's method, for example, reduce the computational effort required for each iteration without degrading the local nonlinear convergence rate ([43, 73]). Common approaches to achieving this are Inexact- and Quasi-Newton methods, or a combination of both ([52, 25]). Given an accurate and updated Jacobian matrix, Inexact-Newton (IN) methods apply approximate linear solvers to compute the Newton direction. When combined with parameterized controllers for the linear solver accuracy, IN methods can successfully manage an efficiency to accuracy trade-off. For example, Krylov-Newton controllers can compute crude Newton steps away from the solution, and more accurate ones within the quadratic convergence basin [24]. Quasi-Newton (QN) methods, on the other hand, use approximations to the Jacobian matrix itself, thereby reducing the computational effort required to compute it. Examples of such approaches are the rank-one family of updates that seek to use the iterate information in order to quickly update the Jacobian matrix. While it is challenging to control the accuracy of such updates, QN methods such as the Broyden update class [10] are the method of choice when the Jacobian itself is not available for computation. While such approaches always produce accurate solutions to the original problem, the attained speed-ups are seldom on an order of magnitude.

The objective of this work is devise large-scale solution methods to obtain quasi-linearized iterates with the best of both approaches; the computational speed-ups typically attained by using locality information, and the robustness and accuracy of

exact methods with indirect solvers. The idea is to solve the large linear systems arising through the course of IN and QN strategies by solving considerably reduced systems while still guaranteeing accurate computation of Newton directions to the original problems. The key is to exploit the same locality ideas used by approximate methods such as dynamic upscaling but within the context of solving the full linear systems.

7.1 Basic ideas and motivation

Owing to the nature of the constitutive relations, the Reservoir Simulation governing equations exhibit mixed parabolic-hyperbolic character; the mass flow potential field evolves with a diffusive character, and it is coupled to the underlying advective wave propagation of phase saturation and chemical concentrations.

Pressure and Temperature are the two Reservoir Simulation state variables that predominantly act as potentials for diffusive processes. While Pressure forms the potential for Darcy flow, Temperature forms the potential for conductive heat transfer. As is the case with all stable parabolic models, transient evolution processes occur over a finite characteristic timescale, the duration of which leads undisturbed diffusion processes to their steady state. Despite this finite temporal characteristic scale, transient diffusive processes occur over an infinite spatial domain-of-dependence. Subsequently, diffusive changes such as in Pressure fields are felt throughout the entire domain instantaneously, despite being driven locally such as by a well that begins to draw down. Within the context of Reservoir Simulation models, transient diffusive processes are driven by one of two mechanisms. The first is by models of human control; for example, well controls or subsurface heaters that are turned on or off during the course of a simulation in order to achieve certain engineering objectives. The second is due to coupling to other state variables that are themselves transient and not necessarily governed by a parabolic character. For example, in incompressible flow, the conservation equation that is aligned with pressure is elliptic, and yet pressure can be time-dependent if the mobility coefficients in the elliptic operator depend on other state variables such as Saturation or Concentration which evolve with

hyperbolic character. Regardless of the driving mechanism, diffusive transients in the context of Reservoir Simulation occur rapidly leading to some form of a temporally local steady state. Subsequently, the lack of spatial locality associated with diffusive changes does not undermine the overall locality of reservoir processes.

Phase Saturation, concentration, and other chemical composition variables that are aligned with conservation equations propagate predominantly as traveling waves. Such waves propagate with a local spatial support within the domain, and qualitatively at least, rarefaction or spreading waves are the spatially limiting factor of locality. In compositional problems, the transport equations are a couple system of nonlinear conservation laws, and with the fully implicit treatment, exploiting locality can lead to substantial computational savings.

This qualitative understanding of the spatial and temporal locality of transient effects is the common underpinning of various numerical strategies in Reservoir Simulation that aim improve the computational efficiency attained for a given accuracy. Such strategies tend to exploit locality by following the time evolution of the solution itself, and by using forward extrapolation in time in order to introduce some form of adaptivity. In this work, we aim to bootstrap the locality information directly, without extrapolation. In particular, the key hypothesis is that over a timestep n, if only a few components of the old state vector U^n differ from the solution state, U^{n+1}, then every iteration of a quasi-linearized solution process for U^{n+1} will also be limited in its spatial support, and every Newton step for example will be sparse to some degree. Then at any timestep, given the old state vector and the current iterate, can we already identify the vector components that will change over the course of the next iteration?

At any point in time, the upwind directions on the simulation mesh form a network through which mass transfer takes place over the duration of a timestep. The action of linearized state updates such as a Newton Step can be viewed as propagating material balance errors over the domain during a timestep. These updates are computed using the inverse of the simulation Jacobian matrix, which itself incorporates upwinding information as well as local wave speeds. Such updates are typically non-zero only within the vicinity of sharp fronts in the old state. The objective is to devise a

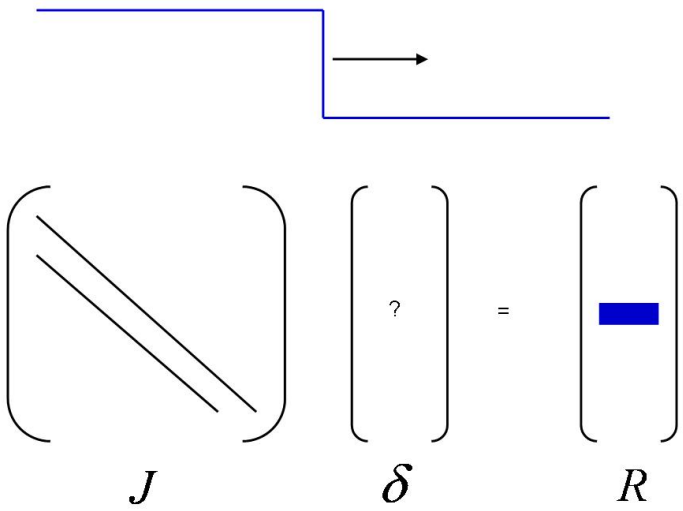
quantitative and reliable strategy to exploit this locality prior to solving the linear system. Two key ideas underly the localization algorithms developed. The first is that in any problem requiring the computation of a linearized update, one indicator of the local origins of wave sources is nonzero entries in the right hand side residual vector. The second idea is that a measure of the directionality and locality of propagation of the error sources over a linearized update is the upwind directed graph formed by the Jacobian matrix.

We examine these properties starting with a focus on pure transport systems, followed by coupled parabolic-hyperbolic problems.

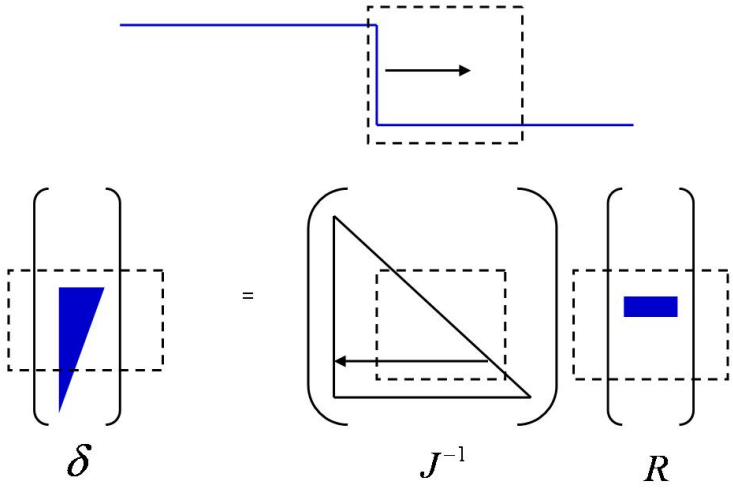
7.2 Locality in transport phenomena

Before exploring locality mathematically we present a high-level outline through a simple example. Figure 7.1(a) is a depiction of the problem of computing a linear update, δ, given an initial saturation state for a given timestep. The saturation is assumed to be a piston-like front, and the flow is from left to right. Consequently, the residual, R, has a single non-zero entry, isolated to the cell flanking the front. Regardless of the size of the residual, there is only one nonzero entry, and so the residual is considered very sparse. At the current state, the Jacobian matrix, J, is lower bi-diagonal.

Figure 7.1(b) shows the action of the inverse of the Jacobian on the residual. The inverse of the Jacobian is lower triangular, and it can be easily shown that its entries decrease in magnitude away from the main diagonal. Consequently, the update is only non-zero in the cell flanking the front, and in the cells downstream of it. Moreover, the magnitude of the update can be shown to be largest in the frontal cell and decreases rapidly along the downstream direction. These observations motivate an approach to isolate non-zero entries in the right-hand-side and to independently inspect the level of locality of their propagation throughout the directed graph of the Jacobian matrix. Using this technique, it is possible to identify which cells will experience linear updates larger than a prescribed magnitude, prior to solving the system. As a result, only a smaller sub-system corresponding to these isolated cells need be solved.



(a) Schematic of the solution for a linearized update to a piston-like displacement over a timestep.



(b) The linearized update is local to the front and decays along the upwind direction.

Figure 7.1: An illustration demonstrating the locality of computed linear updates (continuation tangents or Newton steps) for a one-dimensional piston-like front.

In the hypothetical example of Figure 7.1(b), this reduced sub-system is indicated by the dotted box.

Using the principle of superposition on isolated non-zero entries, we can obtain the solution to problems with general right hand side vectors involving more than one nonzero entry. By linearity, the Newton step can be written as the sum of N sub-steps,

$\delta = \delta_1 + \ldots + \delta_N = J^{-1}R_1 + \ldots + J^{-1}R_N,$

where each sub-step is the Newton component that solves a material balance error originating from a single cell. That is, the residual component, R_j, is a vector with only one logically non-zero entry occurring in the j^{th} component. As already motivated, owing to the wave propagation character of multi-phase flow problems the residual vector is typically sparse. Subsequently, several residual components may be zero. The localization algorithm exploits this principle of superposition in order to estimate or directly obtain the Newton sub-steps corresponding to the non-zero residual components.

7.2.1 Scalar Conservation Laws

For the purpose of motivating our approach, we restrict our attention in this section to quasi-linear scalar conservation laws in one dimension (Equation 7.2.1). The flux function f is assumed to be differentiable with derivative f' and may be spatially dependent, and generally nonlinear; it may also have sonic points and is not necessarily convex.

$$u\left(x,t\right)_t + f\left(x,u\right)_x = 0. \tag{7.2.1}$$

We are interested in square nonlinear systems arising from implicit two-level approximations of the Godunov type (see for example [47]). A general form for the resulting residual is written as,

$$R^{(\nu)} \equiv \left[I + c_x\left(\mathbf{F}^R - \mathbf{F}^L\right)\right] U^{(\nu)} - U^n = 0, \tag{7.2.2}$$

where the mesh ratio $c_x = \Delta t/\Delta x \geq 0$ is fixed, and ν is the nonlinear iteration index.

Note in the definition of the residual above, we restrict the analysis to first-order upwind approximations, which can generally be written as,

$$\mathbf{F}_i^R = \begin{cases} \min\limits_{U_i \le U \le U_{i+1}} f_{i+1/2}(U) & U_i \le U_{i+1} \\ \max\limits_{U_{i+1} \le U \le U_i} f_{i+1/2}(U) & \text{otherwise} \end{cases}, i = 1, 2, \ldots N \tag{7.2.3}$$

Note that $\mathbf{F}_i^L = \mathbf{F}_{i-1}^R$. For a given nonlinear iteration, ν, we have at hand an iterate, $U^{(\nu)}$, and we compute a corresponding residual, $R^{(\nu)}$, and Jacobian, $J^{(\nu)}$. From now on iteration index superscripts will be dropped, and all terms are assumed to be at the ν^{th} nonlinear iteration of a timestep. In this notation, the corresponding Newton step is $\delta \equiv -J^{-1}R$.

By linearity, we can also write the Newton step as a combination of sub-steps, each resulting from the action of the inverse Jacobian on a single component of the residual. That is, let $R = R_1 + \ldots + R_N$ be such that,

$$R_j^{(i)} = \begin{cases} R^{(i)} & i = j \\ 0 & \text{otherwise} \end{cases}, 0 < i \le N,$$

where the superscript index, i, is the block index, and the subscript index, j, is that of the cell with a non-zero residual entry. Subsequently, the Newton step can be written as $\delta = \delta_1 + \ldots + \delta_N$, where $\delta_j \equiv -J^{-1}R_j$. Each Newton sub-step represents the state update over an iteration due to a nonzero residual entry in a single cell. We will next show that each of these sub-steps essentially propagates the isolated nonzero entry in the residual according to the local upwind direction and the local wave-speeds. More precisely, we derive a quantitative relation for the numerical range-of-influence due to a given nonzero residual entry. This is first described for problems without sonic points. We then generalize the scheme to counter-current flow problems involving sonic points. Finally we combine this information to formulate a quantitative relation for the total range of influence.

Flux functions without sonic points.

Given an iterate, we first suppose that the flux function has no extremal points over the cell-state range. In this case, the upwind directions will be unidirectional throughout the entire domain. If the flux function has nonnegative derivatives, $f' \geq 0$, the resulting flow is from left to right (upwind directions are to the left). Alternately, for $f' \leq 0$, the flow is from right to left. We derive in detail the main results assuming $f' \geq 0$, and state the corresponding result in the other case.

Denoting the local linearized CFL number as $\sigma_i \equiv c_x \left| f_i' \right|$, the i^{th} entry in the resulting Newton step is given by,

$$\delta^{(i)} = -\frac{1}{1+\sigma_i} \left(R^{(i)} - \sigma_{i-1} \delta^{(i-1)} \right), 0 < i \leq N.$$

The magnitude of the Newton step in a given cell depends on, at most, the residual values in the cells upstream of it. We can exploit this fact in order to quantitatively identify the numerical range of influence due to a nonzero residual entry in each cell independently.

For a given cell $0 < j \leq N$, we can write the corresponding Newton sub-step $\delta_j = -J^{-1} R_j$ as,

$$\delta_j^{(i)} = \begin{cases} 0 & 0 \leq i < j \\ -\zeta_j R^{(j)} & i = j \\ -\left(\prod_{k=j}^{i-1} \theta_k \right) \zeta_i R^{(j)} & j < i < N \end{cases}, 0 < i \leq N, \tag{7.2.4}$$

where we have the *local scaling constant*, ζ_i, written as,

$$\zeta_i \equiv \frac{1}{1+\sigma_i}, \tag{7.2.5}$$

and the *local attenuation ratios*, θ_k, given by,

$$\theta_k \equiv \frac{\sigma_k}{1+\sigma_k}. \tag{7.2.6}$$

The Newton sub-step does not modify the iterate state in cells upstream of the disturbed cell j. The cells downstream of j will be updated by a quantity that is a scaled version of the update at the disturbance source. The question we answer here, is how far out downstream into the domain are the effects of the disturbance felt?

Since $\sigma_i \geq 0$ for $0 < i \leq N$, both the local attenuation ratios, and the scaling constants are bounded above by one. Denoting the maximum and minimum CFL numbers over all cells downstream of j as $\sigma_{max,j}$ and $\sigma_{min,j}$, we derive bounds on the local scaling constants as,

$$0 < \zeta_i \leq \zeta_{\max,j} \equiv \frac{1}{1+\sigma_{\min,j}} \leq 1, \tag{7.2.7}$$

and on the local attenuation ratios as,

$$0 \leq \theta_i \leq \theta_{\max,j} \equiv \frac{\sigma_{\max,j}}{1+\sigma_{\max,j}} < 1. \tag{7.2.8}$$

We are interested in identifying the propagation rate through the Newton sub-step. Using the bounds in Equations 7.2.7 and 7.2.8, a conservative upper bound on the magnitude of the sub-step update in any cell downstream of the disturbance is given as,

$$\left|\delta_j^{(i)}\right| \leq (\theta_{\max,j})^{i-j}\, \zeta_{\max,j} \left|R^{(j)}\right|, j \leq i \leq N. \tag{7.2.9}$$

The alternate case, $f_i' \leq 0$, is treated similarly. For a given Newton sub-step due to a disturbance in cell j, we have that the magnitude of the update will be bounded as,

$$\left|\delta_j^{(i)}\right| \leq (\theta_{\max,j})^{j-i}\, \zeta_{\max,j} \left|R^{(j)}\right|, 1 \leq i \leq j. \tag{7.2.10}$$

Sonic points and counter-current flows.

More generally, the upwind direction may not be uniform throughout the domain. In particular, the flux function may map from extremal points over the range of state values in a given iterate. With single-point upwind schemes, this implies that at any

cell-face, the upwind flux of a particular phase may be evaluated at either the left-, right-, or sonic-point states. In the first two cases, the upwinding implies a coupling from the upstream cell to one or more cells downstream. In the case of a sonic point, numerically, no coupling occurs over the timestep; that is the zero speed characteristic of a Riemann fan is independent of the left and right states. Subsequently, it is easy to show that for general states and a given sub-step of interest, one of Equations 7.2.9, and 7.2.10 would hold.

Figure 7.2 depicts a mesh over which a Newton iterate has a varying upwind direction throughout the domain. Within the disjoint closed sub-domains labeled Ω_1, and Ω_4, the flow direction is positive. Within Ω_3, it is negative. Finally, the state within sub-domain Ω_2 is that of a Riemann fan over the timestep interval. Subsequently, a non-zero residual entry in some cell $j \in \Omega_1$ or $j \in \Omega_4$, can give rise to non-zero correction entries in the Newton sub-step, $\delta_j^{(i)}$, only within cells $\{i : i \in \Omega_{1,4}, i \geq j\}$. Within these cells, the precise magnitude of the corrections satisfy Equation 7.2.4. Moreover, the sub-step correction entries are bounded above as in Equation 7.2.9, where now, the maximum and minimum local CFL numbers are to be taken over the set of cells $\{i : i \in \Omega_{1,4}, i \geq j\}$. Similarly, for a non-zero residual entry in $j \in \Omega_3$, the effects can only be felt through $\{i : i \in \Omega_3, i \leq j\}$, and the bounds in Equation 7.2.10 hold. Finally, the effects of a non-zero residual in Ω_2 are isolated to that single cell.

Algebraic localization algorithm.

Having *a priori* conservative bounds that are easily computable, we can specify a desired absolute tolerance $\epsilon_{tol} > 0$, for which we wish to determine how far downstream of the disturbance will the magnitude of the Newton sub-step components exceed the tolerance. That is, we wish to determine an index $j \leq i_{range} \leq N$, such that for each $k \in \{j, \ldots, i_{range}\}$, we have $\left|\delta_j^{(l)}\right| \leq \epsilon_{tol}$. Indeed such a conservative bound is easily derived as,

$$\hat{i}_{range} \equiv \left\lceil \frac{\ln\left(\varepsilon_{tol}\right) - \ln\left(\zeta_{\max,j}\left|R^{(j)}\right|\right)}{\ln\left(\theta_{\max,j}\right)} \right\rceil + j,$$

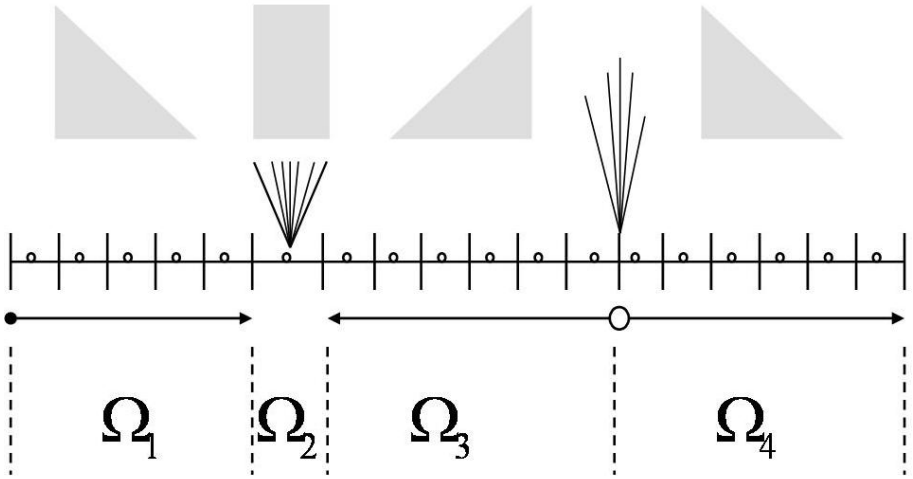


Figure 7.2: Sketch of a 1 dimensional mesh over which an iterate involves changing upwind directions.

$$i_{range,j} = \max\left(j, \min\left(\hat{i}_{range}, N\right)\right).$$

Algorithmically, the absolute error tolerance, $\epsilon_{tol} > 0$, can be set using the same criteria used in controlling inexact linear solvers. That is, we may apply all of the available theoretical machinery for relating the overall nonlinear convergence rate with the accuracy of computing Newton steps. What is unique about our range-of-influence approach is that even for very tight tolerances $\epsilon_{tol} \to 0$, the savings in computational effort can be substantial.

Finally, this analysis provides a strategy to localize computations for each nonlinear iteration. Given the iterate, we can simply determine the index sets $\mathcal{I}_{active,j} = \{i : j \leq i \leq i_{range,j}\}$ independently for each $j \in \{1, \ldots, N\}$. Cells indexed by a set will contain non-negligible Newton sub-steps. Cells indexed by the complement sets will have negligible Newton changes. The union of all active index sub-sets $\mathcal{I}_{active} = \bigcup_{j=1}^{N} \mathcal{I}_{active,j}$ flags the cells that must be updated. The reduction procedure is a standard column and row extraction from the Jacobian matrix and corresponding residual elements followed by a linear solve.

Computational Examples

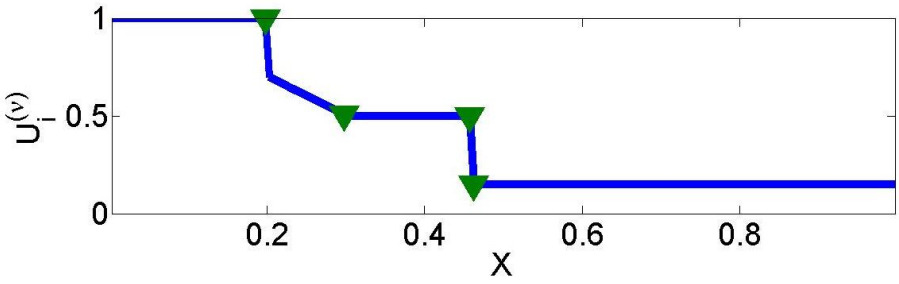


Figure 7.3: The imposed initial condition for a hypothetical downdip Buckley-Leverret problem. The triangular markers label components that produce nonzero entries in the residual vector.

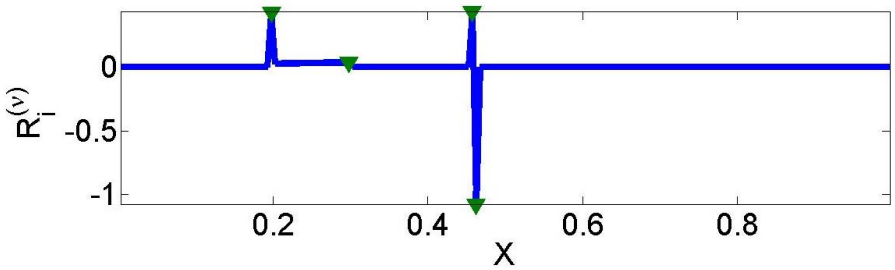


Figure 7.4: The residual obtained by using the imposed initial condition in Figure 7.3 as a first guess. The triangular markers show the four components that have the largest nonzero residual errors.

We consider as an example a Buckley-Leverret model with a downdip orientation. The fractional flow flux function is non-convex and has at least one sonic point, introducing the possibility for counter-current flow. Figure 7.3 shows a hypothetical initial state for a timestep. The old state itself is taken as a first guess for a Newton process. Figure 7.4 shows the residual vector obtained. The residual vector is rather sparse, and the largest four nonzero error components seem to sum up the information contained in the residual.

Using the localization algorithm developed, we obtain a number of localized newton steps and compare them to that obtained by solving the entire linear system. Figure 7.5 shows the results obtained using a relatively loose absolute error tolerance,

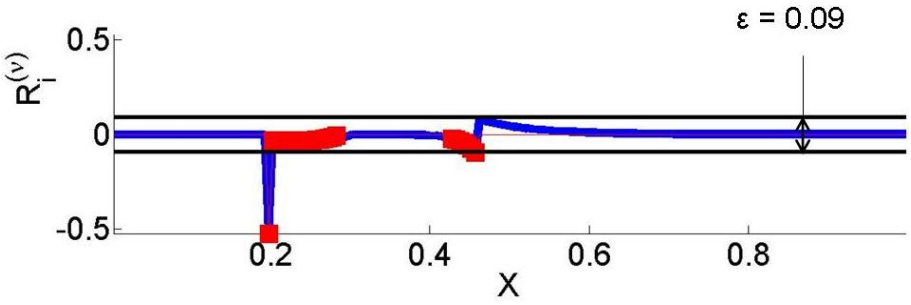


Figure 7.5: The blue solid line shows the Newton update obtained by solving the entire system. The red line with square markers shows the Newton update obtained by solving for only the flagged components. This is obtained for an absolute error tolerance of 0.09.

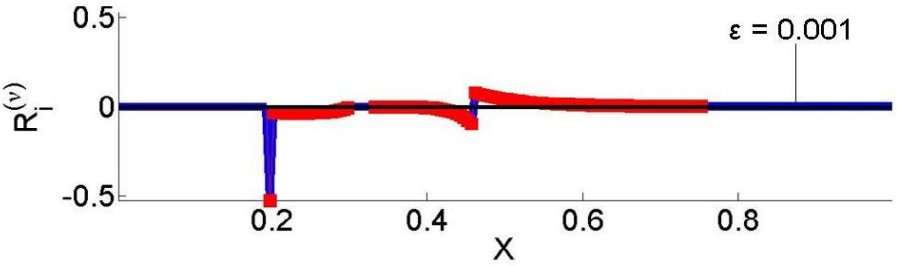


Figure 7.6: The solid blue line shows the Newton update obtained by solving the entire system. The red line with square markers shows the Newton update obtained by solving only the flagged components of the problem. This is obtained for an absolute error tolerance of 0.001.

$\epsilon = 0.09$. The localized solution involves less than a quarter of the domain, and produces an approximate Newton step that is conservatively more accurate. Figure 7.6 shows the result obtained for a tighter tolerance. About a third of the domain is flagged for a localized solution.

Figure 7.7 shows the range of efficacy of the localized flagging for a span of different absolute error tolerances. For very loose tolerances, as few as three components need be solved to meet the tolerance.

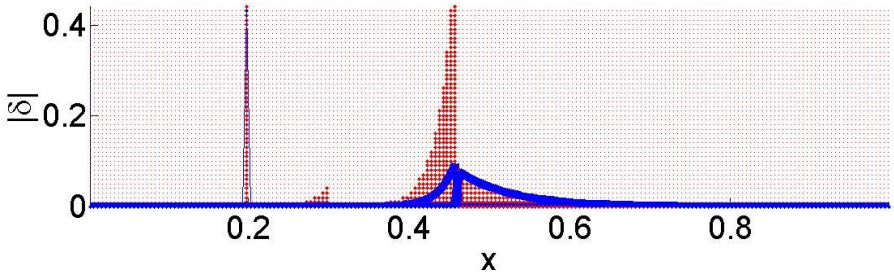


Figure 7.7: This figure presents the flagging results using a range of maximum absolute error criteria. In blue, is the absolute value of the Newton update that is obtained if the entire system were solved. Each of the red dashed level-lines demarks a flagging trial using the level's absolute error tolerance. The red stars on each level line show the flagged portion of the domain to be solved by the localization algorithm.

7.2.2 Multi-dimensional problems

Owing to the local nature of saturation and concentration waves, the individual Newton sub-steps can be obtained or estimated without solving the entire system. This is because each sub-step has a right-hand-side with one non-zero entry. We developed this process for hyperbolic conservation laws in one-dimension. The extension of the sub-step solution process extends to multi-dimensional problems on general meshes. Figure 7.8 illustrates this. The application of the principle of superposition, where we treat non-zero residual entries independently, holds. The entries in the Jacobian matrix form a weighted directed graph through the mesh. By generalizing the one-dimensional relations already derived, we can apply the same procedure to identify cells with non-zero updates. In particular, for problems where there are no directed cycles within the upwind graph, this is achieved by first performing a breadth-first traversal originating at the cell with the non-zero material balance error. For an introduction to the breadth-first-traversal graph algorithm, see for example [21]. By a second traversal limited to the cells that were already visited, the update components are computed sequentially. This is the case for all scalar problems in saturation, as well as for multi-phase problems without counter-current flow ([46]).

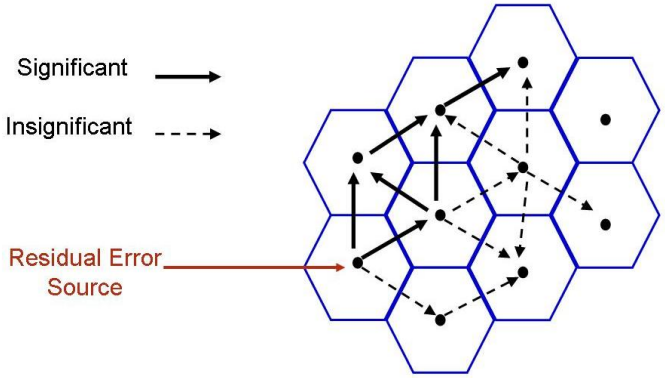


Figure 7.8: A single non-zero entry in the residual can be tracked through the directed upwind graph in order to determine its range of influence on the corrective linearized update.

7.3 General Coupled Systems

General coupled systems may easily introduce one or more directed cycles in the upwind graph. The consequence of this is that the system cannot in essence be triangularized or solved sequentially. Heuristics must be employed to estimate the range-of-influence. This is the case for problems with counter current flow and with compressibility. Instead of using a local analytical inversion procedure, in such cases, the heuristic we employ is to systematically disconnect edges that cause cycles. The impact of this is not expected to be severe, since the subsequent iteration can be shown to spread the material balance error onto a larger proportion of the domain, requiring a complete system solve due to these global interactions.

7.4 Summary

We establish an algorithm that solves the large, coupled linear systems that arise within nonlinear iterations for the solution of an implicit timestep of a hyperbolic problem. The algorithm uses the sparsity structure of the residual and the directed graph of the Jacobian matrix in order to solve the whole problem by directly solving a smaller one. The size of the smaller system that is solved primarily depends on

the local wave speeds in the problem, and computation is typically localized to the vicinity of traveling waves. The size of the localized system also depends on the level of accuracy that is sought. Extensions to hyperbolic systems and to coupled parabolic-hyperbolic models are suggested. More work is needed to establish the viability of this approach for large problems involving complex geology and multiple wells.

Chapter 8

A Continuation-Newton Algorithm On Timestep Size.

Implicit/coupled methods offer unconditional stability in the sense of discrete approximations. In these methods several or all primary unknowns are treated implicitly giving rise to a nonlinear coupled system of discrete residual equations. These coupled nonlinear systems must be solved at each timestep of a simulation in order to obtain the timestep's state solution. The unconditional stability of these methods comes at the expense of transferring the inherent physical stiffness onto the coupled nonlinear residual equations which need to be solved at each timestep. Current reservoir simulators apply safe-guarded variants of Newton's method. These solvers often can neither guarantee convergence, nor provide estimates of the relation between convergence rate and time-step size. In practice, time-step chops become necessary, and are guided heuristically. With growing complexity, such as in thermally reactive compositional models, this can lead to substantial losses in computational effort, and prohibitively small timesteps.

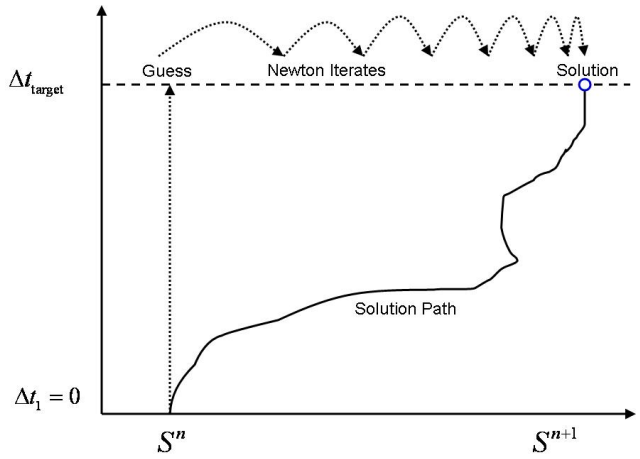
This work devises an algorithm to address these shortcomings of using Newton-like methods. Moreover, the approach introduced in this work embeds the process of timestep selection directly in to the nonlinear iterative solution process. More specifically, the idea is to devise an iterative solution process for which each iterate is a solution to a timestep that is smaller than the target step. The implications of

this are that (1) convergence is always guaranteed, (2) timestep chops do not involve wasted computation, and (3) timestep selection for convergence considerations is no longer necessary.

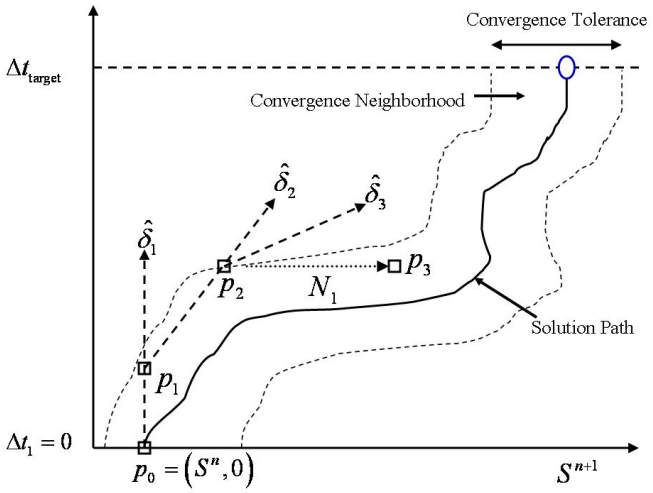
8.1 Associating a timestep size with iterates

To motivate our approach, we consider the nature of solutions to reservoir simulation timesteps. Well-posed reservoir simulation residuals have a unique solution for each timestep. The solution only needs to be unique within a prescribed domain; for example in a cell, saturations are bounded by zero and one, pressure is positive, etc. Let $U_0 \in \mathbb{R}^N$ denote such a discrete solution state vector for a particular time. Then, independently for every possible timestep from this state, $\Delta t \geq 0$, there corresponds a solution $U \in \mathbb{R}^N$ that satisfies $R\left(U, \Delta t; U_0\right) = 0$, where R is the vector of discrete residual equations. All such pairs of solution and corresponding timestep, $(U, \Delta t)$, can be interpreted as points defining a curve in an $N+1$-dimensional space, where N is the number of residual equations, and the additional dimension is the timestep size, Δt. The existence and uniqueness of these points can be shown to imply that they form a continuous curve, emanating from the initial point with a timestep size of zero, $(U_0, 0)$. Furthermore, this curve has a strictly positive gradient along the timestep dimension, since otherwise, uniqueness would be violated. The curve of points satisfying these requirements for a particular initial state, U_0, is called the *solution path* emanating from that state.

For illustration, consider the solution path of a nonlinear problem in a single state unknown, that is $R\left(S^{n+1}; \Delta t_{target}, S^n\right)$. Since there is only one state unknown we may plot the solution path in the two-dimensions, S^{n+1} and Δt. In a typical reservoir simulation problem, there may be millions of unknowns, and each would be represented by a dimension. Figure 8.1(a) illustrates a solution path to this hypothetical problem. The solution path emanates from the initial point $(S^{n1} = S^n, \Delta t = 0)$, and continues to the target time step, Δt_{target}, augmented with its solution, S^{n+1}, in this two-dimensional space. Notice that in this illustration, the solution path always moves forward in timestep size and never folds on itself. Each point on the path is a



(a) Newton's method illustrated on a solution path. All iterates are evaluated at the target timestep, starting from the old state as an initial guess.



(b) Illustration of three iterates in the Continuation-Newton algorithm. The first two iterates are chosen along tangent vectors. The third tangent leads to points outside the convergence tolerance. A safeguarded Newton step is performed.

Figure 8.1: Solution path diagrams and solution methods depicted for a problem with a single time-dependent unknown S. The old state, for a zero timestep, $\Delta t = 0$, is S^k, and the solution at the target timestep, Δt_{target}, is S^{k+1}.

pair of a solution state and timestep size.

Also shown on Figure 8.1(a) are the standard Newton steps (curved dotted lines), which attempt to find the solution at the target time step. With a classic Newton process, the old state is projected to the target time $t_n + \Delta t_{target}$ as an initial guess. A sequence of iterates is obtained, all evaluated using the target timestep size. A convergent Newton iteration provides a final iterate that is close enough to the solution path at the target timestep size. Should too many iterations go by without convergence, all iterates are wasted, and the challenging task of selecting a smaller timestep must be addressed.

8.2 The Continuation-Newton (CN) algorithm.

In order to overcome this challenge, we design a Numerical Continuation algorithm ([3, 2, 74, 61]) that proceeds from the initial state from which the solution path emanates. The process generates a sequence of iterates along the path in the augmented $N+1$-dimensional space, climbing towards the target timestep size. Subsequently, each iterate would be related to a known, smaller timestep. Should too many iterations go by before the target timestep is attained, the final iterate is associated with a solution for a smaller known timestep, from which we may continue the simulation.

8.2.1 High level outline of CN

We develop an alternative iteration to Newton's method in which the iterates are pairs of the unknown state, augmented with a corresponding timestep. The iteration starts from an initial point consisting of the initial state and a timestep of zero. The intent of the iteration sequence is to follow along the solution path toward the solution of the target timestep size. Since we are interested in the solution for a target timestep, we hope to avoid following the solution path (in the $N+1$-dimensional space) too closely, since that will be computationally wasteful. For this reason, we develop a *convergence neighborhood* about the solution path so that points within the neighborhood are deemed close enough to the solution path at their timestep level.

The iterates can follow the path more loosely without sacrificing their proximity to the solution for their timestep component. By parameterizing the solution path with a single parameter along its tangential coordinates, we develop an ability to compute the tangent to the solution path at any point in the augmented space. These tangent computations are developed in detail in the next section.

Here, we outline the Continuation-Newton process using the single unknown hypothetical example discussed earlier, namely, $R\left(S^{n+1}; \Delta t_{target}, S^n\right)$.

Figure 8.1(b) illustrates the proposed algorithm pictorially on the hypothetical solution path similar to that in Figure 8.1(a). The solution path emanates from the initial point $p_0 = (S^n, 0)$. We want to find the solution for a target timestep Δt_{target}. The proposed algorithm defines a convergence neighborhood about the solution path, so that points, $p_{int} = (S_{int}, \Delta t_{int})$, inside the neighborhood are deemed to be close-enough solutions for their timestep component, Δt_{int}. That is, S_{int} is either a solution to a timestep of Δt_{int}, or it is a good starting guess for a standard Newton iteration starting from S_{int} for a timestep of Δt_{int}. Starting from the initial point, p_0, from which the solution path emanates, we can compute the $N + 1$-dimensional tangent vector, $\hat{\delta}_1$. An appropriate step-length along the tangent vector is selected such that the next iterate remains within the convergence neighborhood. In the figure, this iterate is denoted p_1. The tangent update is linear, $p_1 = p_0 + \alpha\hat{\delta}_1$, where $\alpha \geq 0$ is called the *tangent step-length*. Note that since tangents, $\hat{\delta}_i$, and all iterates, p_i, are augmented variables in both the state and the timestep size, these iterations evolve timestep as well as the solution. Similarly, we may repeat this process to obtain the next point p_2. At this point, which is near the boundary of the convergence neighborhood, we find that the next tangent step-length, which keeps the next iterate within the convergence neighborhood, is too small or zero. In this case, the Continuation Newton (CN) algorithm applies a *Newton Correction* step, N_1, as illustrated in Figure 8.1(b). This Newton correction is evaluated at the current timestep size, and brings the iterates closer to the solution path at point p_3. The iterate, $p3$, is guaranteed to be close to the solution path because the convergence neighborhood is chosen so that it is contained within the contraction region of Newton's method. We can continue with tangent steps thereon.

By construction, the sequence of iterates, p_i, are all close-enough solutions to their associated timesteps. The guarantee of convergence for all times is also supported by the choice of the convergence neighborhood. As a result, any necessary corrective Newton steps, N_j, always contract toward the solution path. Since the initial guess to such corrective steps is already a close-enough solution by construction, a single Newton step can be guaranteed to reduce the residual.

In practice, the CN process can be continued until either the target timestep size is attained, or the maximal number of iterations allowed has been expended. In the latter case, the final CN iterate is a close-enough solution to its corresponding timestep size. We can accept this solution and its timestep size, and continue onto the next simulation timestep. Subsequently, no timestep chop selection is ever necessary, and no computational effort is wasted.

Algorithm 4 prescribes the general CN process to solve a target timestep Δt_{target} given an old state U^n, which may involve many unknowns.

Algorithm 4 uses several sub-algorithms, some of which have already been discussed earlier; the Appleyard safe update performs a cell-wise saturation update along the tangent direction, and the Modified Appleyard Newton corrector and solver performs a Newton process using the Modified Appleyard heuristic. In Algorithm 4, local Newton corrections to bring points closer to the solution path are performed by a Newton solver, except that a different looser convergence tolerance is used. Specifically, the corrector convergence tolerance only needs to be smaller than, or equal to, the convergence neighborhood tolerance. The remaining sub-algorithms are SOLVE-TANGENT, which computes the normalized tangent update vector, $\hat{\delta}$, SELECT-TANGENT-STEP-LENGTH, which performs a search along the tangent for the largest step which remains within the convergence neighborhood, and IS-WITHIN-NEIGHBORHOOD, which tests whether a point is inside the convergence neighborhood. These three sub-algorithms are developed next.

Next, we develop the methods to compute tangent updates and to select step-lengths that lie within a convergence neighborhood.

Algorithm 4 CONTINUATION-NEWTON-STEP($U^n, \Delta t_{target}$)

Require: $\Delta t_{target} \geq 0$ and $U^n \in \mathbb{R}^N$
Ensure: $F\left(U, \Delta t; U^n\right) = 0$, $0 < \Delta t \leq \Delta t_{target}$, and, niter $< N_{max}$
  niter $\leftarrow 0$
  $\Delta t \leftarrow 0$
  $U \leftarrow U^n$
  while niter $< N_{max}$, and, $\Delta t < \Delta t_{target}$ **do**
    $\hat{\delta} \leftarrow$ COMPUTE-TANGENT$(U, \Delta t, U^n)$
    $\alpha \leftarrow$ SELECT-TANGENT-STEP-LENGTH$\left(U, \Delta t, \hat{\delta}, U^n\right)$
    $W \leftarrow U$
    for $i = 0$ to $i = N$ **do**
      $W\left(i\right) \leftarrow$ APPLEYARD-SAFE-UPDATE$\left(W\left(i\right), \alpha\hat{\delta}\left(i\right)\right)$
    end for
    $\Delta t_2 \leftarrow \alpha\hat{\delta}\left(N+1\right)$
    niter $\leftarrow$ niter $+1$
    if IS-WITHIN-NEIGHBORHOOD$(W, \Delta t_2, U^n)$ **then**
      $U \leftarrow W$
      $\Delta t \leftarrow \Delta t_2$
    else /* *No tangent step-length is admissible in convergence neighborhood* */
      $U, N_{newton} \leftarrow$ MODIFIED-APPLEYARD-NEWTON-CORRECTOR$(U, \Delta t; U^n)$
      niter $\leftarrow$ niter $+ N_{newton}$
    end if
  end while
  $U, N_{newton} \leftarrow$ MODIFIED-APPLEYARD-NEWTON-SOLVER$(U, \Delta t; U^n)$
  niter $\leftarrow$ niter $+ N_{newton}$
  return U, Δt, and, niter

8.2.2 Parameterizing and following the solution path.

With the assumption that the N-dimensional residual equations have a unique solution for every timestep, we can parameterize the solution and timestep pairs along a single scalar $\lambda \geq 0$. That is, the N-dimensional state solution vector and corresponding timestep form an $N+1$-dimensional space, and the components are considered as functions of a single scalar, λ, so that any point can be written as $p(\lambda) = (U(\lambda), \Delta t(\lambda))$. We want the solution path in the $(U, \Delta t)$ space to emanate from the old state of a given timestep. So for $\lambda = 0$, we have the point $p_o = (U_0, 0)$, where U_0 is the initial state vector of the timestep. We write the residual equations in terms of this parametrization as,

$$R(U(\lambda), \Delta t(\lambda); U^n) = 0. \tag{8.2.1}$$

The total derivative of the residual vector with respect to the scalar λ prescribes the tangent to level-curves in the residual value. Since the particular level-curve of interest is the zero residual level curve (the solution path), we can write,

$$0 = \frac{dR}{d\lambda} = J\frac{dU}{d\lambda} + \frac{\partial R}{\partial \Delta t}\frac{d\Delta t}{d\lambda}, \tag{8.2.2}$$

where, J is the same $N \times N$ Jacobian matrix that would have been used in a Newton method, except that it is now interpreted as a function of timestep as well as state. That is, while within a Newton iteration, the Jacobian is always evaluated using the target timestep, here, the Jacobian may be evaluated using any other timestep size. The Jacobian's elements are defined as $J_{(i,j)} = \frac{dF_{(i)}}{dU_{(j)}}$, and the N-dimensional vector $\frac{dR}{d\Delta t}$ in Equation 8.2.2 is the derivative of the residual vector with respect to timestep.

Since the initial point on the zero level-curve is known (e.g. the solution from the previous timestep), the initial condition for the system is well-defined. Combining this initial condition with Equation 8.2.2, we obtain,

$$\begin{cases} J\left(U,\Delta t;U_0\right)\frac{dU}{d\lambda}+\frac{d}{d\Delta t}R\left(U,\Delta t;U_0\right)\frac{d\Delta t}{d\lambda}=0 \\ U(\lambda=0)=U_0 \\ \Delta t(\lambda=0)=0 \end{cases} . \tag{8.2.3}$$

Equation 8.2.3 describes the dynamics of motion along the solution path in $(U,\Delta t)$ space emanating from the initial point $(U_0,0)$. In this context, the parameter $\lambda\geq 0$ is the arc-length along the solution path in its tangential coordinates. Geometrically, the $(N+1)$-dimensional vector $\delta=\left(\frac{dU}{d\lambda},\frac{d\Delta t}{d\lambda}\right)$ is the tangent to the residual level curves. When the tangent is evaluated at a point on the solution path, which is the zero residual level curve, it is a tangent to the solution path itself.

The solution of a nonlinear residual for any timestep can be computed by numerically integrating the corresponding initial value problem given by Equation 8.2.3 up to $\lambda=\lambda^*$ for which the timestep is equal to the target value; i.e. $\Delta t\left(\lambda^*\right)=\Delta t_{target}$. This, however, is not the objective. We are interested in reaching the solution for the target time step as quickly as possible. Consequently, we should follow the solution path in $(U,\Delta t)$ space only loosely and with as little computational effort as possible.

8.2.3 Computing the tangent to any point on a solution path.

The key computational expense of a CN iteration is in evaluating the tangent vectors to the solution path. The cost is precisely that of performing a single Newton iteration, and involves the solution of the Jacobian matrix. In the previous section, we derived a mathematical parameterization of solution paths along a single tangential coordinate, $\lambda\geq 0$. By substituting this parametrization into the residual, and requiring a zero residual along the curve, we derived the equations of motion that define the curve. Yet, Problem 8.2.3 is under-determined; we have N equations for $N+1$ unknowns. The additional unknown is the timestep size. Thus, an additional equation is necessary to close the system. Without further insight into the problem, we could choose one of several alternatives to close the system, as long as the resulting problem

is consistent. One example is to require that the norm of the tangent be unity (e.g. [3, 2, 61]); that is, $\left\| \left(\frac{dU}{d\lambda}, \frac{d\Delta t}{d\lambda} \right) \right\| = 1$. Another is to enforce some sort of inequality constraint on some elements of the tangent. Not only are such alternatives clearly computationally undesirable, but they also do not exploit any inherent properties of the physics. In this particular setting, the additional equation could be chosen to say something about the rate of change in timestep size with respect to arc-length along the solution path. Something certain about solution paths to well-posed problems is that $\frac{d\Delta t}{d\lambda} > 0$. This is the case, since otherwise uniqueness is violated.

Enforcing this condition algebraically is quite simple. We require that $\frac{d\Delta t}{d\lambda} = C > 0$, where C is any positive constant. To compute the $N+1$-dimensional tangent vector, we solve,

$$\begin{bmatrix} & \\ J & \frac{dR}{d\Delta t} \\ & \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \\ \frac{dU}{d\lambda} \\ \\ \frac{d\Delta t}{d\lambda} \end{bmatrix} = \begin{bmatrix} \\ 0 \\ \\ C \end{bmatrix}, \tag{8.2.4}$$

where the Jacobian, J, is an N by N matrix, the timestep derivative of the residual, $\frac{dR}{d\Delta t}$, is an N-dimensional vector, and the tangent component along timestep size, $\frac{d\Delta t}{d\lambda}$, is a scalar. Notice that the $(N+1) \times (N+1)$ system in Equation 8.2.4 is always solvable, since the Jacobian itself is assumed to be always solvable. The Jacobian is well conditioned in the vicinity of the solution path. This is because all solution points are stable fixed points of the Newton Flow for a fixed timestep, and moreover, we already know that the solution to the timestep size tangent component, $\frac{d\Delta t}{d\lambda}$, is $C > 0$.

The positive constant, C, is chosen arbitrarily. The specific choice is inconsequential, since we have the unique, consistent direction of the tangent vector, and we can select any length along it. To be precise, we can normalize the computed tangent, $\delta = \left(\frac{dU}{d\lambda}, \frac{d\Delta t}{d\lambda} \right)$, as follows;

$$\begin{aligned} \delta &\leftarrow C.\left(-J^{-1}\tfrac{dR}{d\Delta t}, 1\right)^T, \\ \hat{\delta} &\leftarrow \tfrac{\delta}{\|\delta\|}. \end{aligned} \tag{8.2.5}$$

This can always be done because C is positive, and so the tangent norm is bounded away from zero. This normalization is valid even when the solution is at a steady state. Equation 8.2.5 provides a computationally attractive way to compute level-curve tangents at points near the solution path. At such points, the Jacobian is numerically well-conditioned, since it is evaluated in the vicinity of a unique solution, and it has the same structure as standard reservoir simulation Jacobian matrices. Algorithm 5 prescribes the details of computing the tangent vector at a point $(U, \Delta t)$ close to the solution path emanating from the point $(U^n, 0)$.

Algorithm 5 COMPUTE-TANGENT$(U, \Delta t, U^n)$

Require: $\Delta t \geq 0$ is a timestep size for state $U \in \mathbb{R}^N$ from the initial state $U^n \in \mathbb{R}^N$

Ensure: $\hat{\delta} \in \mathbb{R}^{N+1}$ is the unit tangent to the augmented solution curve emanating from $(U^n, 0)$ at the point $(U^n, \Delta t)$

Fix a positive constant C. The choice $C = \sqrt{\epsilon_{mach}}$, where ϵ_{mach} is the machine precision, can be used for numerical conditioning.

$C \leftarrow \max\left(C, \frac{1}{\left\|\frac{\partial}{\partial \Delta t} R(U;\Delta t, U^n)\right\|}\right)$

$\delta_u \leftarrow -C J^{-1} \frac{\partial}{\partial \Delta t} R\left(U; \Delta t, U^n\right)$

$\delta_{\Delta t} \leftarrow C$

$\delta \leftarrow (\delta_u, \delta_{\Delta t})^T$

$\hat{\delta} \leftarrow \frac{1}{\|\delta\|}\delta$

return $\hat{\delta}$

Note that the value of the constant C is chosen as the inverse of the norm of the timestep derivative of the residual system. This improves the numerical scaling. The Jacobian matrix, J, need not be inverted explicitly, and any choice of efficient reservoir simulation linear solver can be used. Here we focus on the nonlinear convergence aspects, and we apply a direct sparse solver with partial pivoting (see [48]).

8.2.4 Defining a convergence neighborhood.

A key component of the CN algorithm is having a quantitative measure of proximity to the solution path in the augmented space. A tight measure results in iterates that are accurate solutions, but may result in poor computational efficiency as the solution path in the $(U, \Delta t)$ space is needlessly traced in detail. On the other hand, a loose tolerance may produce iterates that are farther away from the solution, requiring more Newton corrective steps. Accordingly, one objective is to select this proximity measure so that points within the neighborhood around the $(U, \Delta t)$ solution path provide good starting points for a Newton corrective process for their corresponding timestep. A second objective is to select this neighborhood so that there is a computationally inexpensive procedure to test whether a given point is within it or not.

We denote the solution path emanating from $p_0 = (U_0, 0) \in \mathbb{R}^{N+1}$ as the set of points $\mathcal{C} = \{p(\lambda) = (U(\lambda), \Delta t(\lambda)) : R(U(\lambda), \Delta t(\lambda); U_0) = 0, \lambda \geq 0\}$. Three measures and their corresponding convergence neighborhoods, $\mathcal{N}$, maybe defined as follows:

- **Residual or material balance norm.** A point is in the neighborhood provided that its residual, or material balance norm, is less than a specified tolerance; $p \in \mathcal{N}$ if and only if $\|R(p)\| \leq \epsilon_{tol}$.

- **Absolute or maximum change tolerance estimates.** At any point $(U, \Delta t)$, the norm of the first Newton step, $\|-J^{-1}R(U, \Delta t; U^n)\|$, toward the solution path can be related to the norm of the absolute error. In particular, it is a linearized estimate that becomes more accurate as a measure of absolute error within the vicinity of the solution path.

- **Jacobian matrix norm or curvature estimates.** Within the convergence basin of the Newton Flow, the Jacobian matrix induces a Lipschitz property on the residual. That is, within the neighborhood of the solution, the curvature of the residual is bounded. Using this property, a computational estimate of a Kantorovich condition on the Jacobian matrix norm can be used to estimate whether the point in question is a good point to start a Newton iteration (see

for example [25, 52]).

Residual based measures require the computation of the residual only, and they are accurate measures of proximity if one is close to the solution. Residual measures alone, however, do not necessarily guarantee favorable Newton Convergence properties. Jacobian matrix norm measures invariably require computing the Jacobian and provide estimates of the local convergence properties of a Newton iteration. They do not measure proximity directly, however. Absolute error estimates are the most computationally expensive requiring the computation of a Newton step. Absolute error measures can combine more accurate measures of both proximity to the solution, as well as of convergence rate properties. A combination of these three broad types of measures may be used to ensure robustness, while trading off computational performance. Note that the most strict criterion is to use a residual tolerance so that every point in the neighborhood of the augmented solution path is itself an acceptable solution for the residual equations. While such a strict criterion will always generate actual solution iterates, it may lead to needlessly following the solution path too closely, and that can increase the number of continuation steps per target timestep. In this work, we apply a residual tolerance criterion only. Algorithm 6 describes the details of testing whether a given point $(U, \Delta t)$ is within the convergence neighborhood about the solution path emanating from $(U_0, 0)$. The choice of residual tolerance cut-off is subjective. Accepting the locally quadratic convergence rate of Newton's method, we use a tolerance that is one or two orders of magnitude looser than that required to judge if an iterate is a valid solution.

8.2.5 Step-length selection along tangents.

Another component of the CN algorithm is the selection of an appropriate step-length along a tangent vector. The CN update is written as,

$$\begin{pmatrix} U \\ \Delta t \end{pmatrix}_{k+1} \leftarrow \begin{pmatrix} U \\ \Delta t \end{pmatrix}_{k} + \alpha \begin{pmatrix} \hat{\delta}_U \\ \hat{\delta}_{\Delta t} \end{pmatrix}, \tag{8.2.6}$$

Algorithm 6 IS-WITHIN-NEIGHBORHOOD$(U, \Delta t, U_0)$

Require: $\Delta t \geq 0$ is a timestep size for state $U \in \mathbb{R}^N$ from the initial state $U_0 \in \mathbb{R}^N$

Ensure: Returns whether U is within the convergence neighborhood at timestep Δt from state U_0

Fix a positive tolerance ϵ_{cnrtol}. A typical choice may be $\epsilon_{cnrtol} = \sqrt{\epsilon_{rtol}}$, where ϵ_{rtol} is a residual tolerance used for a Newton correction processes.

if $\|R(U, \Delta t; U_0)\| < \epsilon_{cnrtol}$ **then**
  return true
else
  return false
end if

where α is the positive step-length.

A larger step-length results in a larger timestep advancement, since the timestep component of the tangent, $\hat{\delta}_{\Delta t}$, is always positive. This implies a higher computational efficiency as the timestep advancement per tangent computation increases. On the other hand, a larger step-length also implies a larger absolute error away from the solution path. Using continuity around solutions, theory guarantees that for small step-lengths from points on the solution curve, any residual tolerance may be satisfied. In practice, CN iterates are not exactly on the solution path, however, and for a given tangent, there may be no positive step-length that results in an acceptable point that lies within the convergence neighborhood; $p_{k+1} \in \mathcal{N}$. In such cases, the step-length selection algorithm must trigger Newton correction steps. This switch to Newton corrections is dictated by the nature of the physics being modeled. For example, slowly transient solutions can be stepped through with larger tangent step lengths, while faster transients may require shorter tangent steps.

In practice, a minimal step-length, α_{min}, is chosen as a parameter. Note that from Equation 8.2.6, for a step-length α, the corresponding updated timestep size is simply $\Delta t^k + \alpha\hat{\delta}_{\Delta t}$. We can choose α_{min} to satisfy a minimal timestep advancement per tangent step, Δt_{min}.

There may be many step-lengths that are larger than the minimum, $\alpha \geq \alpha_{min}$, that keep the new iterate within the convergence neighborhood. In this work, the

objective of the step-length algorithm is to select the largest such step-length in order to maximize the timestep advancement achieved per continuation step. Since constrained univariate optimization is generally more efficient than an unconstrained counter-part, we also set a maximal step-length parameter, α_{max}. Choices for the maximal step-length parameter, α_{max}, can be made in several ways. One approach could be to choose the smallest step-length that makes all updated normalized variables reach their physical range. Another approach, which is used throughout the examples in this work, is to select it so that the tangent timestep update satisfies a maximal timestep advancement per tangent step, Δt_{max}.

The univariate optimization problem for the tangent step-length can be expressed as,

$$\begin{cases} \text{maximize} & \alpha \\ \text{subject to,} & \\ & p_k + \alpha\hat{\delta}^k \in \mathcal{N}, \\ & \alpha \in [\alpha_{\min}, \alpha_{\max}] \end{cases} \tag{8.2.7}$$

In the examples in this work, we apply a derivative free backtracking algorithm to solve this problem. The algorithm used to solve the univariate problem is derivative free. Each iteration simply requires the evaluation of the residual. We apply a backtracking approach that starts by evaluating whether α_{min} results in an update which lies within the convergence neighborhood. If this is not the case, the search is terminated, and Newton corrections are triggered. Otherwise, backtracking from α_{max}, we search for the first step-length that produces an update within the neighborhood. The maximum number of backtracking steps is a parameter of the algorithm. Increasing this quantity may result in a solution with fewer tangent steps at the expense of more residual evaluations performed per tangent step. Algorithm 7 describes the details of this process.

Algorithm 7 SELECT-TANGENT-STEP-LENGTH$\left(U, \Delta t, \hat{\delta}, U^n\right)$

Fix a maximum number of backtracking iterations; $Iter_{max} \geq 1$.
Fix a minimal timestep size advancement per tangent step, $\Delta t_{min} > 0$. This can be based on a suitable CFL number such as one.
Fix a maximal timestep size advancement per tangent step, $\Delta t_{max} > \Delta t_{min}$. A typical CFL number may be ten.
$\alpha_{min} \leftarrow \frac{\Delta t_{min}}{\hat{\delta}(N+1)}$
$\alpha_{max} \leftarrow \frac{\Delta t_{max}}{\hat{\delta}(N+1)}$
$\Delta \leftarrow \frac{\alpha_{max} - \alpha_{min}}{Iter_{max}}$
$\alpha \leftarrow \alpha_{max}$
do
  for $i = 0$ to N **do**
    $U^*(i) \leftarrow$ APPLEYARD-SAFE-UPDATE$\left(U(i), \alpha\hat{\delta}(i)\right)$
  end for
  $\Delta t^* \leftarrow \Delta t + \alpha\hat{\delta}(N+1)$
  is-converged $\leftarrow$ IS-WITHIN-NEIGHBORHOOD$(U^*, \Delta t^*, U^n)$
  $\alpha \leftarrow \alpha - \Delta$
until $\alpha < \alpha_{min} - \Delta$ or is-converged = **true**
return α

8.3 Computational Examples

8.3.1 Single-cell model

For problems with a single unknown, the Continuation-Newton augmented space is two-dimensional. In order to visualize the solution process, we consider a model problem that is designed after a single cell view of incompressible two phase flow through a horizontal domain. This enables us to visualize solution processes in a three-dimensional space formed by the primary unknown dimension, timestep size, and the residual value dimension. Figure 8.2 illustrates the setup of this model problem.

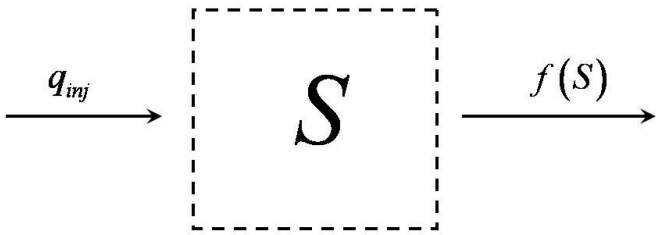


Figure 8.2: Illustration of a model problem designed after a single cell view of two-phase incompressible flow.

Let $S(t)$ denote the average time-dependent wetting-phase saturation in a fixed volume of porous media at time t. Equation 8.3.1 is the governing Ordinary Differential Equation (ODE) for flow through the media assuming a fixed influx q_{inj}.

$$\begin{cases} \frac{dS}{dt} = q_{inj} - f(S) \\ S(0) = S_{init} \end{cases} \tag{8.3.1}$$

We denote the initial saturation as S_{init}, and the fractional flow function f is defined as,

$$f(S) = S^2 \frac{1 - Ng(1-S)^2}{S^2 + M_r^0 (1-S)^2}, \tag{8.3.2}$$

where the Gravity Number, Ng, and the end-point Mobility Ratio, M_r^0, are constants.

Let S^n denote the numerical approximation of the wetting-phase saturation in the cell at time-step n. The time at the n^{th} time-step is $t = n\Delta t$, where $\Delta t > 0$ denotes a fixed time-step size. Equation 8.3.3 is the discrete implicit residual for this problem.

$$\begin{cases} S^{n+1} - S^n + \Delta t \left(f\left(S^{n+1}\right) - q_{inj} \right) \\ \\ S^0 = S_{init} \end{cases} \tag{8.3.3}$$

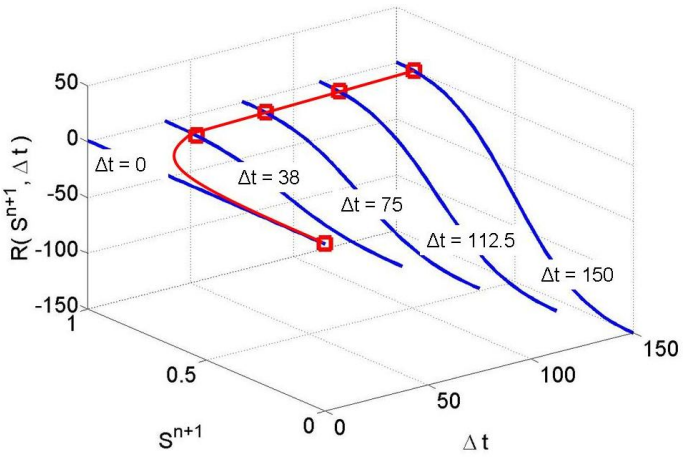


Figure 8.3: The residual curves of a single cell problem evaluated for various timestep sizes (solid blue lines). Each residual curves intersects the zero residual plane at one solution point (red square markers). The locus of all solution points forms the Solution Path (solid red line).

At each time-step of a simulation, Equation 8.3.3 must be solved for S^{n+1}, given S^n. We consider the specific case of horizontal flow, $Ng = 0$, through a domain with a zero initial saturation, $S_{init} = 0$, a constant injection flux, $q_inj = 1$, and an end-point mobility ratio of $M_r^0 = 10$. Figure 8.3 shows plots of the nonlinear residual curves that are obtained by evaluating the residual (8.3.3) using five different time-step sizes. For a timestep size of zero, the residual curve is linear as it is simply the accumulation component. For larger timestep sizes, the residual curve is essentially a re-scaled form of the non-monotone flux function. In the figure, each residual curve

intersects a zero residual level at a unique solution point that is marked with a red square marker. The locus of all solution points for every possible timestep size is the Solution Path, and in the figure it is depicted by the solid red line.

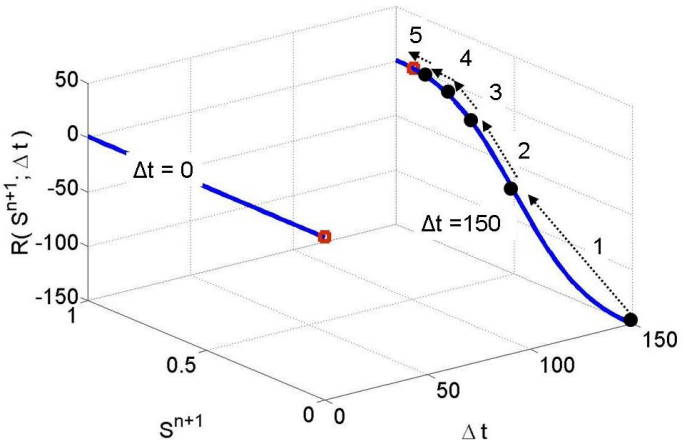


Figure 8.4: Illustration of the sequence of iterates obtained while solving a timestep using the method in [41]. The solid circular markers denote the five iterates that occur during the solution process. The dashed arrows are the enumerated sequence of steps taken.

Owing to the non-monotone nonlinear structure of the flux function, the standard Newton iteration fails to converge for all but the smallest timestep sizes. The safe-guarded iterations introduced in [41] guarantee convergence for any timestep size of this problem. Moreover, the rate of convergence using the algorithms in [41] are likely the fastest available using a general Newton-like iteration. Figure 8.4 illustrates the sequence of iterations obtained by using the second modified algorithm in [41] to solve a timestep, $\Delta t = 150$. Five iterations are necessary for convergence in this case. The first iteration attempts to cross the inflection point and the sign of the curvature changes at the two iterates. The algorithm selects the first iterate as the point in between denoted by the solid black circle in the figure. The following four steps do not cross the inflection point and therefore are unaltered Newton steps.

Figure 8.5 illustrates the sequence of iterates obtained using the Continuation-Newton algorithm. For this case, three tangent steps and two Newton corrections are required, giving a total of five iterations. The model example illustrates a key

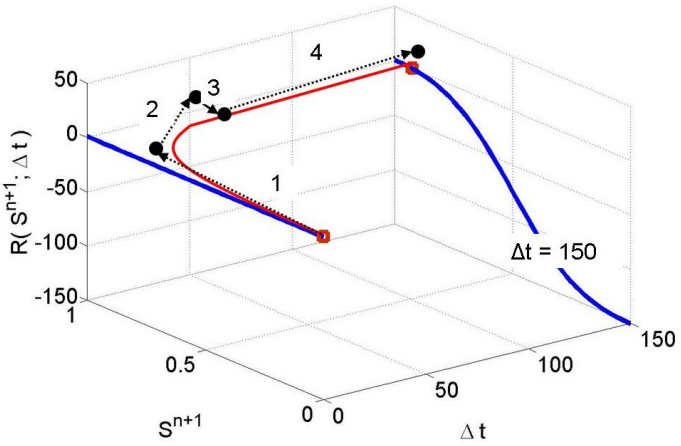


Figure 8.5: Illustration of the sequence of iterates taken to solve a timestep using the Continuation-Newton algorithm. Steps three and five are Newton correction steps, while the remainder are tangent steps.

qualitative difference between using the Continuation-Newton algorithm and Newton-like methods. While Newton-like methods need to be safeguarded to resolve nonlinear structure in the residual curve, Continuation-Newton methods need to safely traverse strong nonlinearity in the Solution Path. In this example, the CN iterates are within a convergence neighborhood that excludes the inflection points. On the other hand, the initial rapid change in saturation implies strong curvature in the solution path, which in turn force the use of smaller tangent steps.

8.3.2 Buckley-Leverett models

We consider two illustrative cases of the Buckley-Leverett Problem, which is described in Section 6.1.2. The first case is a horizontal piston-like displacement, and the second includes gravity effects and exhibits counter-current flow. In both cases, the injection saturation is one, $S_{inj} = 1$, and the relative permeability functions are quadratic with end-points of zero and one.

Horizontal displacements.

For this case, the end-point mobility ratio M^0 is chosen as 10, the Gravity Number Ng is chosen as 0 for a horizontal problem, the initial saturation, S_{init}, is zero, and $N = 150$.

We illustrate the CN concepts by applying a step of the algorithm for this problem, starting from a simulation time of 0.5. At this time, the saturation state, denoted S^0, and the tangent update $\hat{\delta}^0$ computed using Algorithm 5, are presented in Figure 8.6.

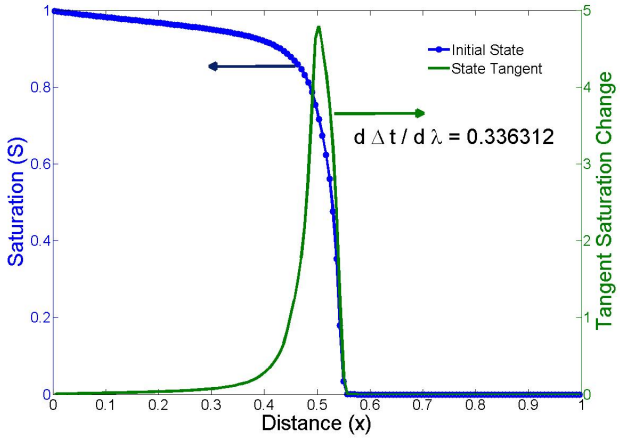


Figure 8.6: Starting iterate and tangent update for a 1-dimensional Buckley-Leverett problem with no gravity.

The CN starting iterate to solve a target timestep from this time consists of the saturation profile, S^0, in Figure 8.6, augmented with a timestep size of zero; that is $p_0 = (S^0, 0)$. The solution path emanates from p_0, and is in an $N + 1$-dimensional space. The tangent to the solution path at the starting point is the state update component plotted in Figure 8.6, augmented with $\frac{d\Delta t}{d\lambda}$, which in this case is 0.336. The next CN iterate is a step from the starting point, along a linearly scaled update of the tangent. This update is written as $p_1 \leftarrow p_0 + \alpha\hat{\delta}$, where the scalar step-length, $\alpha_{min} \leq \alpha \leq \alpha_{max}$, is to be selected by Algorithm 7.

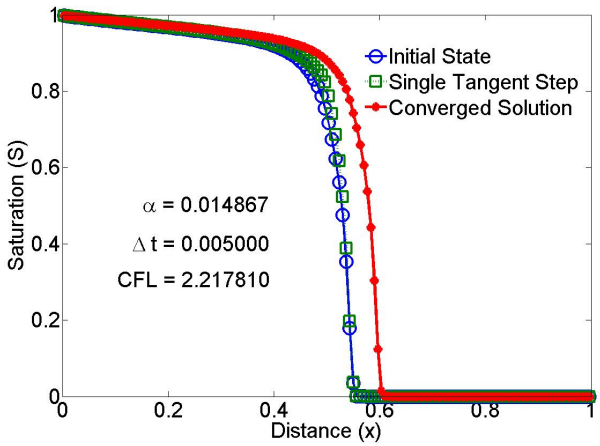
We illustrate the qualitative nature of this tangent update using two different step-lengths, $\alpha = 0.015$ and 0.089. For each step-length, the corresponding tangent

update timestep size is $\alpha\frac{d\Delta t}{d\lambda}$, and in this case, these are 0.005 and 0.03. Moreover, defining the Courant-Friedrichs-Lewy (CFL) Number as $\frac{\Delta t}{\Delta x}\max|f'(S)|$, these step-lengths correspond to 2.25 and 13.35 CFL. Figures 8.7(a) to 8.7(b) show the initial saturation state, as well as the updated states obtained by taking a single tangent step using each of the two selected step-lengths. The solutions for each of the two timestep sizes which correspond to the two step-lengths are also shown in Figures 8.7(a) to 8.7(b). In these figures, the solutions are obtained by a Newton process, which requires several iterations, whereas the tangent steps are single iterates in the CN algorithm. Qualitatively, it is observed that as the step-length is increased, the tangent step becomes a worse approximation of its corresponding solution, as expected. The objective is to choose the largest step-length that still gives a good approximation (i.e., is close to the solution path).

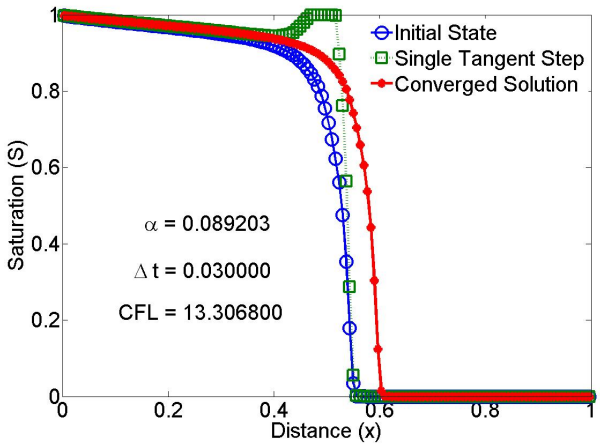
Figure 8.8 shows the trend in the residual norm for various choices of step-length along the tangent. The figure also shows the residual norms obtained using the old state as an initial guess for the corresponding timestep size of the continuation step-length. Within smaller step-length ranges, there is higher confidence in the tangent update solution than in the old state. The step-length selection algorithm attempts to select the largest step-length that retains a residual below a certain tolerance. For low residual tolerances, which are typically used to define a convergence neighborhood, the continuation tangent step affords a timestep advancement that is (1) known *a priori* through the relation $\Delta t^1 \leftarrow \Delta t^0 + \alpha\frac{d\Delta t}{d\lambda}$, and (2) always closer to the solution than the old state is.

In terms of the overall performance of CN for a single timestep, Figure 8.9(a) shows the number of iterations required to solve a set of different target timestep sizes ranging up to a size corresponding to over 440 CFL. For each target timestep size, starting from the simulation time 0, the iterations are performed using the (1) Appleyard Newton approach, (2) Modified Appleyard Newton method, and (3) the proposed CN algorithm. In Figure 8.9(a), we show the split between the number of iterations taken in CN tangent steps and in CN Newton corrections.

The standard Appleyard-Newton algorithm does not converge for any of the



(a)



(b)

Figure 8.7: For a particular time, the current state is depicted with circle markers, the tangent update with star markers, and the actual solution with square markers. This is presented for two different continuation step-lengths.

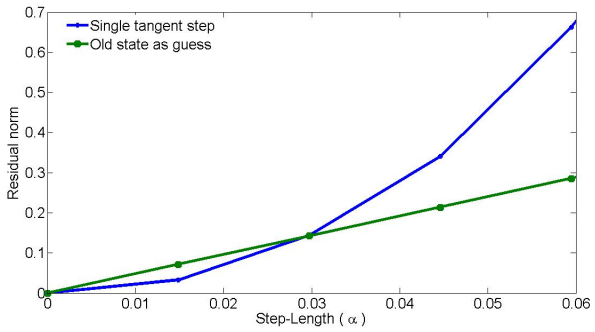


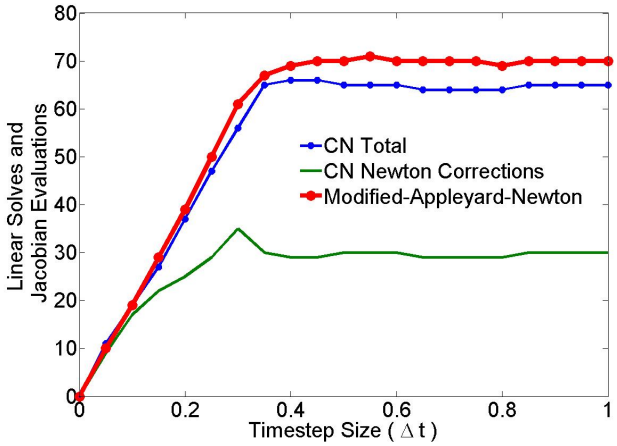
Figure 8.8: Residual norms for various step-lengths along a single tangent.

timestep sizes attempted. Moreover, while in general, the CN and Modified Appleyard approaches are expected to perform on par, in this case the CN algorithm requires fewer overall iterations. This is the case since the additional dimension of the tangent update, $\frac{d\Delta t}{d\lambda}$, provides a temporal scaling regarding the range of validity of an update, whereas scaling a Modified Appleyard Newton step never influences the timestep which it attempts to solve. Figure 8.9(b) shows the number of residual evaluations required to converge a timestep by each of the methods. Owing to the step-length search component of the CN algorithm, it requires more residual evaluations. The precise number required is bounded by the maximum allowed backtracking steps per tangent step. In this case they were limited to 5. Note that although in this case, the proposed algorithm performs on par with a state-of-the-art solver, each of the CN iterates is associated with a time-step size. Subsequently, in return for a few additional evaluations of the residual, the CN algorithm does not require timestep chops nor will it waste any iterations over the course of an entire simulation.

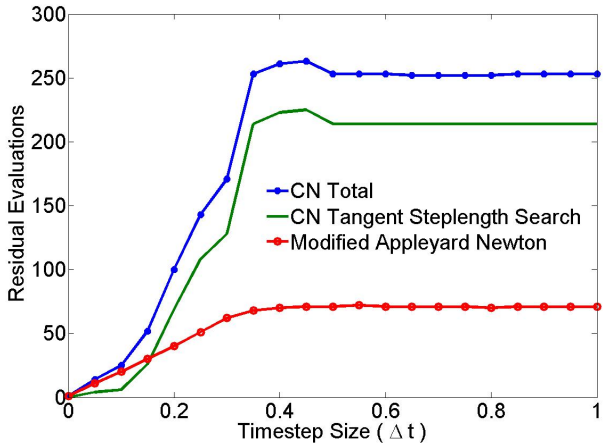
Down-dip displacements.

For this case, the end-point mobility ratio, M^0, is chosen as 0.5, and the Gravity Number, Ng, is chosen as -5 for a down-dip problem. The number of grid blocks used is $N = 150$. The initial condition has a saturation of one up to a distance of 0.34, and a saturation of zero elsewhere.

We consider a single timestep starting at the time of 0.1, and we illustrate the



(a) Number of Jacobian evaluations and linear solves required for convergence using CN and Modified Appleyard Newton



(b) Number of residual evaluations required for convergence using CN and Modified Appleyard Newton

Figure 8.9: Convergence characteristics of the CN method compared to Modified Appleyard Newton.

components of a single CN iteration. The current saturation state and the computed tangent update are shown in Figure 8.10. In this case, counter-current flow of the two phases results in negative components of the tangent vector. Moreover, the displacement involves spreading wave portions. For two step-lengths corresponding to timestep sizes of 0.001 and 0.006, Figures 8.11(a) and 8.11(b) show the initial state, the tangent iterate, and the corresponding complete solution for the timestep. Defining the Courant-Friedrichs-Lewy (CFL) Number for this problem as $\frac{\Delta t}{\Delta x} \max |f'(S)|$, the timesteps correspond to 0.7 and 4.2 CFL. Qualitatively, the linear tangent updates with large step-lengths are worse approximations in the sense that they display sharp changes in saturation.

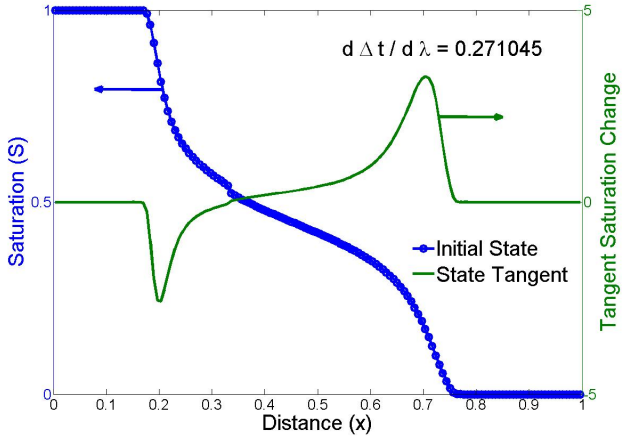
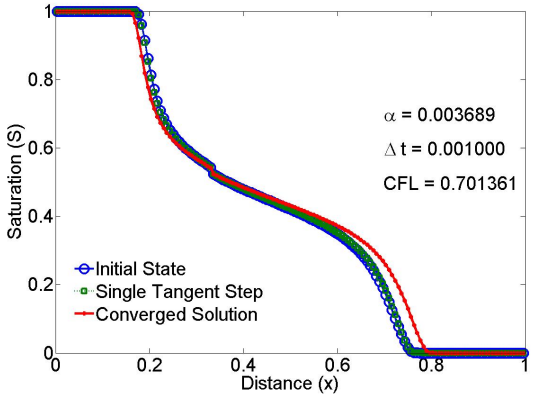


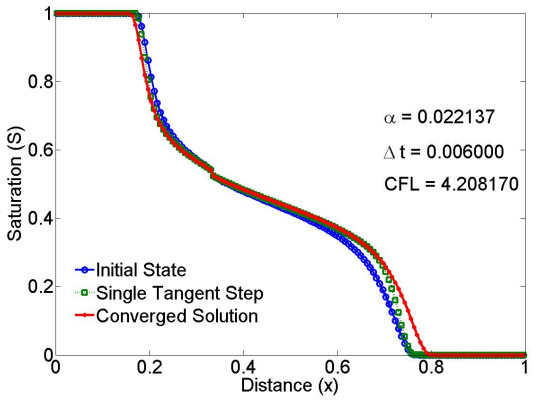
Figure 8.10: A starting point for a continuation timestep and the corresponding initial tangent vector.

Figure 8.12 shows the relationship between the computed residual norm and increasing step-length. Once again, for smaller residual tolerances, the tangent approximations provide better estimates than the old state.

Figures 8.13(a) and 8.13(b) show the overall performance for various timesteps from the start of the simulation. The largest timestep size tested is 0.2, which corresponds to a CFL number of 140. In this case, the standard Appleyard Newton method converges only for the smallest timestep size. Moreover, beyond a timestep size of 0.04, corresponding to a CFL number of 30, the CN algorithm requires more



(a)



(b)

Figure 8.11: For a particular time, the current state is depicted with a solid line, the tangent update with a dotted line, and the actual solution with a dashed line. This is presented for two different continuation step-lengths.

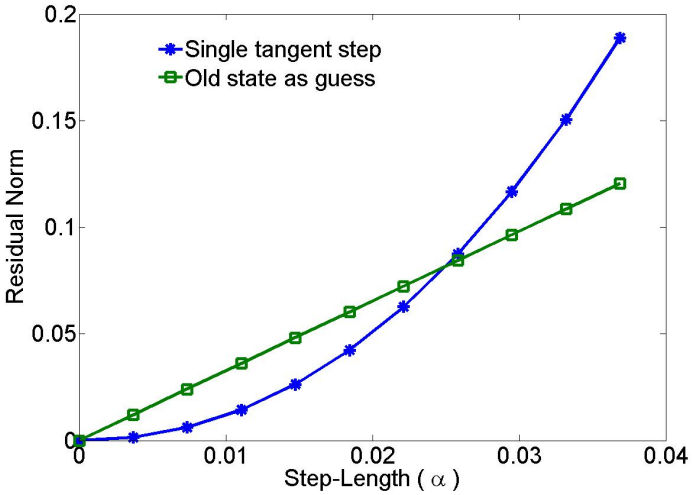
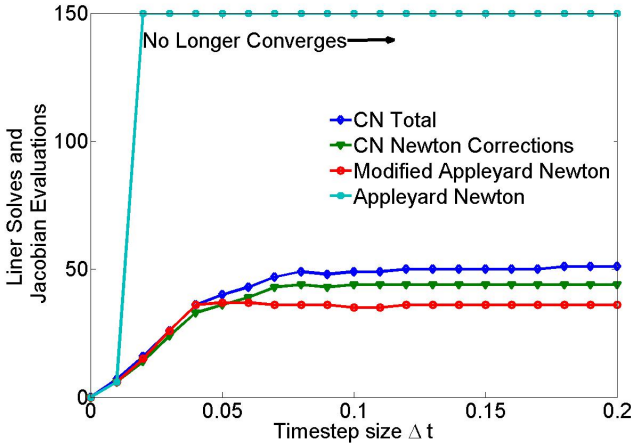
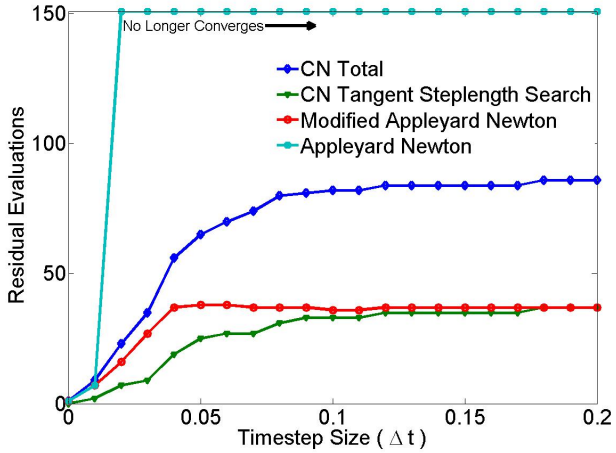


Figure 8.12: The residual norm versus step-length size using a tangent step, and the initial state as a guess.

iterations than the Modified Appleyard Newton method. This is due to the fact that the solution involves a spreading wave that influences the entire domain.



(a) Number of Jacobian evaluations and linear solves required for convergence using CN and Modified Appleyard Newton



(b) Number of residual evaluations required for convergence using CN and Modified Appleyard Newton

Figure 8.13: Convergence characteristics of the CN method compared to Modified Appleyard Newton.

Chapter 9

Computational Examples

The Localization algorithm is combined with Continuation-Newton such that all linear solution processes, whether they be for Newton steps or for Tangent steps, use the Localization routine. A suite of cases of two-phase flow with gravity is used to investigate the properties of the proposed Adaptively Localized Continuation Newton (ALCN) algorithm.

9.1 Two-phase compressible flow in two dimensions

The problem involves two-phase compressible flow in two-dimensions. Gravity effects are superposed with nonlinear relative permeability models and injection and production wells in order to stress the nonlinear solution aspect. The unknowns are pressure, $p(x, y, t)$, and water saturation, $S_w(x, y, t)$. The governing equations for the conservation of oil and water appear in Equations 9.1.1 and 9.1.2 respectively.

$$[\phi(1 - S_w)]_t - \nabla \left[K \frac{K_{ro}}{\mu_o} \nabla (p + \gamma_o h) \right] = q_o \tag{9.1.1}$$

$$[\phi S_w]_t - \nabla \left[K \frac{K_{rw}}{\mu_w} \nabla (p + \gamma_w h) \right] = q_w \tag{9.1.2}$$

The compressibility in the problem is due to a pressure-dependent porosity relation as specified by Equation 9.1.3.

$$\phi = \phi_{ref} \left(1 + c_r \left(p - p_{ref}\right)\right) \tag{9.1.3}$$

Other parameters in this problem are;

- A diagonal permeability tensor, K, which may be heterogeneous.
- μ_w, and μ_o, denoting constant water and oil viscosity respectively.
- K_{rw}, and K_{ro}, denoting the water and oil relative permeability described by Equations 6.1.6 and 6.1.7.
- γ_w, and γ_o, denoting the constant water and oil gravimetric density.
- h, denoting depth along the direction of gravity.

We apply a fully implicit discretization with standard single-point phase-based upstream weighting. The oil conservation equation is aligned with pressure, and the water equation with water saturation. We assume no flow boundary conditions across the rectangular domain of dimensions L_x and L_y. A rate-controlled injector and Bottom Hole Pressure (BHP) producer are introduced, and are completed in single blocks. The initial condition maybe transient, allowing the specification of an arbitrary initial saturation distribution. In such cases, the pressure distribution is initialized according to gravity and a specified pressure at the top of the reservoir.

9.1.1 Examples

We present results for two representative cases. The first is a gravity segregation problem with a non-stationary initial condition. This case is known to stress the nonlinear solution aspect of the transport. The second case superposes gravity effects and nonlinear relative permeability with injection and production wells. Both cases share the following common parameters. The square reservoir is assumed to have no flow boundaries, with dimensions of 100ft, and is discretized using 100 blocks

along both dimensions. An initial pressure of $500psi$ is applied to the top of the reservoir. We assume a uniform initial porosity of $\phi_{ref} = 0.4$ at a reference pressure of $P_{ref} = 14.7$ psi, a rock compressibility of $c_r = 1E-06psi^{-1}$, and an isotropic permeability field of 500 mD. The normalized relative permeability models used are quadratic, and are scaled to end-points of zero and one. The oil and water have gravimetric densities of 50 and 100, and viscosities of 5cP and 1cP respectively. The residual 2-norm convergence tolerance of 1E-06 is applied, and a CN convergence neighborhood tolerance of 1E-03 is used.

Pure gravity segregation.

We consider a case of pure gravity segregation where initially, oil saturates the lower portion of a reservoir, and water saturates the top. Since the boundary conditions are no flow, the problem involves bouncing waves that interact every time they collide with each other, or with the top and bottom reservoir boundaries. The steady state solution is an equilibrium with oil on the top. Figures 9.1(a) through 9.1(c) show oil saturation snapshots over the course of a simulation.

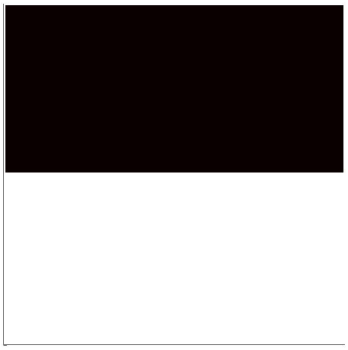
In order to derive an appropriate cell-based CFL number, we assume incompressible flow and a zero total velocity. Given these assumptions, the cell-based CFL number for cell i, is given as,

$$CFL_i = \Delta t \frac{K\left(\gamma_w - \gamma_o\right)}{\phi_i \mu_o \Delta y} \max F'\left(S\right), \tag{9.1.4}$$

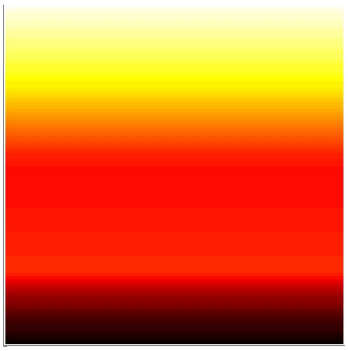
where $F'\left(S_i\right)$ is the slope of the following flux function,

$$F\left(S_i\right) = \frac{K_r o}{1 + \frac{K_r o}{K_r w}\frac{\mu_w}{\mu_o}}. \tag{9.1.5}$$

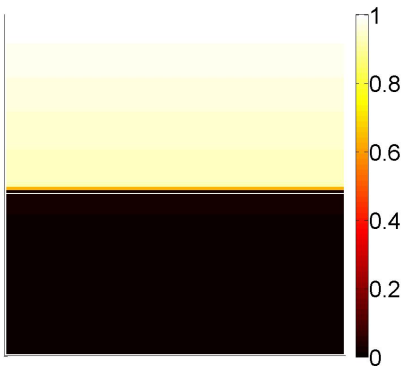
Figure 9.2(a) shows the sorted distribution of CFL numbers throughout the mesh. These cell-based CFL numbers are computed using the initial condition. For timestep sizes greater than 4.5 days, all of the cells in the domain experience CFL numbers that are greater than one.



(a) 0 days



(b) 1350 days



(c) 4500 days

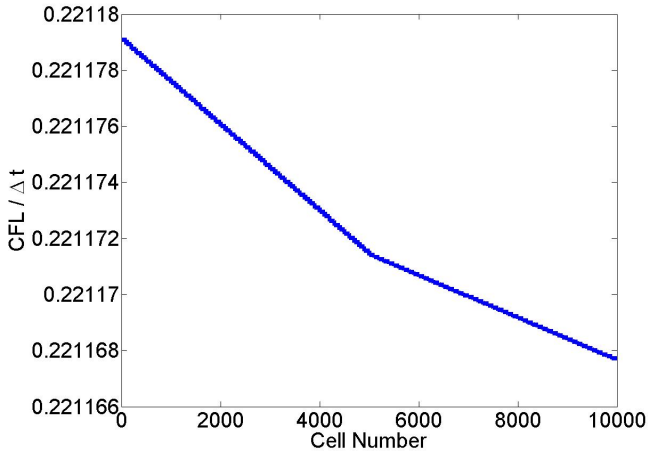
Figure 9.1: Oil saturation snapshots over simulations for a gravity segregation case.

Figure 9.2(b) shows the iteration count to convergence over a series of experiments. In each experiment, a target timestep is set from the initial condition, and the corresponding system is solved. A maximum number of 150 iterations is allowed, and an iteration count of 150 implies lack of convergence. The standard Newton and Appleyard-Newton methods failed to converge for the smallest timestep size tested. Figure 9.2(b) shows that the proposed algorithm provides improved asymptotic convergence rates, as well as robustness compared to the Modified-Appleyard-Newton method. Figure 9.3 shows the average fraction of the total unknowns that is solved for at each iteration using ALCN. The fractions vary with timestep size since the locality of the saturation changes also vary in time. As the segregation process approaches steady state, we observe greater locality, whereas at early times, the saturation over the entire domain experiences a slowly varying spreading wave. Moreover, since the Newton correction steps within the ALCN algorithm are computed within the convergence neighborhood, we see that they only modify a small percentage of the unknowns.

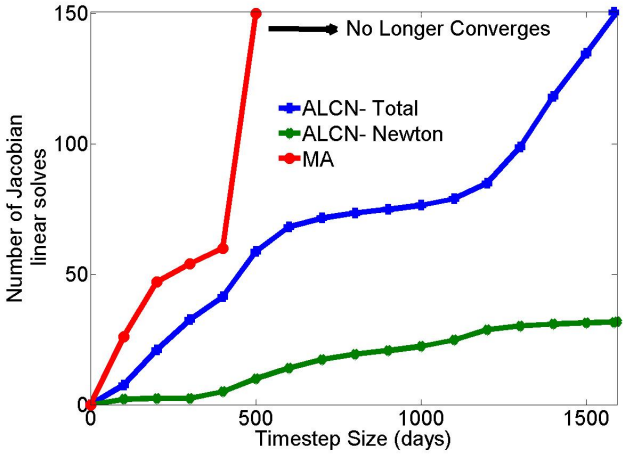
Gravity effects coupled to well-driven flow.

Next, we consider a case that involves both gravity segregation and well-driven flow. Initially, the reservoir is saturated with oil in the upper half and water in the lower half. A single block injector is completed in the upper-right corner, and a producer is completed in the lower-left. A constant water injection rate of 10 BBL per day is specified. The producer is operated with a constant Bottom Hole Pressure of $500psi$. Figures 9.4(a) through 9.4(c) show water saturation snapshots at 0, 120, and 245 days over the course of a simulation.

The results in Figure 9.5 illustrate the computational effort required to solve a full simulation using a single timestep. Various timestep sizes are tested using the proposed Adaptively-Localized-Continuation-Newton (ALCN) algorithm, as well as the Modified Appleyard Newton method. A dimensionless measure of these timestep sizes is the Cell Pore Volumes Injected (CPVI), which for this case corresponds to 10 CPVI for a timestep size of 1 day. Figure 9.5(a) shows the count of linear solves required to converge a timestep using both methods. Once again, the proposed ALCN



(a) Sorted distribution of CFL numbers throughout the domain



(b) The number of iterations required to solve a set of sample timestep sizes using the Modified Appleyard heuristic, and the proposed ALCN algorithm.

Figure 9.2: Results for a gravity segregation problem with compressibility.

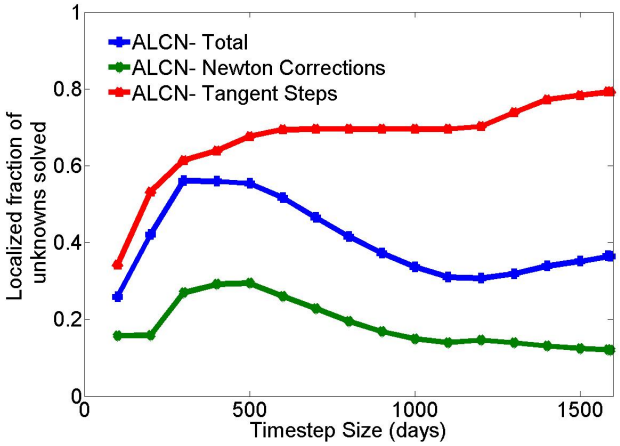
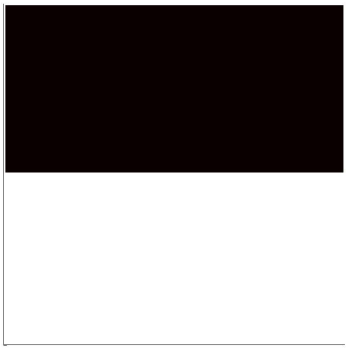


Figure 9.3: The average fraction of unknowns solved for at each iteration using localization.

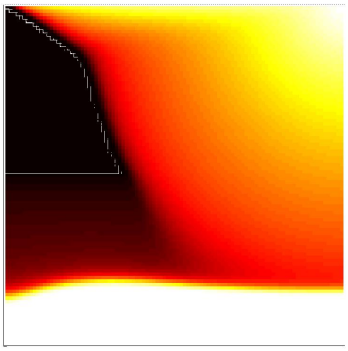
algorithm provides improved asymptotic convergence rates over the tuned Modified Appleyard method. Moreover, both the standard Newton and Appleyard Newton methods fail to converge for the smallest timestep size tested. Figure 9.5(b) shows the count of residual evaluations required by the solution processes. Note that the ALCN method does require more residual evaluations due to the step-length selection component of ALCN.

9.2 Summary

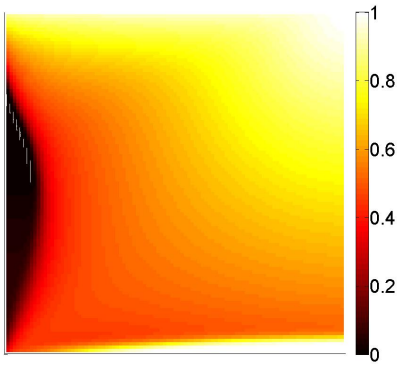
The CN algorithm evolves the residual equations in the augmented $(U, \Delta t)$ solution space providing iterates that are solutions to known timestep sizes. Subsequently, whenever a CN iteration is terminated, we either obtain the solution to the target timestep, or a solution to a smaller known timestep. The rate of convergence is empirically shown to be on par with that of state-of-the-art safeguarded Newton's methods, whenever such schemes converge. Qualitatively, the number of iterations required to convergence by both variants of Newton's method and CN scale with the dimensionless target timestep. This is because both iterations exploit linearized local wave propagation speed information. The chief difference is that while Newton's



(a) 0 Days

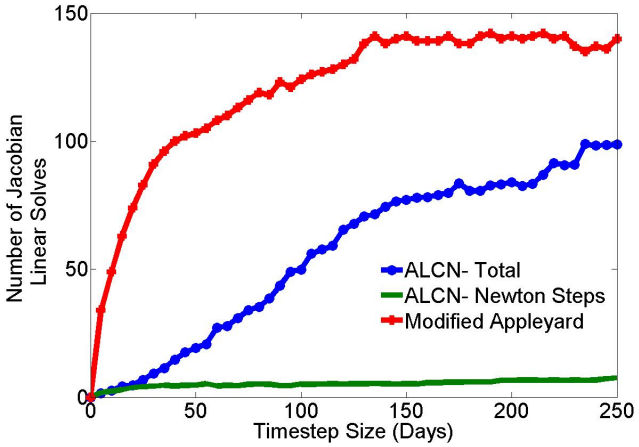


(b) 120 Days

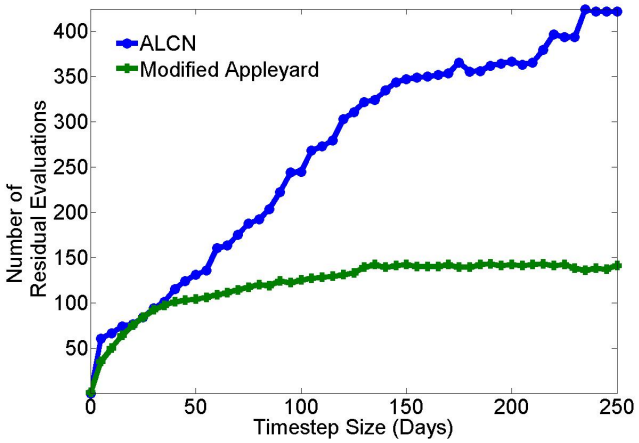


(c) 245 Days

Figure 9.4: Water saturation snapshots for an unstable injection problem with gravity.



(a) The number of linear solves (iterations) required to solve a set of sample timestep sizes using the Modified Appleyard heuristic, and the proposed Adaptively-Localized-Continuation-Newton (ALCN) algorithm.



(b) The number of residual evaluations performed during the solution of a set of sample timestep sizes using the Modified Appleyard heuristic, and the proposed Adaptively-Localized-Continuation-Newton (ALCN) algorithm.

Figure 9.5: Computational effort required to solve a full simulation in one timestep using the proposed Adaptively-Localized-Continuation-Newton (ALCN) algorithm, and the Modified Appleyard Newton method.

method is oblivious to the nature of the time evolution of the solution within a timestep, the CN algorithm exploits the evolution of the discrete residual equations in time.

Chapter 10

Discussion And Future Work

Modern timestep controllers for convergence rely on *a posteriori* estimates and on extrapolation models of these estimates in order to anticipate the convergence behavior of a Newton-like method to solve an upcoming timestep. Invariably, practical implementations of such controllers (see for example [59, 38]) necessitate the use of *try-adapt-try-again* strategies; try a recommended timestep size, and observe the resulting convergence behavior. If necessary, adapt the *a posteriori* estimates using the observations, and recommend a new, restricted timestep size to try out. This wasteful strategy is necessary since the *a posteriori* estimates are not strict bounds and the extrapolation models are inherently not predictive for nonlinear phenomena. If available, accurate *a priori* measures of the asymptotic relation between the nonlinear convergence rate and time-step size can be used to directly without the need for an extrapolator. This however remains an understudied problem in the context of large scale nonlinear problems where few locality or smoothness assumptions can be made in general. In this work, we restrict interest to the types of problems encountered in flow through porous media, and in doing so, we exploit the nature of the underlying physics in order to arrive at an *a priori* criterion.

We developed and illustrated an algorithm that solves implicit residual systems using a combination of Newton's method and a Continuation on timestep size. The algorithm guarantees convergence for any timestep size, and if the iteration process is stopped before the target timestep is reached, the last iterate is a solution to a

smaller known timestep. The performance of the algorithm was illustrated using several challenging nonlinear problems. Compared to state-of-the-art problem-tuned heuristic methods, the CN algorithm remains competitive. Additionally, the CN algorithm does not require wasteful timestep cuts, unlike all Newton variants. Moreover, a localization strategy that is based on solution propagation properties of the transport equations, is used to reduce the computational cost associated with solving the linear systems. With the combination of these key ideas, reservoir simulators no longer require timestep chopping heuristics, no computational effort can be wasted, and the amount of computation required per iteration can be reduced substantially.

The CN algorithm presented in this work applies a first-order explicit scheme to follow the solution path loosely. We anticipate increased computational efficiency through the development of stabilized CN variants which exploit the relationship between Newton and tangent steps within the augmented space. This may lead to a characterization of the CN convergence rate in terms of the order of accuracy in approximating the solution path.

We also anticipate the application of CN in order to introduce timestep control based on accuracy concerns. That is, the CN algorithm offers a built-in facility to perform timestep selection for the control of time approximation errors. Since each iterate solves a larger timestep than its successor, error extrapolation techniques may be applied directly to dictate which timestep to stop at, thereby providing an *a priori* error controller that is built into the solver, and that does not waste computation.

Nomenclature

Abreviations

AD Automatic Differentiation
ADETL Automatically Differentiable Expression Templates Library
AIM Adaptive Implicit Method
ATLAS Automatically Tuned Linear Algebra System
AVSO Automatic Variable Set Object
BHP Bottom Hole Pressure
BLAS Basic Linear Algebra Set
CFL Courant-Friedrichs-Lewy
CN Continuation Newton
EA Eclipse Appleyard
EOR Enhanced Oil Recovery
ET Expression Template
FIM Fully Implicit Method
GPGPU General Purpose Graphics Processing Unit
GPRS General Purpose Research Simulator
IN Inexact Newton
JIT Just In Time
MA Modified Appleyard
MTL Matrix Template Library
OO Operator Overloading
OOAD Operator Overloaded Automatic Differentiation

OOP Object Oriented Programming
OSKI Optimized Sparse Kernel Interface
PDAE Partial Differential Algebraic Equations
POOMA Parallel Object Oriented Methods and Applications
QN Quasi Newton
SN Standard Newton
SPLC SParse Linear Combination
STL Standard Template Library

Symbols

i = unknown index.
j = index of cell with non-zero material balance.
k = Continuation-Newton iteration index.
n = timestep number.
p_k = point in the augmented space.
p_{int} = point in the augmented space that is within the interior of a convergence neighborhood.
t_n = time at timestep number n.
C = a positive constant.
J = Jacobian matrix.
M^0 = endpoint Mobility Ratio.
Ng = Gravity Number.
R = residual system of nonlinear equations.
S = saturation unknowns.
S_{inj} = injection saturation.
S_{init} = initial saturation.
S_{int} = saturation state within the interior of a convergence neighborhood.
U^n = vector of unknowns at timestep number n.
U_0 = vector of unknowns at the beginning of a CN process.

α = CN tangent update steplength.

α_{max} = maximum allowable CN tangent update step-length.

α_{min} = minimum allowable CN tangent update step-length.

δ = CN tangent in augmented space.

$\hat{\delta}$ = normalized CN tangent in augmented space.

δ_U = state update component of CN tangent.

$\delta_{\Delta t}$ = timestep update component of CN tangent.

λ = solution path arc-length parametrization.

ν = Newton iteration index.

θ = local attenuation ratio.

$\Delta\nu$ = step-length along a Newton direction.

Δt = timestep size.

Δt_{target} = target timestep size.

Δt_{int} = timestep component of a point within a convergence neighborhood in augmented space.

Λ = diagonal weighting matrix for a general safeguarded Newton iteration.

ζ = local scaling constant.

$\mathcal{C}$ = solution path in augmented space.

$\mathcal{N}$ = convergence neighborhood about a solution path in augmented space.

Bibliography

[1] Alexandrescu, A. *Modern C++ Design: Generic Programming and Design Patterns Applied*. The C++ In-Depth Series, Addison-Wesley, MA, 2001.

[2] Allgower, E., and Georg, K. Simplicial and continuation methods for approximations, fixed points and solutions to systems of equations. *SIAM Review 22* (1980).

[3] Allgower, E., and Georg, K. *Introduction to Numerical Continuation Methods*. Classics in Applied Mathematics, SIAM, 2003.

[4] Aziz, K., and Settari, A. *Petroleum Reservoir Simulation*. Elsevier Applied Science, 1979.

[5] Beckner, B., Hutfilz, J., Ray, M., and Tomich, J. EM: New reservoir simulation system. In *SPE 68116, Proceedings of the 2001 SPE Middle East Oil Show* (March 2001), SPE.

[6] Bendtsen, C., and Stauning, O. FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, aug 1996.

[7] Bischof, C., Corliss, G., Green, L., Griewank, A., Haigler, K., and Newman, P. Automatic differentiation of advanced CFD codes for multidisciplinary design. *Computing Systems in Engineering 3*, 6 (1992), 625 – 637.

[8] Bischof, C., Khademi, P., Mauer, A., and Carle, A. Adifor 2.0: automatic differentiation of Fortran 77 programs. *Computational Science Engineering, IEEE 3*, 3 (fall 1996), 18 –32.

[9] Bischof, C. H., Roh, L., and Mauer-Oats, A. J. ADIC: an extensible automatic differentiation tool for ANSI-C. *Software Practice and Experience 27* (December 1997), 1427–1456.

[10] Broyden, C. G. A class of methods for solving nonlinear simultaneous equations. *Math. Comp. 19* (1965), 577–593.

[11] Bucker, M., Corliss, G., Hovland, P., Naumann, U., and Norris, B. On automatic differentiation. In *Automatic Differentiation: Applications, Theory, and Implementations* (2006), Springer Lecture Notes in Computational Science and Engineering.

[12] Bulka, D., and Mayhew, D. *Efficient C++: performance programming techniques*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[13] Cao, H. *Development of Techniques for General Purpose Simulators*. PhD thesis, Stanford University, 2002.

[14] Cao, H., and Aziz, K. SPE 77720, performance of IMPSAT and IMPSAT-AIM models in compositional simulation. In *SPE Annual Technical Conference, San Antonio* (Sept. 2002), SPE.

[15] Chen, Y., and Durlofsky, L. Adaptive local-global upscaling for general flow scenarios in heterogeneous formations. *Transport in porous Media 62* (2006), 157–185.

[16] Christie, M. A., and Blunt, M. J. SPE 72469, Tenth SPE comparative solution project: A comparison of upscaling techniques. *SPE Reservoir Evaluation And Engineering 4* (August 2001), 308–317.

[17] The Computer Modelling Group. *STARS User's Guide.* http://www.cmg.com.

[18] Corey, A. The interrelation between gas and oil relative permeabilities. *Producers Monthly 19*, 1 (1954), 38 – 41.

[19] Corliss, G., Faure, C., Griewank, A., Hascoët, L., and Naumann, U., Eds. *Automatic differentiation of algorithms: from simulation to optimization.* Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[20] Corliss, G. F., Bischof, C., Griewank, A., Wright, S. J., and Robey, T. Automatic differentiation for PDEs: Unsaturated flow case study. In *Presented at the 7th International Association of Mathematics and Computer Simulation (IMACS) International Conference on Computer Methods for Partial Differential Equations, New Brunswick, NJ, 22-24 Jun. 1992* (1992), H. J. Shih, W. Schiesser, & J. Ellison, Ed., pp. 22–24.

[21] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms.* MIT Press, Cambridge MA, 2001.

[22] Deb, M. K., Reddy, M. P., Thuren, J., and Adams, W. A new generation solution adaptive reservoir simulator. In *SPE 30720 presented at the 1995 SPE Annual Technical Conference, Dallas* (October 1995), SPE.

[23] DeBaun, D., Byer, T., P.Childs, Chen, J., Saaf, F., Wells, M., Liu, J., Cao, H., L.Pianelo, Tilakraj, V., Crumpton, P., Walsh, D., Yardumain, H., Zorzynski, R., Lim, K. T., Schrader, M., Zapata, V., Nolen, J., and Tchelepi, H. SPE 93274: An extensible architecture for next generation scalable parallel reservoir simulation. In *Proceedings of the 2005 SPE Reservoir Simulation Symposium* (February 2005), SPE.

[24] Deuflhard, P. Global inexact Newton methods for very large scale nonlinear problems. *IMPACT of Computing in Science and Engineering 3*, 4 (1991), 366 – 393.

[25] Deuflhard, P. *Newton Methods for Nonlinear Problems; Affine Invariance and Adaptive Algorithms.* Springer, 2004.

[26] Duff, I. S., Heroux, M. A., and Pozo, R. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Trans. Math. Softw. 28* (June 2002), 239–267.

[27] Fischer, H. Special problems in automatic differentiation. In *Automatic Differentiation of Algorithms* (1991), SIAM, PA.

[28] Forth, S., Tadjouddine, M., Pryce, J., and Reid, J. Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding. *ACM Transactions on Mathematical Software 30* (2004).

[29] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing Series, Addison-Wesley, MA, 1995.

[30] Garlan, D., and Shaw, M. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering* (Singapore, 1993), V. Ambriola and G. Tortora, Eds., World Scientific Publishing Company, pp. 1–39.

[31] Garlan, D., and Shaw, M. An introduction to software architecture. Tech. Rep. CMU-CS-94-166, Carnegie Mellon University, January 1994.

[32] Gebremedhin, A., Manne, F., and Pothen, A. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review 47*, 4 (December 2005), 629 – 705.

[33] Geoquest, Schlumberger. *Eclipse 100 Technical Description 2000A.* http://www.sis.slb.com/content/software/simulation/eclipse-blackoil.asp.

[34] Griewank, A. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications* (1990), Kluwer Academic Publishers, IL.

[35] Griewank, A., Juedes, D., and Utke, J. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software 22*, 2 (1996), 131–167.

[36] Griewank, A., and Walther, A. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, second ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

[37] Gropp, W., Kaushik, D., Keyes, D., and Smith, B. High performance parallel implicit CFD. *Parallel Computing 27* (2001), 337–362.

[38] Gustafsson, K. Control-theoretic techniques for stepsize selection in implicit Runge-Kutta methods. *ACM Transactions of Mathematical Software* (1994), 496–517.

[39] Haney, S. Is C++ fast enough for Scientific Computing? *Computers in Physics 8* (1994).

[40] Haney, S., and Crotlinger, J. How templates enable high-performance scientific computing in C++. *Computing in Science Engineering 1*, 4 (jul/aug 1999), 66 –72.

[41] Jenny, P., Tchelepi, H. A., and Lee, S. H. Unconditionally convergent nonlinear solver for hyperbolic conservation laws with s-shaped flux functions. *Journal of Computational Physics 228*, 20 (2009), 7497 – 7512.

[42] Karmesin, S., Crotinger, J., Cummings, J., Haney, S., Humphrey, W. J., Reynders, J., Smith, S., and Williams, T. Array design and expression evaluation in POOMA II. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments* (London, UK, 1998), ISCOPE '98, Springer-Verlag, pp. 231–238.

[43] Keyes, D. Terascale implicit methods for partial differential equations. *Proceedings of The Barrett Lectures, AMS Contemporary Mathematics, Providence 306* (2002), 29–84.

[44] Kim, J., and Finsterle, S. Application of automatic differentiation in Tough2. In *Proceedings of The Tough Symposium, LBNL* (May 2003), LBNL.

[45] Kirby, R. C. A new look at expression templates for matrix computation. *Computing in Science Engineering 5*, 3 (may-june 2003), 66 – 70.

[46] Kwok, F., and Tchelepi, H. Potential-based reduced newton algorithm for nonlinear multiphase flow in porous media. *Journal of Computational Physics 227* (2007), 706–727.

[47] LeVeque, R. J. *Numerical Methods For Conservation Laws. Lectures in Mathematics, ETH Zurich*. Birkhauser Verlag, Switzerland, 1992.

[48] Li, X. S., and Demmel, J. W. SuperLU DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transact ions on Mathematical Software (TOMS) 29*, 2 (2003), 110–140.

[49] Mclachlan, R. I., and Quispel, G. R. W. Splitting methods. *Acta Numerica 11*, -1 (2002), 341–434.

[50] Muller, S., and Stribia, Y. Fully adaptive multiscale schemes for conservation laws employing locally varying time stepping. *Journal of Scientific Computing 30* (2007), 493–531.

[51] Naccache, P. A fully-implicit thermal reservoir simulator. In *Proceedings of The 14th SPE Symposium on Reservoir Simulation, Dallas* (June 1997), SPE, pp. 97–103.

[52] Ortega, J., and Rheinboldt, W. *Iterative Solution of Nonlinear Equations in Several Variables*. New York Academic Press, 1970.

[53] Parashar, M., Wheeler, J., Pope, G., K.Wang, and Wang, P. A new generation EOS compositional reservoir simulator: Part II - framework and multiprocessing. In *SPE 37977 presented at the 1997 SPE Reservoir Simulation Symposium, Dallas* (June 1997), SPE.

[54] Plewa, T., and Linde, T. *Adaptive Mesh Refinement - Theory and Applications. Proceedings of the Chicago Workshop on Adaptive Mesh Refinement in Computational Science and Engineering, 2003*. Springer, Berlin, 2003.

[55] Pope, G. A., and Nelson, R. C. A chemical flooding compositional simulator. *SPE Journal 18*, 5 (1978), 339–354.

[56] Rall, L. Perspectives on automatic differentiation: Past, present, and future. In *Automatic Differentiation: Applications, Theory, and Implementations)* (2005), Lecture Notes in Computer Science and Engineering, Springer-Verlag.

[57] Robison, A. C++ gets faster for Scientific Computing. *Computers in Physics 10* (1996).

[58] Rostaing, N., Dalmas, S., and Galligo, A. Automatic differentiation in Odyssée. *Tellus Series A 45* (Oct. 1993), 558–+.

[59] Schlumberger. *Eclipse Technical Description 2008.2*. Schlumberger, 2008.

[60] Shiriaev, D., and Griewank, A. ADOL-F automatic differentiation of Fortran codes. In *Computational Differentiation: Techniques, Applications, and Tools* (1996), SIAM, pp. 375–384.

[61] Shroff, G., and Keller, H. Stabilization of unstable procedures; the recursive projection method. *SIAM Journal of Numerical Analysis 30* (1993).

[62] Siek, J. G., and Lumsdaine, A. The matrix template library: Generic components for high-performance scientific computing. *Computing in Science and Engineering 1*, 6 (Nov 1999), 70–78.

[63] Stroustrup, B. *C++ Programming Language*. Addison-Wesley, MA, 2000.

[64] Sutter, H. The concurrency revolution. *C/C++ Users Journal 23* (2005).

[65] Sutter, H. The free lunch is over. *Dr. Dobb's Journal 30* (2005).

[66] Toronyi, R. M., and Ali, S. M. F. Two-phase, two-dimensional simulation of a geothermal reservoir. *SPE Journal 17*, 3 (1977), 171–183.

[67] Vandevoorde, D., and Josuttis, N. M. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, Nov. 2002.

[68] Veldhuizen, T., and Jernigan, M. Will C++ be faster than Fortran? In *1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)* (1997), Lecture Notes in Computer Science. Springer-Verlag.

[69] Veldhuizen, T. L. Arrays in Blitz++. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments* (London, UK, 1998), ISCOPE '98, Springer-Verlag, pp. 223–230.

[70] Veldhuizen, T. L. C++ templates as partial evaluation. In *IN ACM SIGPLAN WORKSHOP ON PARTIAL EVALUATION AND SEMANTICS-BASED PROGRAM MANIPULATION* (1999), ACM Press, pp. 13–18.

[71] Veldhuizen, T. L., and Gannon, D. Active Libraries: Rethinking the roles of compilers and libraries. *ArXiv Mathematics e-prints* (Oct. 1998).

[72] Verma, S., and Aziz, K. SPE 36007, FLEX: An Object-Oriented reservoir simulator. In *SPE Petroleum Computer Conference, Dallas* (June 1996), SPE.

[73] Watson, L. Numerical linear algebra aspects of globally convergent homotopy methods. *SIAM Review 28* (1986).

[74] Watson, L., Billups, S., and Morgan, A. Algorithm 652, HOMPACK, a suite of codes for globally convergent homotopy algorithms. *ACM Transactions on Mathematical Software 13* (1987), 281–310.

[75] Yanenko, N. N. *The method of fractional steps: the solution of problems of mathematical physics in several variables*. Springer, 1971.