MULTISTAGE PRECONDITIONER FOR WELL GROUPS AND AUTOMATIC DIFFERENTIATION FOR NEXT GENERATION GPRS

A REPORT
SUBMITTED TO THE DEPARTMENT OF ENERGY RESOURCES ENGINEERING
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Yifan Zhou
June 2009

I certify that I have read this report and that in my opinion it is fully adequate, in scope and in quality, as partial fulfillment of the degree of Master of Science in Petroleum Engineering.

Prof. Hamdi Tchelepi Principal Co-Advisor

I certify that I have read this report and that in my opinion it is fully adequate, in scope and in quality, as partial fulfillment of the degree of Master of Science in Petroleum Engineering.

Prof. Khalid Aziz Principal Co-Advisor

Preface

This report includes two distinct parts: "Multistage Preconditioner for Well Groups" and "Automatic Differentiation for Next Generation GPRS".

The first part describes the following work: (1) Extension of the CPR-based multistage preconditioner to group constraints on multi-segment (MS) wells; (2) modification of the data structure for adaptively working with the AIM (Adaptive Implicit Method) formulation and well-control switching; and (3) estimation of initial junction pressure for group-rate control and junction-pressure control mode with initial-rate estimates. The extensions are tested with large homogeneous and heterogeneous cases with group constraints. With the above improvements, GPRS is able to solve fully coupled reservoir-facilities simulation including well groups, with either FIM (Fully Implicit Method) or AIM, in an efficient and accurate manner.

The second part describes two extensions of the ADETL (Automatically Differentiable Expression Templates Library) framework, which was conceived and designed by Rami Younis. Three primary stages (building, evaluation, and tearing down) are involved in the automatic generation of gradients behind the scenes. The extensions performed as part of this research effort include: (1) three new allocator options for the expression building stage, and (2) a block-sparse data structure for the expression evaluation stage. Additional improvements have also been implemented and integrated into ADETL. A variety of numerical examples are investigated and comparisons are made between GPRS and ADETL-based simulators. With the new allocator and block-sparse data structure, the overhead of ADETL ranges from -4% to 45%. Automatic differentiation is a flexible and efficient approach for building complex reservoir simulators. The next-generation of GPRS will be based on ADETL.

Acknowledgements

First I would like to present my full appreciation to my advisors Prof. Hamdi Tchelepi and Prof. Khalid Aziz for their encouragement, support and guidance throughout both parts of this work. I profoundly appreciate their suggestions, comments and valuable inputs to this report. I thank both of them for providing me motivation and confidence to dive into these interesting topics.

My thanks also go to Dr. Yuanlin Jiang, who contributed deeply and broadly to the development of GPRS. We had a lot of helpful and insightful discussions on the multistage preconditioning extensions and GPRS data structures. Thanks are also due to Dr. Huanquan Pan, the administrator of GPRS, who is always ready to help.

Moreover, I want to express my sincere gratitude to Dr. Rami Younis, who initiated and developed ADETL framework. The second part of this report would not have been finished without his advice and assistance. A lot of ideas came out from our regular discussions. Also, Dr. Denis Voskov is acknowledged for providing the prototype ADETL-based simulator and initial test cases.

Financial support from the Stanford University Petroleum Research Institute programs on Reservoir Simulation and Advanced Well (SUPRI-B and SUPRI-HW) is gratefully acknowledged.

Last but not least, I want to give my thanks to the faculty, staff and fellow students in this department, for their help in various aspects, ever since I came to Stanford. I owe a lot to the Stanford Chinese Association of Petroleum Engineers(SCAPE), in which I learned a lot and always had a wonderful time. Special thanks are due to Xiaochen Wang for her help and support in coursework, research and daily life, and my parents for their long-term care and inspiration.

Contents

List of Tables

List of Figures

Part I

Multistage Preconditioner for Well Groups

Chapter 1

Introduction

1.1 Background and Motivation

Accurate modeling of wells is an integral part of reservoir flow simulation. As the complexity and cost of advanced (e.g., instrumented multi-lateral) wells grow, there is a growing need to model the flow physics in the wellbore and the tight coupling between the well and the reservoir accurately and efficiently. As part of the effort to simulate flow processes in reservoirs with advanced wells, a model for handling well groups was proposed and implemented in Stanford's General Purpose Research Simulator (GPRS) by Jiang [1]. His treatment provides fully coupled group constraints on Multi-Segment (MS) or standard wells. Jiang also developed multi-level sparse block data structures, and a block Generalized Minimal Residual (GMRES) solver for fully implicit system with a Constrained Pressure Residual (CPR) preconditioner for both MS and standard wells, but without constraints on well groups.

With constraints on groups of wells, strong coupling between the reservoir model, the well group, and the individual wells in the group is introduced into the system. The computational challenge increases in the presence of large numbers of complex multi-segment wells with various operating constraints (pressure, total-rate, liquid-rate, etc.) on individual wells and well groups. As a result, a robust and computationally efficient preconditioner is needed to solve large heterogeneous reservoir models with many advanced wells (e.g., multi-segment wells with constraints on individual

wells and well groups). Moreover, due to the clear advantages of the Adaptive Implicit Method (AIM) and the necessity for switching well controls during a simulation, adaptive and efficient data structures are needed. Furthermore, good initial guesses are essential for both group-rate and junction-pressure constraints in order to ensure fast convergence. In the previous approach, a constant was used as a rough estimate for the initial junction pressure for group-rate control. Moreover, it was not possible to specify a constraint on the junction-pressure.

In order to extend GPRS to handle flexible well-group constraints and the AIM formulation, the CPR-based multi-stage preconditioner [1, 3–5] is extended. Adaptivity is introduced to the solver, preconditioner, and their relevant data structures. Moreover, initial pressure estimation for group-rate control, and the junction-pressure control with initial rate estimates are designed and implemented.

1.2 Multi-Level Sparse Block Data Structure

The Multi-Level Sparse Block (MLSB) data structure is a hierarchical system for storing sparse Jacobians and performing computations. The various levels correspond to the discrete structures of the physical models. The MLSB structure was designed for better abstraction and organization of the system with many relatively independent components including reservoirs, wells (standard, MS well or well group) and surface facilities [1]. In this data structure, a higher level matrix is composed of a number of independent submatrices, the implementation details of which are completely hidden from their host matrix. Therefore, the MLSB structure is efficient and extensible.

As shown in Figure 1.1, the MLSB matrix has three levels: on the first level, we have the global matrix, incorporating all the derivatives of the different governing equations with respect to all the variables. It can be separated into four parts: $\mathbf{J_{RR}}$, $\mathbf{J_{RF}}$, $\mathbf{J_{FR}}$, $\mathbf{J_{FF}}$, where the first subscript represents the equation and the second represents the variable, e.g., $\mathbf{J_{RF}}$ represents the derivatives of the reservoir equations with respect to facility variables; on the second level, the four reservoir and facility matrices have their own structures. In $\mathbf{J_{RR}}$, the diagonal, upper and lower off-diagonal nonzero block entries of the reservoir part are stored one by one. In $\mathbf{J_{RF}}$, $\mathbf{J_{FR}}$ and $\mathbf{J_{FF}}$,

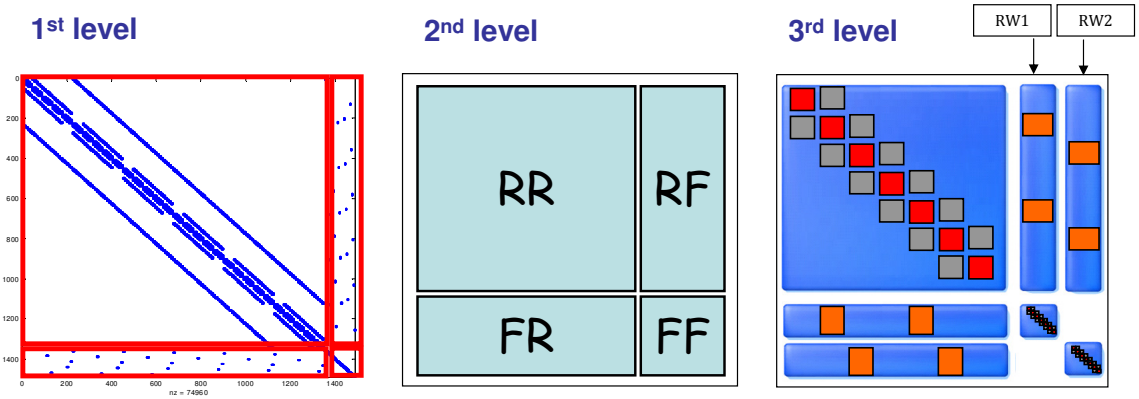


Figure 1.1: Multi-Level Sparse Block Matrix [1]

each submatrix of different size corresponds to one facility, which can be a standard well (1 equation / variable), a Multi-Segment (MS) well (4X equations / variables), or a well group. Then, on the third level, each well matrix (e.g., $\mathbf{J_{RW1}}$,$\mathbf{J_{RW2}}$) also has its own substructure and can be very flexible in both size and format based on the model type.

1.3 CPR Preconditioning

CPR stands for Constrained Pressure Residual, which is a multi-stage preconditioning method developed specifically for reservoir simulation by Wallis et al. [3, 4]. Its motivation comes from the fact that the errors in reservoir simulation problems usually span a large frequency spectrum. The local error (high frequency) can be successfully removed by the solver during the first few iterations, but it is very difficult for Incomplete LU decomposition (ILU) preconditioned Krylov subspace solvers to remove the low frequency error [6]. Such low-frequency effects can be eliminated effectively with multigrid methods [7]. CPR is a two-stage preconditioner, which can be used to combine both multigrid and ILU type preconditioners. The high efficiency of CPR has been proved for reservoir systems with standard and multi-segment wells [1, 8]. The two-stage CPR preconditioner can be expressed as:

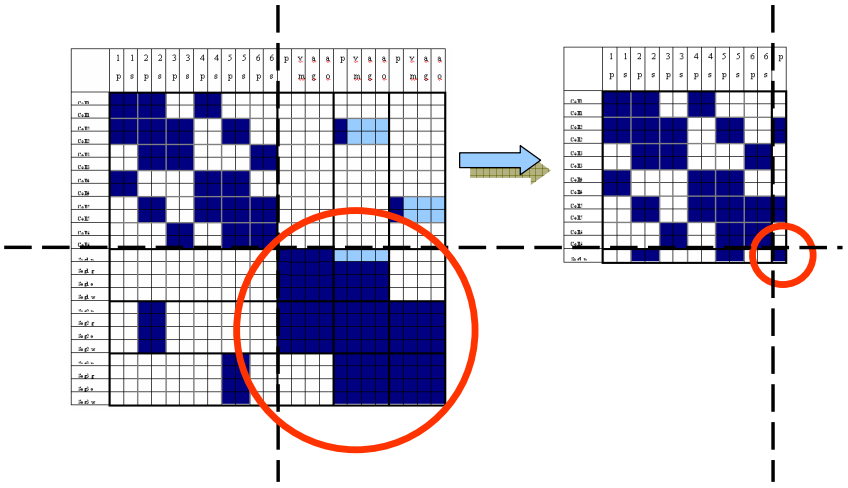


Figure 1.2: Algebraic reduction of MS well to standard well [1]

$$M_{1,2}^{-1} = M_2^{-1}[I - AM_1^{-1}] + M_1^{-1}, \tag{1.3.1}$$

with its first stage working globally on a pressure-like system:

$$M_1^{-1} = PA_P^{-1}R, \tag{1.3.2}$$

and its second stage working locally on the overall system:

$$M_2^{-1} = [B]ILU(A). \tag{1.3.3}$$

1.3.1 First stage: global pressure-like system

The traditional CPR method was applied to problems with standard wells [3–5]. In order to extend CPR to handle the Multi-Segment (MS) well model, algebraic reduction can be applied to the full system of equations such that MS wells are approximated as standard wells with a single unknown, namely pressure. This approach is described in detail by Jiang [1] and Jiang & Tchelepi [8]. Figure 1.2 illustrates this reduction process.

The steps of the CPR first stage for dealing with MS wells are: (1) perform

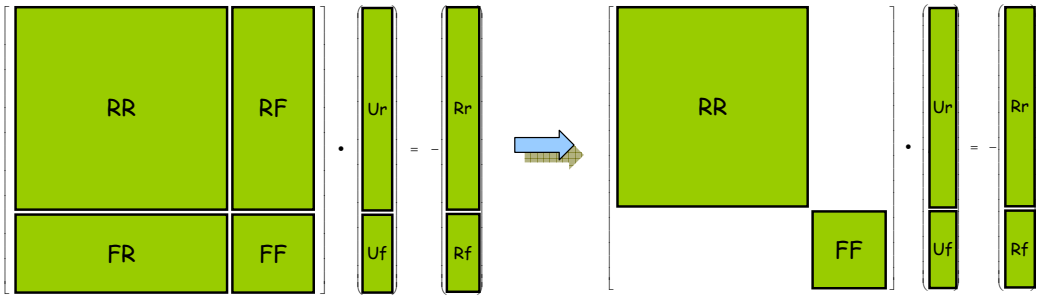


Figure 1.3: Decoupling property of system matrix [1]

algebraic reduction on each MS well to an approximate standard well; (2) use true IMPES reduction [1, 5] on the reservoir part to form a pressure-like system; (3) solve the near-elliptic system of the coupled reservoir model with standard-like wells with an AMG preconditioner [9]; (4) update the residual vector using the computed correction.

1.3.2 Second stage: local preconditioner applied to the overall system

In the second stage, the various Jacobian components are treated separately using a local preconditioner. Specifically, the following strategy is used:

- Use **BILU(0)** (a blockwise incomplete LU factorization preconditioner with no fill-in) [10] to solve the reservoir part: $\mathbf{J_{RR}} \cdot \mathbf{U_r} = \mathbf{R_r}$;
- Use **BILU(1)** (blockwise ILU with one fill-in level) to solve each submatrix in the facility matrix FF: $\mathbf{J_{FF_i}} \cdot \mathbf{U_{f_i}} = \mathbf{R_{f_i}} \quad (i = 1, 2, ..., N_f)$, where N_f is the number of facilities (i.e., submatrices in FF).

Blockwise ILU provides improved stability (only requires invertible diagonal blocks) and better memory utilization (less data fetching) than the pointwise ILU. This combination of the BILU preconditioners (level zero for the reservoir model and level one for multi-segment wells and facilities) is proved to be an efficient choice for the second stage of CPR when solving coupled reservoir-facilities system with advanced wells (i.e., multi-lateral, multi-segment) [1].

Chapter 2

Theory and Methods

2.1 CPR Preconditioner for Well Groups

2.1.1 Well-group model in GPRS

The well group model in GPRS uses a fully implicit treatment of the relationship between the group constraint and the rates for individual wells. This is different from a guide-rate, which is specified by the user, or calculated according to well production potential from the last timestep. The guide-rate approach is employed in some simulators [11].

In GPRS, the WellGroup class is inherited directly from the common Ifacility class, just like the StdWell and MSWell classes. The original CPR preconditioner in GPRS could only handle the preconditioning data associated with each facility in the form of one column in the $\mathbf{J_{RF}}$ matrix, one row in the $\mathbf{J_{FR}}$ matrix, and one element in the $\mathbf{J_{FF}}$ matrix, as provided by the StdWell and MSWell classes (through algebraic reduction). Of course, one can reduce the preconditioning data of a well group to that of an approximate standard well by algebraic reduction. Then, the original CPR preconditioner would be able to construct a pressure-like system. However, if we have several multi-segment wells in a single well group, especially when we consider the possibility of nested well groups (e.g., several well groups under the control of a higher-level well group), the above procedure will still reduce the entire group into

a single equation with a single unknown variable, and such a procedure is likely to cause considerable error in the computed rates of individual wells in the group and impact the quality of the preconditioner negatively.

Therefore, the well groups should be handled in a more accurate way in the CPR preconditioner. One method is to represent each of the N wells in the group without the group constraint, i.e., the preconditioner treats individual standard or multi-segment wells in the group as independent wells, each with a pressure unknown and a rate constraint that is calculated in the same way as in the guide-rate approach. Then the preconditioning data will contain N columns in the $\mathbf{J_{RF}}$ matrix, N rows in the $\mathbf{J_{FR}}$ matrix and N diagonal elements in the $\mathbf{J_{FF}}$ matrix.

This could be further extended to nested well groups, where N becomes the sum of the number of individual wells contained in the base well group and the total number of wells in all of the nested well groups. This approximation is expected to be much better than the treatment that reduces the entire group to a single equation and variable.

In a well group, a "junction" is defined as the point where the wells are joined and where the constraint is applied. In group-rate control mode, where the total oil/water/gas/liquid rate of all wells in the group is set to a fixed amount, the junction constraint introduces $N + 1$ new variables (one junction pressure P_J, and N well rates, Q_i), which correspond to the same number of extra equations (one constant-rate control and N pressure relation equations between the well reference pressure and the junction pressure):

$$R_0 = \sum_{i=1}^{N} Q_i - Q_T = 0, \tag{2.1.1}$$

$$R_i = P_i^w - P_J - \Delta P(Q_i) = 0 \quad (i = 1, 2, ..., N), \tag{2.1.2}$$

where Q_T is the specified total rate for the well group, p_i^w is the reference pressure (well-head pressure in the MSWell model) for well i, and $\Delta P(Q_i)$ is the pressure correlation for the pipelines between the well-heads and the junction, which can be expressed as:

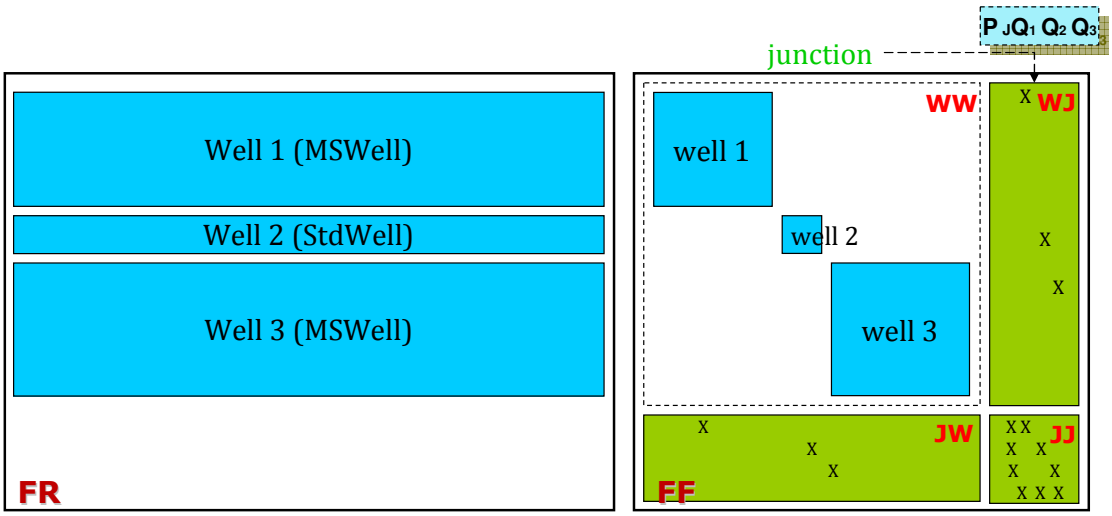


Figure 2.1: Equation pattern for a typical well group model [1]

$$\Delta P(Q_i) = \left(\frac{c \cdot f_{tp} \cdot \rho \cdot L \cdot Q_i^2}{D^5} \right), \tag{2.1.3}$$

where c is a unit conversion constant, f_{tp} is the dimensionless Darcy friction factor, ρ is the standard density of the phase to which the rate constraint is applied, L is the length of the pipeline from the well-head to the junction, and D is the diameter of the pipe.

The non-zero incidence matrix for a typical well group model including two MSWells and one StdWell in the $\mathbf{J_{FR}}$ and $\mathbf{J_{FF}}$ matrices is shown in Figure 2.1. In a sub matrix for a well group in the $\mathbf{J_{FF}}$ matrix, the $\mathbf{J_{WW}}$ part includes N block matrices, which contain derivatives of well equations with respect to well variables, of individual wells. The $\mathbf{J_{WJ}}$, $\mathbf{J_{JW}}$ and $\mathbf{J_{JJ}}$ parts are specially for the well group model, where the subscript "J" represents junction, e.g., the $\mathbf{J_{JW}}$ part assembles the derivatives of the junction-pressure relation and group constraint equations with respect to well variables.

2.1.2 Extended first stage: global on pressure-like system

The key idea of the first stage of the CPR preconditioner (implemented in GlobalCPRPre class) for well groups is to temporarily ignore the group constraint and disaggregate the group into N individual wells, each with a pressure unknown and a rate constraint calculated according to well production potential from the last iteration. The detailed procedure is as follows:

1. For each well group, let all facilities contained in it generate their reduced $\mathbf{J_{FR}}$, $\mathbf{J_{RF}}$ and $\mathbf{J_{FF}}$ parts:
   - For each MS well, use algebraic reduction to form one equation relating to only pressure unknowns and one variable (well-head pressure) of an equivalent standard well.
   - For each standard well, keep only pressure derivatives in its equation and keep its well pressure variable.
2. Use standard true IMPES reduction to reduce the reservoir matrix ($\mathbf{J_{RR}}$ part).
3. Assemble the reduced matrices to form the pressure system $A_p \delta p = r_p$.
4. Solve the reduced system with AMG preconditioner for pressure update δ_p.
5. Perform prolongation: $\delta u_1 = C^T \delta p$.
6. Update residual vector: $r = r - A\delta u_1$.

2.1.3 Extended second stage: local on overall system

To support the well-group model, the second stage of the CPR preconditioner (implemented in ResFacBILU class) contains the following steps:

1. Use the **BILU(0)** preconditioner to solve the reservoir part: $\mathbf{J_{RR}} \cdot \mathbf{U_r} = \mathbf{R_r}$ (the $\mathbf{J_{RF}}$ part is decoupled from the $\mathbf{J_{RR}}$ in the first stage through the residual vector update)

2. In a sub matrix for a well group in $\mathbf{J_{FF}}$, solve $\mathbf{J_{JJ}} \cdot \mathbf{U_j} = \mathbf{R_j}$ using pointwise ILU (the $\mathbf{J_{JW}}$ part is dropped due to pressure decoupling in the first stage)

3. Use the **BILU(1)** preconditioner to solve each sub matrix in $\mathbf{J_{WW}}$ (e.g., well 1, 2 and 3 in Figure 2.1): $\mathbf{J_{WW_i}} \cdot \mathbf{U_{w_i}} = \mathbf{R_{w_i}}$ $(i = 1, 2, ..., N_w)$, where N_w is the number of individual wells in the well group (the $\mathbf{J_{FR}}$ part is also dropped due to pressure decoupling in the first stage).

2.1.4 Data structure modification for preconditioning

In order for the preconditioner to correctly recognize a well group, several new variables are added to the Ifacility class (base class of all facility models), as shown in Table 2.1. Correspondingly, the member function "getPreMatInfo", which fetches the preconditioning data of one facility is changed to include these three new variables in its parameter list. The values of these new variables are set during the initialization stage of a facility. Then the linearization function of the WellGroup class collects the necessary preconditioning data from individual wells and assembles them into the corresponding matrices, which have the same data structure as the preconditioning matrices of StdWell and MSWell models. The CPR preconditioner can then recognize these matrices from the well-group model and interpret them as preconditioning data for N separate wells with the help of the three new variables. In this way, the first two steps in the pressure solution stage of the CPR preconditioner are realized, and the remaining steps can be applied in exactly the same manner as for StdWell and MSWell models.

2.2 Adaptive Data Structures

Originally, the block GMRES Solver, which is the preferred solver option in GPRS, was not compatible with the AIM formulation, since it could not deal with dynamically changing data structures in the multi-level sparse block matrix. For the same reason, the block GMRES solver could not handle switching the well constraint of a

Table 2.1: New variables introduced into Ifacility for a well group

Name	Type	Meaning	Value
nTWells	int	total number of wells in a facility	1 for StdWell and MSWell; N for well group
mnWellEqns	int*	number of governing equations for each well	1 integer (= nObjEqns) for StdWell and MSwell; N integers for well group
mSubPerf	int*	number of perforations for each well	1 integer (= nPerfs) for StdWell and MSwell, N integers for Well group

standard well from rate control (1 extra equation / variable) to Bottom Hole Pressure (BHP) control (no extra equation / variable).

In order to accommodate dynamic changes to the data structure, the following modifications have been made to the following multi-level sparse block matrix classes:

- SysFFJWrapper class (for $\mathbf{J_{FF}}$ matrix and $\mathbf{J_{WW}}$ part in a well group)
- SysFRJWrapper class (for $\mathbf{J_{FR}}$ matrix)
- SysRFJWrapper class (for $\mathbf{J_{RF}}$ matrix)
- CoordMatWrapper class (for $\mathbf{J_{JW}}$, $\mathbf{J_{WJ}}$ parts in a well group and the blank parts of the $\mathbf{J_{FR}}$ and $\mathbf{J_{RF}}$ matrices)

For the first three matrix assembly classes, the following operations are added for adaptivity:

- Remove a submatrix from the assembly, for switching well controls:
  void DelSubmatrix (int ib); // Remove the submatrix *ib* from the wrapper
- Recalculate their size and reconstruct the temporary column vector for matrix-vector multiplication: void calcSize();

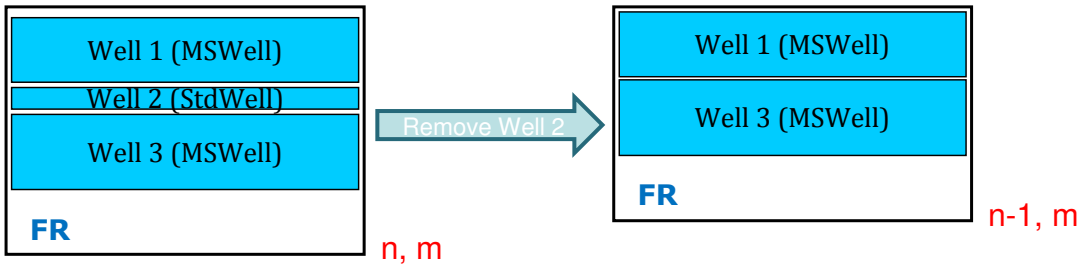


Figure 2.2: Sub-matrix removal and size recalculation

The above procedure is demonstrated in Figure 2.2, in which the equation of a standard well is removed from the $\mathbf{J_{FR}}$ matrix, and the size of the matrix is recalculated. For the sparse coordinate matrix (CoordMatWrapper) class, the following operation is added for adaptivity:

- Change the dimension without reconstructing the object, in order to provide the $\mathbf{J_{FR}}$ and $\mathbf{J_{RF}}$ matrices with the flexibility to adapt to changes in size:
  void changedim(int nRows, int nCols); // Change dimension to nRows x nCols

The above operations are applied to the Block GMRES solver in order to modify the data structure when GPRS detects a changing requirement in the structure of the Jacobian matrix, either in the reservoir, due to the AIM formulation, or in the facilities, due to changes in the well controls. Corresponding changes are then applied to the structure of the wrapper matrices, i.e., to delete unnecessary matrices, set implicit variables of sub matrices in SysFFJWrapper, SysFRJWrapper, SysRFJWrapper, or change the dimension of CoordMatWrapper in order to give it the correct size. Finally the size of the global matrix (first level in MLSB matrix, including $\mathbf{J_{RR}}$, $\mathbf{J_{RF}}$, $\mathbf{J_{FR}}$ and $\mathbf{J_{FF}}$ matrices) is recalculated.

With the above adaptive approach, the block GMRES solver with the MLSB matrix structure can handle AIM formulation as well as switching of well constraints correctly and efficiently.

2.3 Well-Group Model Enhancements

2.3.1 Initial junction-pressure estimation for group-rate control

Originally, the junction pressure was set to a constant value in the implementation of the well-group model in GPRS, which is a very rough estimate and not accurate for most cases. Since there is some threshold on the Newton update of junction pressure, i.e., the maximum changing rate in a time step, it would take a long time to find the proper junction pressure if the initial guess is far from the correct value. Sometimes, a converged solution for the system cannot be obtained using a constant junction pressure as the initial guess. As a result, a more rigorous initial guess is needed to compute a converged solution in an efficient manner.

First consider the extra governing equations of a well-group model in the group-rate control mode:

$$P_i^w - P_J - \left(\frac{c \cdot f_{tp} \cdot \rho \cdot L \cdot Q_i^2}{D^5}\right) = 0 \quad (i = 1, 2, ..., N), \quad \sum_{i=1}^{N} Q_i - Q_T = 0. \quad (2.3.1)$$

If P_i^w is taken as a known coefficient, which is actually computed using the "calcStartupProps" function in each iteration, these N+1 equations (2.3.1) can be solved for the vector containing the N+1 variables: $\mathbf{X^k} = [P_J, Q_1, Q_2, ..., Q_N]^T$. The residual vector $\mathbf{R^k}$ can be written as follows:

$$\mathbf{R^k} = \begin{bmatrix} P_J + c\rho \cdot \left(\frac{f_{tp} \cdot L}{D^5}\right)_1 \cdot Q_1^2 - P_1^w \\ P_J + c\rho \cdot \left(\frac{f_{tp} \cdot L}{D^5}\right)_2 \cdot Q_2^2 - P_2^w \\ \vdots \\ P_J + c\rho \cdot \left(\frac{f_{tp} \cdot L}{D^5}\right)_N \cdot Q_N^2 - P_N^w \end{bmatrix}. \quad (2.3.2)$$

Using Newton's method, the Jacobian matrix $\mathbf{J^k}$ (blank entries are zeros) is constructed as follows:

$$\mathbf{J^k} = \begin{bmatrix} 1 & 2c\rho \cdot \left(\frac{f_{tp} \cdot L}{D^5}\right)_1 & & & \\ 1 & & 2c\rho \cdot \left(\frac{f_{tp} \cdot L}{D^5}\right)_2 & & \\ \vdots & & & \ddots & \\ & -1 & -1 & \cdots & -1 \end{bmatrix}. \tag{2.3.3}$$

In each Newton iteration, perform the steps below:

1. Set the well rate to Q_i^k (the solution from the last iteration).
2. Call the "calcStartupProps" function to calculate the initial reference pressure P_i^w.
3. Construct the residual vector, $\mathbf{R^k}$, and the Jacobian matrix, $\mathbf{J^k}$.
4. Check convergence: if $|\mathbf{R^k}| < \epsilon$, converged results are obtained; otherwise, go to step 5.
5. Solve $\mathbf{J^k}\delta_{\mathbf{X}}^{\mathbf{k+1}} = -\mathbf{R^k}$.
6. Perform the Newton update: $\mathbf{X^{k+1}} = \mathbf{X^k} + \delta_{\mathbf{X}}^{\mathbf{k+1}}$, and go to the next iteration.

When the system converges, i.e. $|\mathbf{R^k}| < \epsilon$, we obtain an accurate initial pressure and reasonable initial rates for each well in the group. This is especially important if the individual well configurations are very different from each other, since the scheme computes a distinct initial rate for each well instead of setting the rate to be the same for all the wells in the group (Q_T/N).

2.3.2 Junction-pressure control with initial rate estimates

GPRS has been extended to use the junction pressure to control a well group. Under this control mode, instead of changing the extra equation from the total-rate constraint ($\sum_{i=1}^{N} Q_i - Q_T = 0$) to a pressure constraint ($P_J = P_{const}$), the system actually removes the extra P_J variable and the constraint equation. P_J is considered as a constant and is moved to the right hand side of the system of equations. Hence the number of additional variables and equations for a well group becomes N, instead of $N + 1$. As shown in Figure 2.1, the group constraint equation is the last one while the junction-pressure variable is the first one. Note that a zero pivot will appear in the ILU decomposition if the constraint equation $P_J = P_{const}$ is not removed under junction-pressure control.

In order to give reasonable initial guesses to the well rates under junction-pressure control, the following algorithm is implemented in the WellGroup class, where for each well i in the group, we perform the following operations:

1. Give an initial guess for the well rate Q_i^0, and set $k = 0$.
2. Set the rate of well i to Q_i^k, and call the "CalcStartupProps" function to calculate the initial well-head pressure, P_i^w.
3. Calculate $r_i^k = P_J - P_i^w + c\rho \cdot \left(\frac{f_{tp} \cdot L}{D^5}\right)_i \cdot Q_i^2$.
4. Check convergence: If $r_i^k < \epsilon$, the initial rate for well i is determined. Let $i = i + 1$, and go to step 1, until $i = N$; otherwise, go to step 5.
5. Calculate $Q_i^{k+1} = \sqrt{(P_i^w - P_J) / \left(c\rho \cdot \left(\frac{f_{tp} \cdot L}{D^5}\right)_i\right)}$.
6. Let $k = k + 1$, and go to step 2 for the next iteration.

In this way, the initial rate for all the wells in the group can be estimated with the condition $|r_i| = \left|P_J - P_i^w + c\rho \cdot \left(\frac{f_{tp} \cdot L}{D^5}\right)_i \cdot Q_i^2\right| < \epsilon, \quad 1 \leq i \leq n$ satisfied. Since the initial reference pressure for a multi-segment well is not a simple function of the well rate, we cannot get an explicit form for $\frac{\partial P_i^w}{\partial Q_i}$. This is the reason why Newton's method

is not used to obtain the solution and also why P_i^w is considered as a coefficient during the initial pressure estimation for a group-rate control.

Chapter 3

Numerical Examples

Six numerical test cases are presented in this chapter. The comparison includes three solution schemes using the same solver (block Generalized Minimal Residual (GMRES) solver) but different formulations (the Fully Implicit Method (FIM), or Adaptive Implicit Method(AIM)), and different preconditioners (Block Incomplete LU decomposition (BILU), or Constrained Pressure Residual(CPR) preconditioner). The CPR preconditioner applied here uses Algebraic Multigrid(AMG) in the first stage and BILU in the second stage preconditioners. The three schemes are:

- FIM, Block GMRES Solver, BILU Preconditioner
- FIM, Block GMRES Solver, CPR (AMG+BILU) Preconditioner
- AIM, Block GMRES Solver, CPR (AMG+BILU) Preconditioner

3.1 Base Case

Case Description: (see Figure 3.1)

- 20x20x1, oil-water system
- 1 standard injector with BHP control at 7000psia
- 3 multi-segment producing wells, each with 5 segments and a single completion

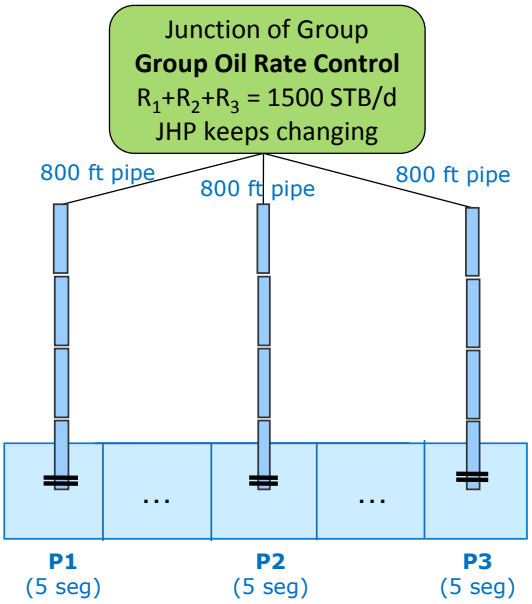


Figure 3.1: Schematic diagram for test case 1

- All producers are in a well group under a group oil-rate control of 1500 STB/day
- Run for 365 days with a maximum time step of 1 day

The simulation results (oil production rate and junction pressure history) for this base case are given in Figure 3.2. We can see that the CPR results match the BILU results very well and the AIM solution is also close to both of them.

Table 3.1 shows the efficiency comparison. Compared with BILU, the extended CPR preconditioner improves the overall performance significantly. The new CPR approach decreases the solver iterations by 82.7% and solver time by 51.0%. For this 2-phase, 2-component small case, AIM does not have a large impact on the performance. However, the AIM formulation is expected to provide significant improvements when the number of components and the size of the problem increase.

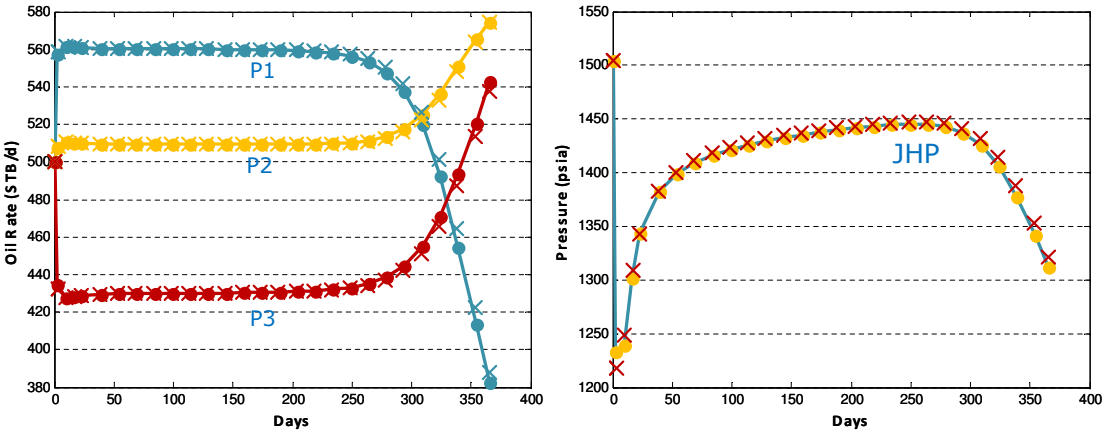


Figure 3.2: Oil rates and junction pressure comparison for test case 1

Table 3.1: Efficiency comparison for test case 1

Solver Option	FIM, BILU	FIM, CPR	AIM, CPR
Time Step	370	370	371
Newton Iteration	3962	3402	3458
Solver Iteration	79074	13677	13662
Pressure Iteration	0	13677	13662
Solver Time (sec)	32.18	15.76	12.34
Total Time (sec)	58	39	37

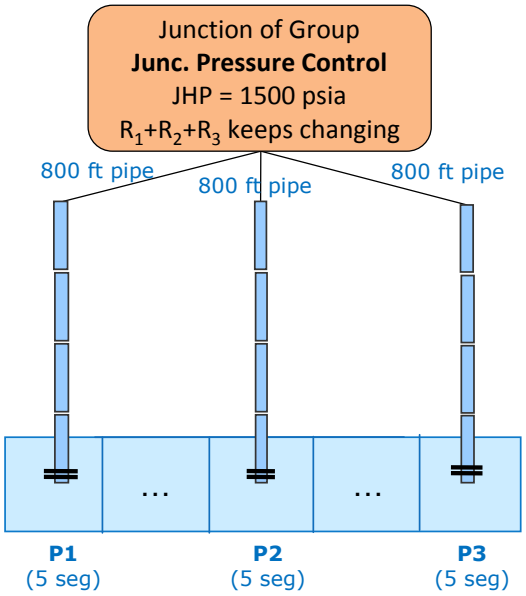


Figure 3.3: Schematic diagram for test case 2

3.2 Larger Problem Size, Junction Pressure Control, Well Control Switch

Case Description: (see Figure 3.3)

- 80x80x1 grid blocks, oil-water system
- 1 standard injector with water-rate control of 2100 STB/day and a BHP backup constraint of 8000 psi
- 3 multi-segment producers, each with 5 segments and a single completion
- All producers are in a well group under a junction-pressure control at 1500 psi
- Run for 365 days with maximum time step of 1 day

The simulation results (oil production rate and junction pressure history) are shown in Figure 3.4. We can see that the junction-pressure control with initial rate estimates works well. Well-control switching can also be properly handled using the adaptive data structure.

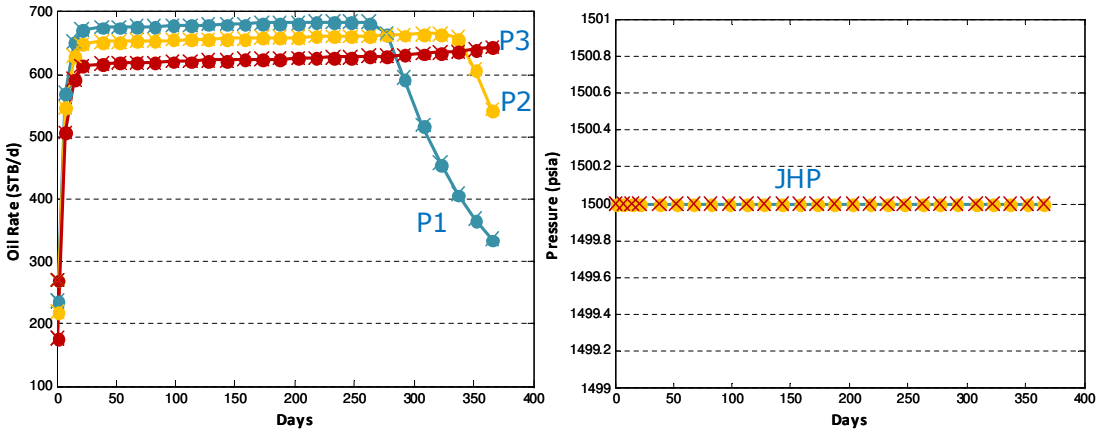


Figure 3.4: Oil rates and junction pressure comparison for test case 2

Table 3.2: Efficiency comparison for test case 2

Solver Option	FIM, BILU	FIM, CPR	AIM, CPR
Time Step	372	372	372
Newton Iteration	1843	1831	1821
Solver Iteration	137646	6304	6123
Pressure Iteration	0	6304	6123
Solver Time (sec)	1358.76	108.66	81.60
Total Time (sec)	1446	196	174

Table 3.2 shows the efficiency comparison for this larger case, in which the extended CPR method has a much larger impact on the solver iterations and time. The new scheme saves 95% in solver iterations and 92% in solver time. The AIM formulation has a more obvious impact in this case. It further decreases the solver time by 24.9% from that obtained by extended CPR alone, which is a considerable improvement.

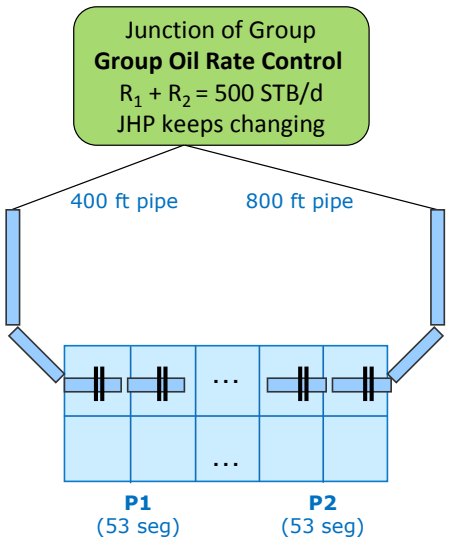


Figure 3.5: Schematic diagram for test case 3

3.3 3D Gas-oil System with High Initial Junction Pressure

Case Description: (see Figure 3.5)

- 50x50x2 grid blocks, gas-oil system
- Primary depletion
- 2 multi-segment producers, each with 53 vertical / tilted / horizontal segments with perforations in every block along the x-direction (50 completions)
- Both producers are in a well group under a group oil-rate control at 500 STB/d
- Run for 200 days with a maximum time step of 1 day

The simulation results (oil production rate and junction pressure history) for this 3D gas-oil test case are given in Figure 3.6. We can see that initial junction-pressure estimation works well in this case, where the estimated value is far from the constant value (1449.43psia) used in the previous implementation. Also, the block GMRES

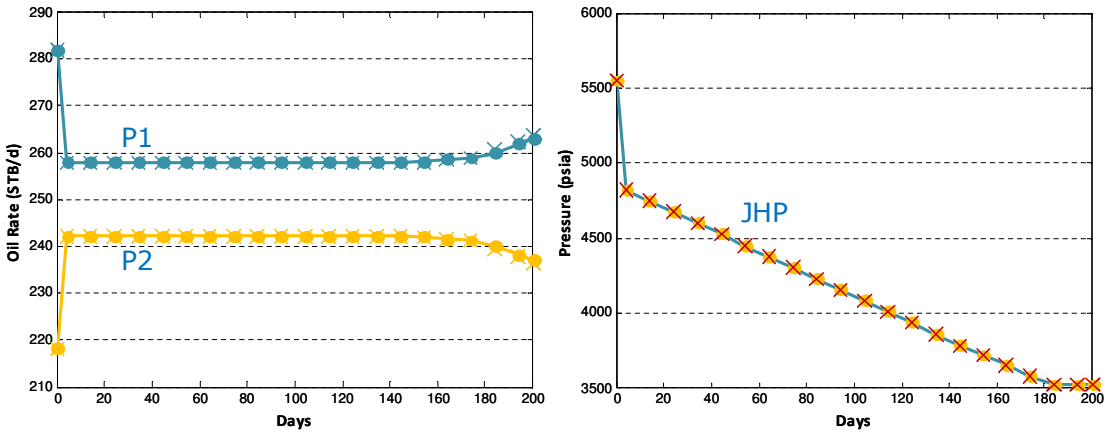


Figure 3.6: Oil rates and junction pressure comparison for test case 3

Table 3.3: Efficiency comparison for test case 3

Solver Option	FIM, BILU	FIM, CPR	AIM, CPR
Time Step	206	206	206
Newton Iteration	527	530	529
Solver Iteration	35872	5385	5403
Pressure Iteration	0	5385	5403
Solver Time (sec)	283.80	63.21	48.71
Total Time (sec)	323	102	89

solver with the AIM formulation and extended CPR preconditioner works well for this 3D system with a gas-phase presence.

Table 3.3 shows the efficiency comparison. Considerable improvement in solver performance is observed. As expected, the AIM formulation reduces the solver time by 22.94% compared to FIM.

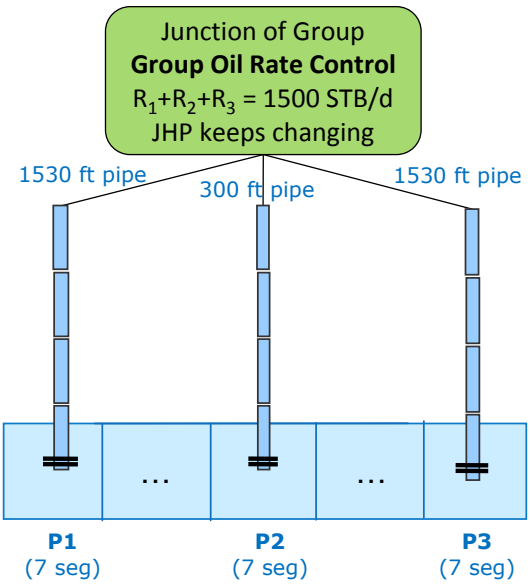


Figure 3.7: Schematic diagram for test case 4

3.4 Longer Simulation Time and Larger Time Step Size

Case Description: (see Figure 3.7)

- 110x110x1 grid blocks, oil-water system
- 1 standard injector with BHP control of 4900 psi
- 3 multi-segment producers, each with 7 segments and a single completion
- All producers are in a well group under a group oil-rate control at 1500 STB/d
- Run for 8000 days with maximum time step of 30 days for the first 4000 days and 20 days for the remaining 4000 days

The simulation results (oil production rate and junction pressure history) for this case with a longer simulation time and larger time step sizes are shown in Figure 3.8. Table 3.4 shows the efficiency comparison. Similar observation of the improvement in solver performance (89.9% time reduction using extended CPR, and 28.1% further

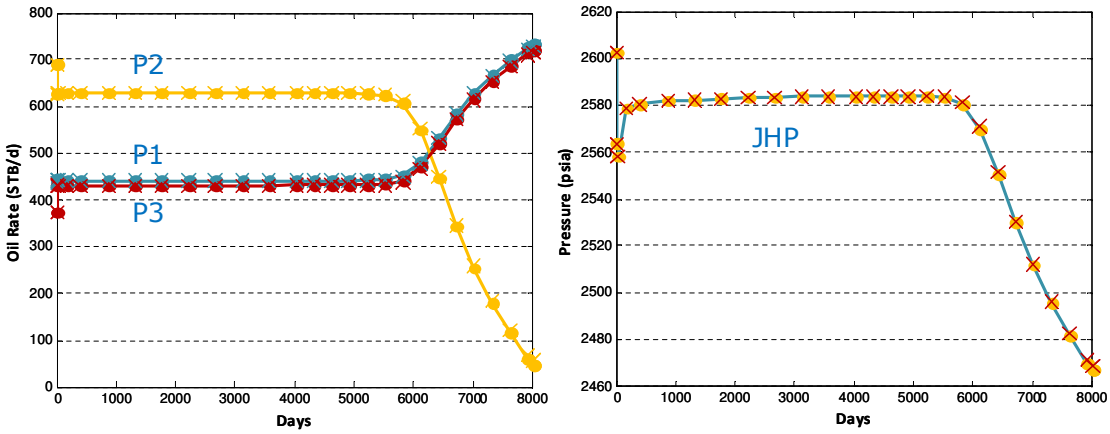


Figure 3.8: Oil rates and junction pressure comparison for test case 4

Table 3.4: Efficiency comparison for test case 4

Solver Option	FIM, BILU	FIM, CPR	AIM, CPR
Time Step	349	349	349
Newton Iteration	1191	1499	1340
Solver Iteration	105327	8661	8049
Pressure Iteration	0	8661	8049
Solver Time (sec)	2209.75	223.55	160.72
Total Time (sec)	2346	370	311

reduction using AIM formulation) can be made and the stability of the solver and preconditioner is validated.

3.5 Large 3D Homogeneous Reservoir - 400000 Cells

Case Description: (see Figure 3.9)

- 200x200x10 grid blocks, oil-water system
- 1 standard injector with water rate control of 1500 STB/d

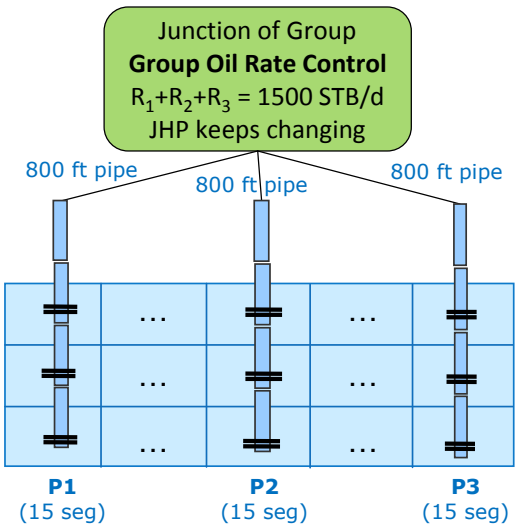


Figure 3.9: Schematic diagram for test case 5

- 3 multi-segment producers, each with 15 segments with perforations in every block along the z-direction (10 completions)
- All producers are in a well group under a group oil-rate control of 1500 STB/d
- Run for 365 days with maximum time step of 1 day (first two settings) and 10 days (third setting)

The simulation results (oil production rate and junction pressure history) for this large 3D test case are plotted in Figure 3.10. Table 3.5 shows the efficiency comparison. For this case, BILU is too slow and cannot get converged solutions, while FIM with large time step size has considerable time truncation error and cannot obtain accurate solutions, both of which are unacceptable. In the first two settings (small time step size) shown here, such a large system can be solved in 14.12 hours with the CPR preconditioner under FIM formulation (or 12.57 hours under AIM formulation) on a single 2.2GHz CPU with 8GB RAM. While in the third setting (AIM with large time step size), the simulation time can be further decreased to 5 hours, which is more accurate and efficient than FIM. This improved performance is due to the powerful combination of our extended CPR and the use of the AIM formulation.

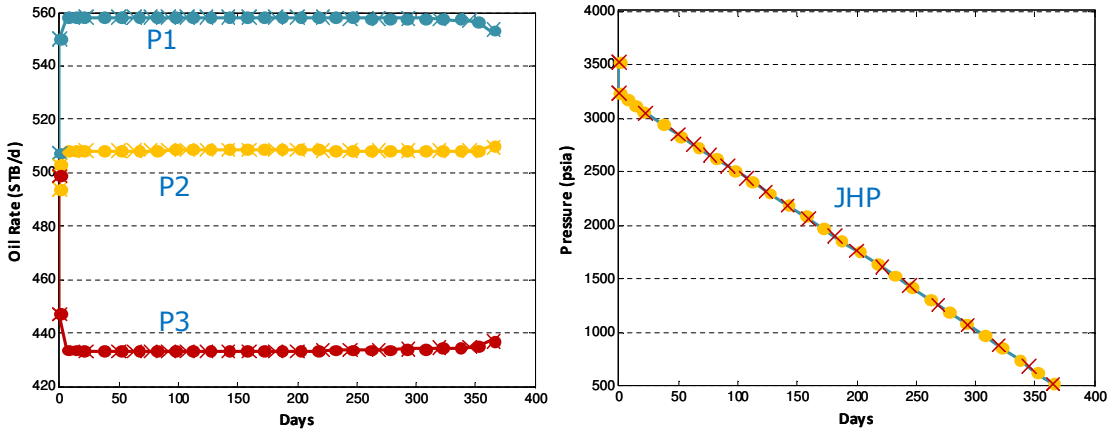


Figure 3.10: Oil rates and junction pressure comparison for test case 5

Table 3.5: Efficiency comparison for test case 5

Solver Option	FIM, BILU	FIM, CPR	AIM, CPR
Newton Iteration	4106	3738	1507
Solver Iteration	20682	19251	8445
Pressure Iteration	20682	19251	8445
Solver Time (sec)	45202.4	39545.0	15817.8
Total Time (sec)	50833	45261	18040

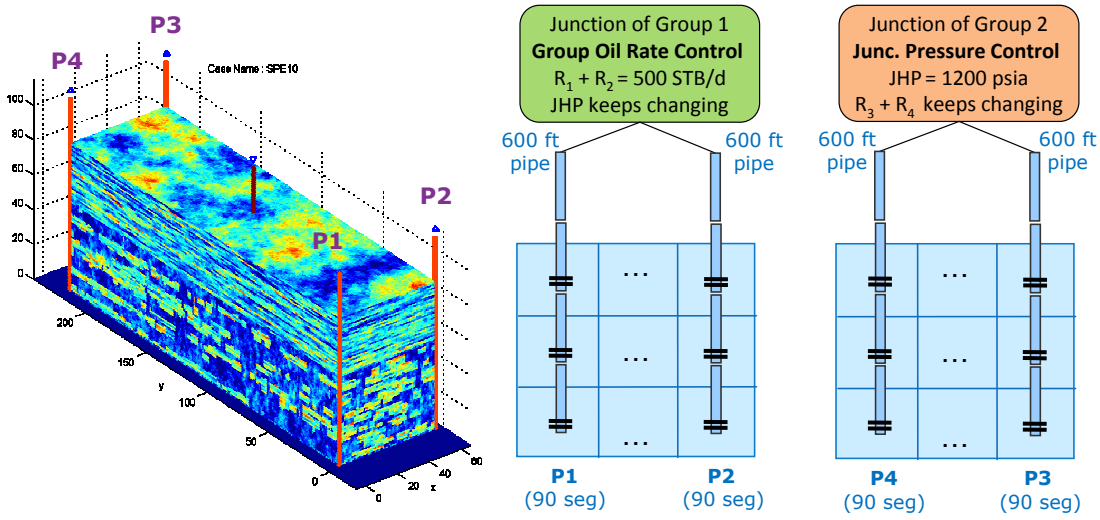


Figure 3.11: Schematic diagram for test case 6

3.6 Full SPE10 with Group Constraints on Multi-segment Wells

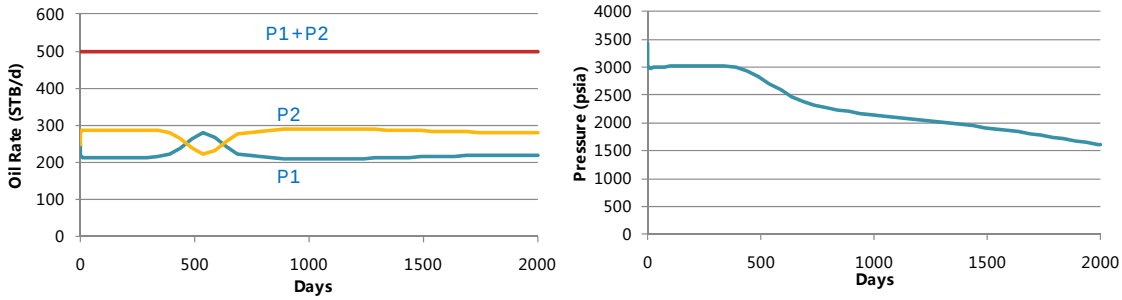
Case Description: (see Figure 3.11)

- 60x220x85 (totally 1122000) grid blocks, oil-water system
- 1 standard injector with BHP control of 8100 psia
- 4 multi-segment producers, each with 90 segments with perforations in every block along z-direction (85 completions)
- Producers 1 and 2 are under a group oil-rate control of 500 STB/day
- Producers 3 and 4 are under a junction-pressure control of 1200 psia
- Run for 2000 days (0.8375 PVI) with a maximum time step of 50 days

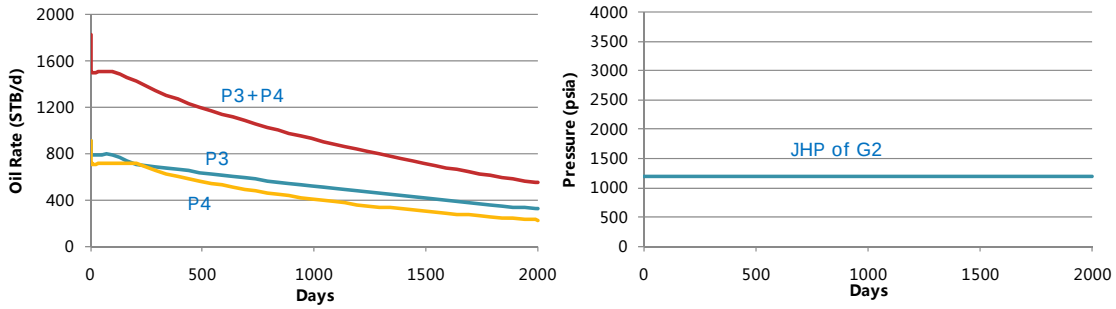
SPE10 is one of the latest comparative solution projects organized by the Society of Petroleum Engineers (SPE) [12]. The case is designed for comparing upscaling

approaches and is also widely used as a benchmark case for large-scale highly heterogeneous reservoir simulation, in which various flow patterns including high permeability channels are involved. Here, we also introduce strong coupling due to the group constraints on multi-segment wells, making it a very challenging test case.

Reasonable results for the two well groups are obtained and shown in Figure 3.12. Table 3.6 shows the efficiency of the solution using the extended CPR preconditoner with the FIM formulation. It took GPRS 16 hours to solve this problem on a single 2.2GHz CPU with 8GB RAM. In this large heterogeneous case, the AIM formulation does not work well with the current time stepping and implicitness labeling strategy. We can expect further efficiency enhancement with a better AIM strategy.



(a) Oil rates and Junction Pressure of Well Group 1



(b) Oil rates and Junction Pressure of Well Group 2

Figure 3.12: Simulation results of test case 6

Table 3.6: Efficiency comparison for test case 6

Solver Option	FIM, CPR
Time Step	62
Newton Iteration	1199
Solver Iteration	12617
Pressure Iteration	12617
Solver Time	27285.9
Total Time	60602

3.7 Concluding Remarks

From the comparisons in this chapter, we can see that the new CPR preconditioner can greatly improve the solver performance compared with using the BILU preconditioner, especially for large-scale problems. If the AIM formulation is used instead of FIM, the simulation time can be further reduced. This is also more obvious for large problems. Moreover, the simulation results using the CPR preconditioner match those using only the BILU preconditioner, and the results using both AIM formulation and CPR preconditioner are close to those using the FIM formulation. The efficiency and accuracy of the new CPR preconditioner and the AIM-enabled block GMRES solver are validated.

Part II

Automatic Differentiation for Next Generation GPRS

Chapter 4

Automatic Differentiation for Reservoir Simulation

4.1 Motivation

With growing needs for accurate modeling of subsurface flow in heterogeneous reservoirs and increasing computational power, more and more physical, chemical, and geomechanical phenomena must be incorporated in general-purpose reservoir simulators. As a consequence, the nonlinear governing equations grow in size and complexity.

The numerical solution methods of these governing equations can be classified into two main categories: explicit and implicit methods (including fully, partial and adaptive implicit methods). To ensure stability, explicit methods would often lead to prohibitively small time steps. Fully implicit methods, which are unconditionally stable, are generally required in this situation. In order to obtain the solution to the coupled nonlinear system of equations using implicit methods, Newton's method is widely used, since it is perhaps the best known method for finding successively better approximations to the solutions of nonlinear systems of conservation laws.

In addition to the linear solution of the linearized system in each Newton iteration, the major effort of applying Newton's method can be divided into three parts: (1) implementation of the discretized residual equations; (2) derivation of the analytical derivatives of the governing equations with respect to the independent variables; (3)

implementation of these derivatives and assembling them into a sparse Jacobian matrix. Once the discretized nonlinear equations are implemented in residual form, the residual vector can be computed using the solution from the last Newton iteration. As for Jacobian generation, hand (manual) differentiation is the most common approach in today's commercial and publicly available simulators [5, 11, 13]. However, generation of the Jacobian matrix using hand (manual) derivatives is both tedious and error prone.

Extendability to a wide range of formulations, recovery processes, and solution algorithms is an important objective of general purpose simulators [14–16]. This is also highly emphasized in our in-house General Purpose Research Simulator (GPRS) [1,5]. However, if we want to accommodate new extensions with hand differentiation, e.g., switching the primary variables and equation sets, changing the flux approximation scheme, or introducing additional physical models into the system, the process can be quite slow, resulting in extensive code redesign and complex implementations.

An alternative method to generate Jacobian matrices is numerical differentiation [17], which uses truncated Taylor series to approximate the derivatives (e.g., second order central differencing for first derivative: $f'(x) = \frac{f(x+\Delta x)-f(x-\Delta x)}{2\Delta x} + O(\Delta x^2)$). The implementation is usually simpler than hand differentiation because an explicit form of the derivatives is not required, and only the residual equations need to be evaluated at multiple points in the variable space (depending on the number of variables and the scheme of approximation). For instance, if central differencing is used to approximate the partial derivatives of a residual equation f with N variables, $2N$ function evaluations are needed, since for each variable x_i, we need to evaluate $f(x_1, ..., x_{i-1}, x_i + \Delta x_i, x_{i+1}, ..., x_N)$ and $f(x_1, ..., x_{i-1}, x_i - \Delta x_i, x_{i+1}, ..., x_N)$ in order to compute $\frac{\partial f}{\partial x_i}$.

In order to take advantage of the sparsity of the Jacobian matrix and improve efficiency, graph coloring algorithms [18] can be applied to the partition of the target Jacobian matrix [19] so that the number of function evaluations, which are the primary cost in numerical differentiation, can be minimized through optimized partitioning.

However, this method has three primary disadvantages. First, conditional branches

(e.g., upwinding or variable switching) cannot be treated as desired since their derivatives are not continuous and numerical differentiation would give inaccurate approximation (usually smearing) near the discontinuity. Second, it is not always possible to bound the truncation error a-priori for arbitrary functions, i.e., the differentiation interval Δx_i is not easy to determine, since if Δx_i is too large, large truncation errors may result while a Δx_i that is too small would incur considerable round-off error [17]. Moreover, the asymptotic complexity can limit the efficiency since the whole function has to be evaluated at least once for each derivative. For a large number of independent variables, the algorithmic complexity of this approach in terms of the number of function evaluations grows very quickly and can be quite sizable in large simulation problems [2].

In order to overcome these drawbacks, the automatic differentiation (AD) [20] method is used here. By analyzing the expression parse-tree, i.e., decomposing the expression into basic unary (e.g., sin, cos) or binary (e.g., plus, minus) operations, applying basic differentiation rules (e.g., linearity, product and quotient rule) and transcendental elementary function derivatives (e.g., $sin(v) = cos(v) \cdot v'$), as well as performing the chain rule (i.e. $(f \circ g)' = (f' \circ g)g'$) [21], the AD approach offers flexibility, generality and accuracy up to machine precision, since there is no truncation error involved. However, since the algorithmic complexity of AD is at least comparable to that of analytical differentiation, the ability to develop an optimally efficient AD library is problem specific and often requires a large amount of effort [2].

Today, a sizeable research community continues to develop various aspects of AD including improved methods for higher-order derivatives, parallel AD approaches, and sparsity-aware methods. Comprehensive introductions [20, 22, 23], recent research activities [24, 25], literature and software packages are available (see the web site www.autodiff.org). Although AD is already used in numerical simulation and optimization, it is not yet a mainstream approach in industrial-grade, large-scale simulators [26–28].

As analyzed by Younis [2], the reluctance of using AD is not due to computational complexity, which can be comparable to that for hand differentiation in most cases. Instead, the main issue is the lack of implementation techniques that are efficient,

transparent, and notationally clear. That is to say, if we could improve the efficiency and transparency of implementation, and useability of an AD library would be greatly enhanced so that it would become a practical and preferable platform for the development of the next generation General Purpose Research Simulator (GPRS).

4.2 Existing AD Methods

Generally speaking, modern AD methods are implemented in two alternative approaches: 1) Source Transformation, and 2) Operator Overloading.

4.2.1 Source transformation

This approach involves the development of a pre-processing parser that converts the client code, which provides the residual vector, to the target code for the corresponding derivatives. The resultant code can then be compiled and run. Although several highly efficient, optimizing source parsers are available [29–31], the work-flow makes this approach less desirable than the operator overloading approach. When using this method, every time we make a little change and want to see the results, we have to run a parser to analyze the source code, and all the generated code must be recompiled.

4.2.2 Operator overloading

The fundamental components of this approach are customized data-types that are defined to store both a scalar value and its derivatives along with redefined arithmetic and elementary functions that not only compute the function values but also differentiate them automatically. Then by combining the new data-types, functions, and basic differentiation rules including the chain rule, the derivatives of complicated (nonlinear) residuals can be computed systematically (and automatically) step by step. For example, consider the following redefined functions:

$$\underline{a+b} = \{a+b, a'+b'\}, \tag{4.2.1}$$

$$\underline{sin(f)} = \{sin(f), cos(f)\cdot f'\}, \tag{4.2.2}$$

where for any scalar variable v, v' is its gradient containing the derivatives of v with respect to the designated independent variables. The underlined functions (e.g. $\underline{\sin}$) are redefined arithmetic, or elementary, functions that compute both the function value and associated derivatives. The underlined variables (e.g., $\underline{f}$) are of a new data-type in the AD scheme, which is composed of a pair, namely the scalar value and its gradient: $\{value, gradient\}$.

For instance, using the chain rule, the value and derivatives of $sin(a+b)$ can be computed as:

$$\begin{aligned}\underline{sin(a+b)} &= \underline{sin}(\{a+b, a'+b'\}) = \{sin(a+b), cos(a+b)\cdot(a'+b')\} \\ &= \{sin(a+b), cos(a+b)\cdot a' + cos(a+b)\cdot b'\}, \end{aligned} \tag{4.2.3}$$

where the gradient of $sin(a+b)$ is expressed as a linear combination of the gradients of a and b. According to the basic differentiation rules, including linearity ($(cf)' = cf'$, $(f+g)' = f'+g'$), product ($(fg)' = gf' + fg'$), quotient($(f/g)' = (gf' - fg')/g^2 = (1/g)f' - (f/g^2)g'$), and chain rule ($(f \circ g)' = (f' \circ g)g'$) [21], the resultant gradient of one differentiation step is always in the form of cf', or $c_1 f' + c_2 g'$, where c, c_1, c_2 are constants and f', g' are known gradients. Repeating this scheme, we can conclude that no matter what the residual equation, R, looks like, the associated gradient, R', will always be in the form of a linear combination of sparse gradients that are already defined, or computed:

$$R' = c_1 v_1' + c_2 v_2' + ... + c_N v_N'. \tag{4.2.4}$$

4.3 ADETL Framework

4.3.1 Introduction to ADETL

Although there are many well-written publicly available AD libraries, such as ADIC [29], ADIFOR [30] (source transformation packages), FADBAD [32], ADOL-C [33], and ADOL-F [34] (operator overloading packages), they do not fit our needs to build a complex general purpose reservoir simulator. The work flow of source transformation

packages is not desirable, while the efficiency of existing operator overloading packages is not good enough due to the high cost of maintaining maximum flexibility. To adapt to our specific needs, the ADETL framework was developed.

ADETL stands for Automatically Differentiable Expression Templates Library, which was conceived and developed by Younis [2]. It is designed to provide data-structures and algorithms that exploit the power of the operator overloading approach, while overcoming the inefficiency usually associated with it. ADETL is an efficient and extensible framework, which includes the following aspects:

- Large optimized generic library
- Provides core infrastructure for the new GPRS
- Still growing in capability and efficiency

Under the ADETL framework, only residuals need to be coded, and the Jacobian is generated automatically. As a result, the user can save a lot of time in the development and debugging stages of a reservoir simulator by effectively eliminating the effort of deriving the gradients and coding the Jacobian in the traditional paradigm. Moreover, due to the computational efficiency of ADETL, the user is able to get the new extensions and capabilities at a relatively small additional cost in simulation time.

4.3.2 Usage overview of ADETL

The most important customized data-type introduced in ADETL is called the *AD-scalar*, which includes two parts:

- "Value": stores the scalar value of one variable.
- "Gradient": stores the gradient of that variable with respect to the specified independent variables.

The *ADscalar* type can be templatized [2] with its gradient type, such as a point-sparse, or block-sparse, vector, so that it can be extended to support multiple data structures. The procedure of using ADETL are:

1. Declare the independent variables:

   Independent variables contain derivatives with respect to themselves only. They are seeds to generate any other gradients. Generally, they are declared using the "make_independent" member function of *ADscalar* type as follows:

```
// pressure is the 4th variable and temperature is the 6th
pressure.make_independent( 4 );
temperature.make_independent( 6 );
```

2. Write residual code:

   The residual code is written in the same way as for hand differentiation, except that certain double (i.e., double precision numbers) variables, are replaced with *ADscalar* variables so that the gradients can be automatically computed with the help of ADETL. An example that illustrates the idea is:

```
ADscalar compute_density(const ADscalar& _P, const ADscalar& _T){
    const double c1 = 1e-4, c2 = 1.15e-3, c3 = 0.25;
    return c1 * exp( c2 * _P + c3 * _T );
}
```

```
density = compute_density ( pressure, temperature );
```

3. Use automatically generated gradients:

   The third step is to use the gradients, e.g., converting the gradients to the format that the solver could recognize. Then, a linear solution can be obtained to update the independent variables, and therefore one Newton iteration is complete. In the example here, we just use the "spy" function to see the generated gradients:

```
fastl::spy( cout, density.gradient() ); // prints [ . . . X . X ]
```

   The output shows that density has derivatives with respect to the 4th and 6th variables, which are the pressure and temperature as defined in the first step above.

Moreover, one can change the independent variables and repeat the computation as follows:

```
// Now, temperature is no longer a variable
temperature.make_constant( );
density = compute_density ( pressure, temperature );
fastl::spy ( cout, density.gradient() ); // prints [ . . . X ]
```

As indicated in the output, density now only has a derivative with respect to the 4th variable (pressure), because temperature has been set to a constant. This shows that ADETL can work well with different independent-variable sets and produce the correct derivatives.

4.3.3 Working mechanism of ADETL

One fundamental data type to support the ADETL computation is called SPLC (SParse Linear Combinable) argument, which contains the following member variables: (1) coefficient c_i; (2) pointer to sparse gradient v_i; and (3) pointer to next SPLC argument.

As explained in Section 4.2.2, regardless of the form of the residual equation, its associated gradient will always be in the form of a linear combination of sparse gradients that are already defined, or computed. So each expression object is actually composed of a scalar value and a linked list of SPLC arguments with c_i as combination weights and v_i as the sparse gradients to be combined.

When an expression is written, three primary stages take place under-the-hood:

1. Building the expression object: in the first stage, the expression object is recursively constructed through overloaded operators as:

$$E = ads\_xpr\{value, [c_1v_1, c_2v_2, ..., c_Nv_N]\}. \tag{4.3.1}$$

   The form of the *ads_xpr* object implies that the "value" part also gets computed in this stage. This is because it is more efficient to compute the scalar value together with the computation of the coefficients c_i than to compute it alone in another stage.

2. Evaluating the expression object: in the second stage, the "gradient" part is evaluated by combining the SPLC arguments. Through overloading of the constructor and the assign ("=") operator, the *ADscalar* can get the value directly from the expression object and the gradient through the evaluation algorithm performed on these SPLC arguments, as shown in the following:

$$\begin{aligned}\{value, gradient\} &= Eval(E) \\ &= \{E.value, SPLC\_eval([c_1v_1, c_2v_2, ..., c_Nv_N])\} \\ &= \{E.value, c_1v_1 + c_2v_2 + ... + c_Nv_N\} \end{aligned} \tag{4.3.2}$$

3. Tearing down the expression object: the third stage involves destruction of the expression object. Each SPLC argument in E will be deallocated so that the memory pool will be ready for the next expression.

With these steps, not only the value but also the gradient of an *ADscalar* variable can be computed according to the given residual expressions.

As a concrete instance of using ADETL, a typical compositional reservoir simulation is primarily composed of Successive Substitution Iteration (SSI) for EOS-based stability and flash calculations, Newton's method for stability and flash, property calculation, accumulation-term calculation, and the stenciling operations involved in flux computations. Usually the calling frequency and run-time of each part is quite different, e.g., for a simulation using a millions cells, SSI would be called $O(10^9)$ times and occupy 10-15% of the total time, while stenciling operations would be called $O(10^6)$ times only and use 5-15% of the total time.

It is important to recognize that the gradient patterns for different kernels are quite different. As shown in Figure 4.1, no derivatives are required for the SSI process, in which the computation depends only on the variables (phase compositions) and residual equations (fugacity criteria) and no derivatives are needed. For property calculation, Newton flash, or computation of the accumulation term, there are either point-sparse or block-sparse gradients (usually diagonal), since these processes primarily involve computations in a local block. As for stenciling operations, there

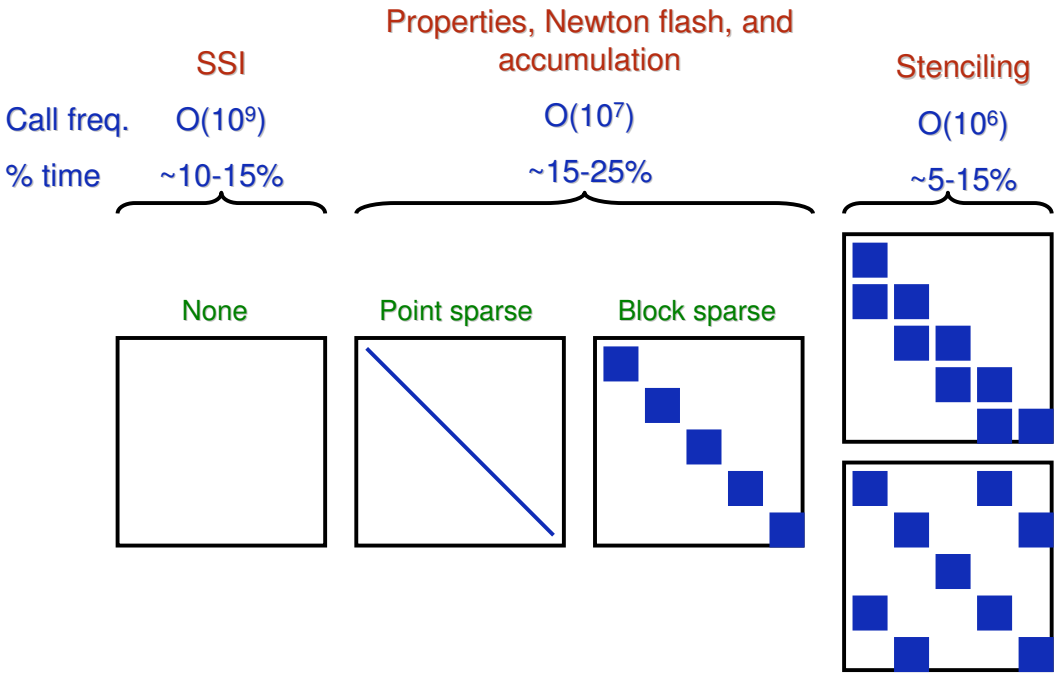


Figure 4.1: Jacobian patterns for a typical compositional simulation [2]

are different patterns (depending on the connection information) of block-sparse gradients, since in this process, the computations are usually between two connected blocks and the coupling is reflected in the Jacobian matrix. These characteristics imply that over the life of a single residual call, the sparsity profiles associated with the Jacobian matrix vary widely. In order for the above three-step procedure to work efficiently for various scenarios employed by different kernels, multiple data structures are supported by ADETL through the class template approach. These templatized classes share common interfaces so that one can change among different allocators, containers, manipulators, and so on, by only changing the template parameter, which is quite flexible and highly efficient.

4.3.4 ADETL roadmap

In this section, the fundamental framework, original components, and recent extensions of ADETL will be briefly introduced.

As shown in Figure 4.2, for the data structure to store the gradients, we could have

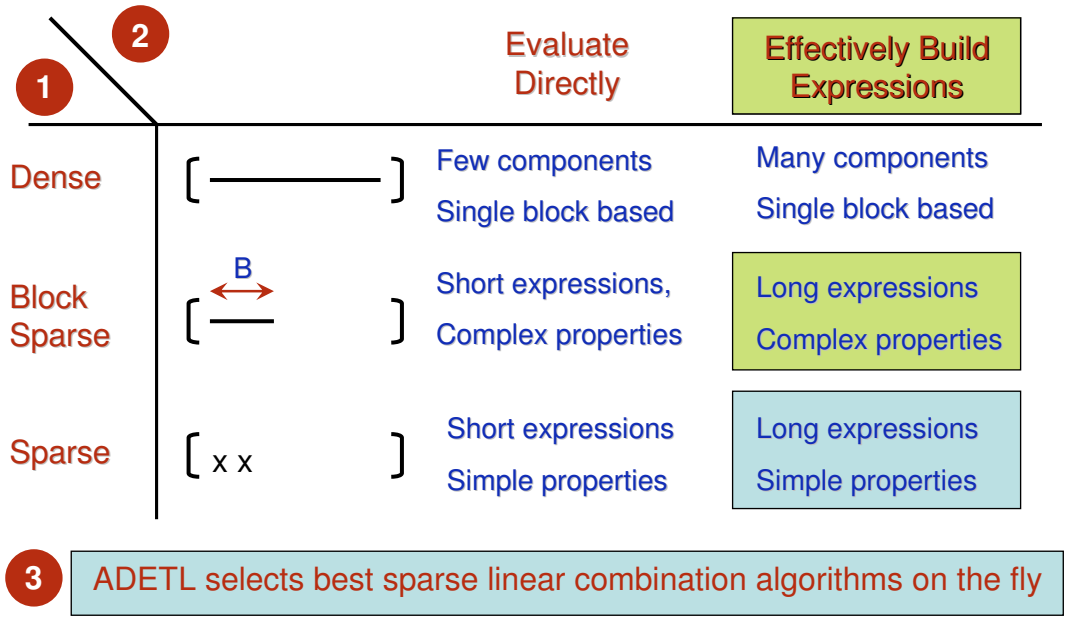


Figure 4.2: ADETL component roadmap [2]

dense (for single block-based computation), block-sparse (for complex properties), or point-sparse (for simple properties) options. As for the evaluation mechanism of expressions, we could have either direct evaluation (for short expression) or evaluation after building the expression objects (for long expressions). Also, ADETL will automatically select the best linear combination algorithm for evaluation at run-time, such as Sparse $Y = aX + Y$ (SAXPY, for local combination) or Implicit Binary Tree (IBTree, for stenciling combination) evaluation algorithms.

In the previous version of the library, the point-sparse option, evaluation-after-building option, as well as automatic selection of evaluation algorithms (SAXPY or IBTree) have already been implemented. As motivated in the previous sections, extension to effective expression construction through customized allocators (discussed in Section 5.1) and block-sparse option (discussed in Section 5.2) are investigated in this report. Additional improvements, such as in the evaluation algorithms and supporting containers, have also been implemented and integrated into ADETL.

Chapter 5

Recent Extensions to ADETL

5.1 Building Stage — Customized Allocators

In this section, the motivation for the improvement over the previous allocation method in expression building and the memory usage patterns related to the design of memory allocators are discussed. Then, four allocator options, including one previous common allocator and three new customized allocators, are investigated in detail:

- Common free list allocator
- Specialized free list allocator
- Non-recyclable, coherent allocator
- Recyclable, coherent allocator

5.1.1 Motivation and memory usage patterns

As is well documented [35], frequent memory allocation and deallocation can play a significant role in degrading application performance. According to the mechanism of expression building, there are a lot of allocation and deallocation operations for SPLC arguments involved in this stage. Thus, the performance of the allocator determines, to a large extent, the efficiency of this stage.

Allocators are the basic level objects for memory management. They provide operations for run-time memory allocation and deallocation. For example, the simplest wrapper allocator directly uses the built-in memory allocation / deallocation function "malloc", "free", "new", and "delete" [36] when corresponding operations are requested. These standard allocators provide a common interface for higher-level usage. On the other hand, a pool allocator utilizes a certain type of preallocated memory pool, and would give out, or take back, pieces of memory from / to the pool when requested. As a result, system memory management function calls can be avoided, which can result in large performance gains in most cases. Although the internal mechanisms of the various allocators can be quite different, they generally share the same interfaces for the four primary parts:

- Constructor, for initializing the allocator and creating the buffer;
- Destructor, for deallocating all the memory dynamically reserved by the allocator;
- Allocation function, for "giving out" a memory chunk from the buffer, which returns a pointer of that chunk;
- Deallocation function, for "putting back" a memory chunk into the buffer.

Basically, the allocators are classified according to the following characteristics:

- The size of each memory piece, which could be:
  - Fixed-size (the piece allocated, or deallocated, each time must be the same size)
  - Variable-size (each allocated memory piece can have a different size)
- Recycling of the memory, which could be:
  - Non-recyclable (deallocation function will not put back the memory piece, which cannot be given out again until the pool is manually emptied, or destroyed. This is best for schemes requiring extreme efficiency and relatively small memory)

  - Recyclable (memory pieces will be put back during deallocation, and they can be given out by the allocator again. This is best for schemes that require large memory and moderate efficiency)

- Coherence of the allocated memory, which could be:

  - Non-coherent (the layout of allocated pieces is random and no effort is invested to make them contiguous. This is best for schemes that do not depend on cache utilization)
  - Coherent (the layout of allocated memory will be as contiguous in raw memory as possible. This is best for schemes with frequent visiting of the memory and heavy dependence on cache)

Our targeting object is the SPLC argument, which is fixed in size. So only fixed-size allocators are considered. Also, for non-recyclable allocators, no extra effort needs to be taken to ensure coherence so that non-recyclable, non-coherent allocators are not considered here. The remaining three types of allocators are discussed in this work, namely, non-recyclable, coherent; recyclable, non-coherent (free list); and recyclable, coherent.

SPLC arguments that compose AD expression objects have very particular memory usage patterns, which we can take advantage of to design customized allocators that are very efficient. These patterns include:

- Serialization: In one thread, only one primary expression is built at a time. Sub-expressions are built, evaluated, and freed before the building of a primary expression can be resumed. Note: Sub-expressions are those expressions in intermediate ADscalar functions inside one primary expression, such as the expression of W in intermediate function "func" in the following example:

```
ADscalar func(const ADscalar& P) {
ADscalar W = 3. * P * P + 1.;
return W;
}
```

```
ADscalar X = T * func(P); // func is an intermediate function
```

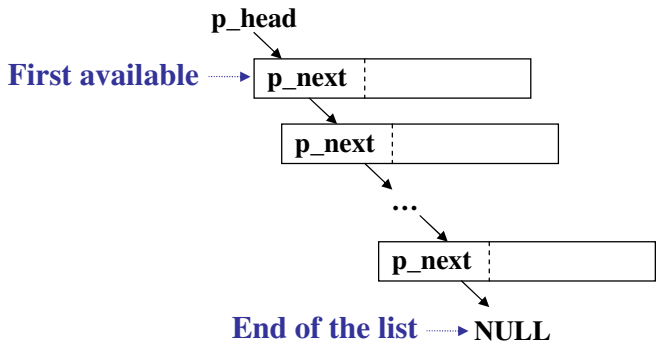


Figure 5.1: Working mechanism of a free-list allocator

- Quick cycle: SPLC arguments are frequently deallocated when evaluation of an expression is done. Therefore, they have a very short life cycle, and we do not allow storage of expression objects.
- Sequential allocation: For coherent allocators, deallocation occurs in a predictable sequence, which means that when an SPLC argument is deallocated, everything after it should also get freed from memory at the same time.

These memory usage patterns will be utilized in the design of new customized allocators.

5.1.2 Previous allocator: common freelist

Previously, a common freelist allocator (recyclable, non-coherent) was used exclusively in the ADETL library for building expressions. It is the most flexible non-coherent allocator for fixed-size objects, and its working mechanism is illustrated in Figure 5.1.

The primary parts of this allocator are implemented as:

- Constructor: Certain number of memory chunks are allocated and connected by their "p_next" pointer, and "p_head" is set to the first chunk.
- Destructor: Iterate from "p_head" to the last chunk: deallocate each chunk in the list.

- Allocation function: Check the availability of a memory chunk. If there is no more chunks available, expand the pool by allocating a certain number of memory chunks. Then, give the first chunk out, and move "p_head" to the next chunk.
- Deallocation function: Put the returning memory chunk ahead of the current list, and set "p_head" to it.

This implementation provide wide flexibility. However, since it does not take advantage of the memory usage patterns listed earlier, the computational efficiency is far from optimal. This leads to the motivation of designing new allocators.

5.1.3 New allocator 1: specialized free list

The first new option is a "specialized free list" allocator, which is built based on the previous allocator option, the "common free list". Its working mechanism is also illustrated in Figure 5.1.

In order to improve the efficiency, we can utilize the "Quick Cycle" pattern, which assumes that the allocated memory chunks storing the SPLC arguments are frequently returned back to the free list, when the evaluation of the expression is done. As long as the initial buffer is sufficiently large, the pool will never be used up. Consequently, the approach includes:

- Set a large enough initial buffer that can hold the arguments of any single expression.
- Never check the availability of memory chunks during allocation; always give out the first chunk.

Moreover, in order to provide better "coherence", small allocations of memory chunks during initialization, or expansion, of the pool, as implemented in the common free list allocator, should be avoided. Instead, the entire buffer should be allocated once at initialization. As a result, the pool is initially coherent but cannot maintain coherence after deallocation operations take place. Although not optimal, this is better than the common free list, which does not guarantee any coherence.

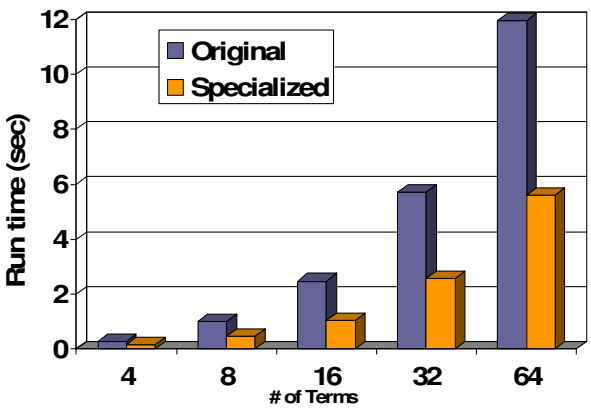


Figure 5.2: Benchmark of specialized and common free list

The primary parts of this allocator are implemented as:

- Constructor: Allocate a large enough initial buffer, divide it into a lot of contiguous memory chunks connected by their "p_next' pointer, and set "p_head" to the first chunk
- Destructor: Deallocate the entire buffer
- Allocation function: Give the first chunk out, and move "p_head" to the next chunk
- Deallocation function: Put the returned memory chunk ahead of the current list and set "p_head" to it

This approach is expected to improve efficiency as demonstrated in the benchmark of building an expression with X terms and four derivatives (Figure 5.2). Compared with the common free list, the time spent by the specialized freelist is less than half of that spent by the common freelist.

However, this strategy involves the limitation that the size of the buffer has to be set in advance, and it must be large enough for all the cases. Otherwise, when the pool is used up, the library would try to access a NULL address, resulting in a "segmentation fault" and termination of the program execution.

5.1.4 New allocator 2: non-recyclable, coherent

For non-recyclable allocators, the priority is efficiency, although more memory could be consumed, since literally no recycling operations are performed.

However, the memory pool is usually not large enough for the whole simulation especially for large-scale problems. For practical simulation, the following approaches are considered:

- Apply the non-recyclable pool only to some "hot spot" expressions, and destroy the pool afterwards.
- Empty the pool from time to time.

The first approach can be effective, if we can analyze the "hot spot" with the help of code analysis, or a profiler. However, since we cannot conveniently templatize expression objects with particular allocator types (otherwise, users have to take care of ADscalars with different allocator arguments, which cannot be combined in the same expression), the only way to realize this approach is through run-time allocator selection, i.e., we explicitly set the allocator for all expressions to the non-recyclable one before hot-spot expressions, and then set it back to a certain recyclable allocator after these expressions. Without changing the form of the expressions to build, virtual allocation / deallocation functions must be involved in the allocators to maintain a common interface. According to preliminary test results, virtual function calls deteriorate the performance because the inlining of the allocation function is prohibited. Hence, this approach is not desired.

The key point of the second approach is the proper time to empty the pool. This can be determined using the "Serialization" and "Quick Cycle" patterns, by which we know that the memory chunks should be deallocated immediately after the evaluation stage, and all of them are expected to be returned to the pool after a primary expression gets evaluated, or equivalently, the entire buffer should be available again at the end of the tearing down stage of a primary expression. Thus, the strategy can be implemented as follows:

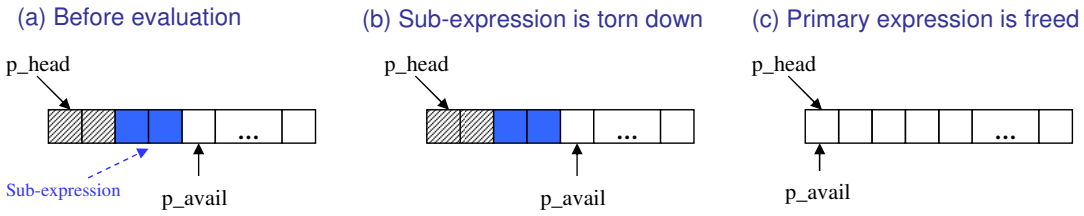


Figure 5.3: Working mechanism of non-recyclable, coherent allocator

- Do nothing when a sub-expression is torn down, i.e., keep the current position of the "p_avail" pointer, as shown in Figure 5.3(b).
- Empty the entire pool when a primary expression is freed, as shown in Figure 5.3(c).

Using the above approach and setting the initial buffer large enough to contain the SPLC arguments of the longest expression (including sub-expression arguments), we can expect the pool to never be exhausted. The primary parts of this allocator are implemented as:

- Constructor: Allocate a large enough initial buffer, and set "p_avail" to the beginning.
- Destructor: Deallocate the entire buffer.
- Allocation function: Give out the chunk pointed by "p_avail", and increase "p_avail".
- Deallocation function: Do nothing.
- Special emptying function: Gets called only when a primary expression gets freed, then set p_avail to the beginning of the buffer.

The advantage of this approach is its high efficiency, which is demonstrated in Figure 5.4. Even compared with the specialized free list, which is proved to be efficient, this non-recyclable, coherent allocator has better performance.

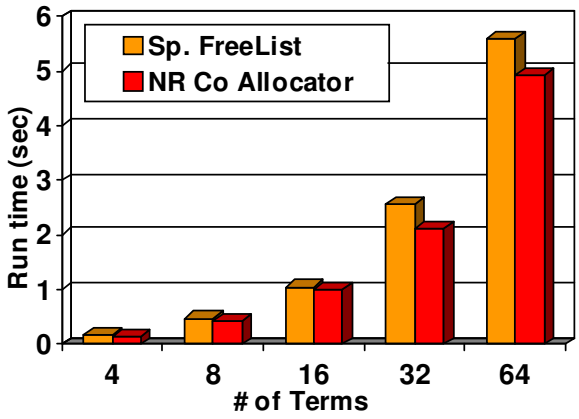


Figure 5.4: Benchmark of specialized free list and non-recyclable, coherent allocator

However, this approach also involves an additional limitation to the requirement of setting the size of the buffer in advance. Specifically, all sub-expressions must be evaluated by the constructor of ADscalar, i.e., we cannot evaluate a sub expression repeatedly in an intermediate function. Otherwise, the SPLC arguments of the primary expression will be lost, resulting in unpredictable results.

5.1.5 New allocator 3: recyclable, coherent

This is the last new allocator, which attempts to achieve recycling and coherence simultaneously. To guarantee coherence, a similar pool structure to that used for the non-recyclable, coherent allocator is adopted (Figure 5.5(a)).

If we try to manage the pool with a linked list of available blocks (a block contains several contiguous memory chunks), the allocation function can be almost as easy as that in the non-recyclable, coherent allocator. During allocation, the allocator simply gives out the chunk at the beginning of first available block and adjusts the beginning address of that block accordingly.

However, the deallocation function can get very complicated, since we need to figure out the exact place to put the returned chunk, which can be immediately before or after one block, or forming a separate block. Even with a heuristic search strategy (i.e., always start searching from the last block found), the complexity is too

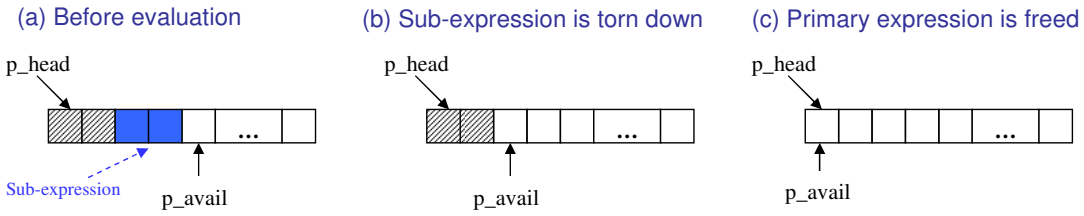


Figure 5.5: Working mechanism of recyclable, coherent allocator

high, and the performance is not acceptable.

Therefore, a second approach, which utilizes "Serialization" and "Sequential Allocation" patterns, is proposed. Based on these two patterns, we can expect the following behaviors:

- Free chunks are always contiguous from one address to the end of the memory pool.
- Once a chunk gets freed, everything after it should get freed as well.

Then the corresponding strategy can be designed and applied to the allocator:

- Only a "p_avail" pointer is used to manage the available block (from "p_avail" to the end of the pool), and no linked list is needed.
- Deallocation operation always sets the "p_avail" pointer to the beginning chunk of an expression, regardless of whether it is a primary- or sub- expression, as demonstrated in Figure 5.5(b)(c)

The primary parts of this allocator are implemented as:

- Constructor: Allocate a large enough initial buffer, and set "p_avail" to the first location.
- Destructor: Deallocate the entire buffer.
- Allocation function: Give out the chunk pointed by "p_avail", and increase "p_avail".

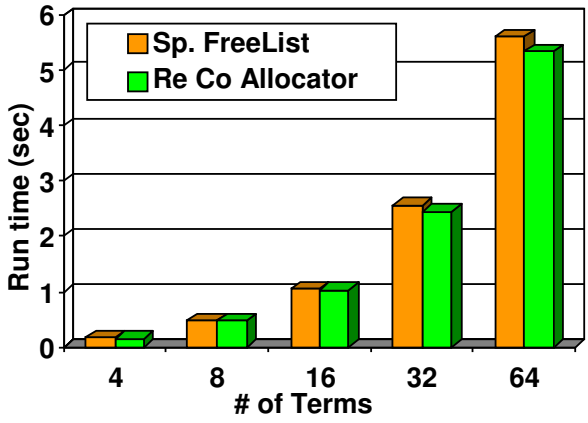


Figure 5.6: Benchmark of specialized free list and recyclable, coherent allocator

- Deallocation function: If the address of the returned memory chunk is before "p_avail", set p_avail to this address. Consequently, after all SPLC arguments are deallocated, "p_avail" will be set to the chunk with the smallest address, which is the beginning of the primary-, or sub-, expression.

The advantage of this approach (see Figure 5.6) is that it is more efficient than the specialized free list, but a little slower than the non-recyclable, coherent allocator. Moreover, there is no restriction on sub-expression creation, which provides flexibility in treating intermediate functions. Of course, setting a large enough initial buffer beforehand is also required for this approach, which is the only limitation.

5.1.6 Concluding remarks

In this section, the most important improvements in the expression building stage of ADETL are discussed. The previous allocator and three new ones are investigated including their working mechanism, detailed implementation, benchmark results, advantages and limitations. Based on our findings, the recommended allocators for further usage are:

- Recyclable, coherent allocator (2nd highest efficiency with flexible sub-expression creation)

- Non-recyclable, coherent allocator (highest efficiency with restriction on sub-expression creation)

The restriction on sub-expression creation is actually not very limiting. For small intermediate functions, a sub-expression can usually be evaluated with only the constructor of ADscalar. Even if we have to use the "=" operator of ADscalar repeatedly, we can use a temporary variable specifically for that intermediate function, and then put it into the primary expression. Of course, for maximum flexibility, the recyclable, coherent allocator is the best choice, although it is a little slower than the non-recyclable allocator.

Regarding extension to multi-threaded environments, since "Serialization" and "Sequential Allocation" patterns are utilized in the last two options, thread locking will affect efficiency considerably. The proposed solution is to use one memory pool (allocator) for each thread. If we always handle one expression by one thread (multiple expressions can then be handled by multiple threads), thread safety can be guaranteed, which is a necessary requirement for using ADETL in a multi-threaded computational environment.

5.2 Evaluation Stage — Block-sparse Data Structure

In this section, the proposed scheme and target evaluating algorithms are first discussed. Then two approaches to implement the block-sparse data structure are investigated in detail:

- Case-dependent block size
- Case-independent block size

Then, some analysis on the usage of the block-sparse option will be given based on the results of practical simulation test cases and systematic expression evaluation benchmarks.

5.2.1 Proposed scheme and target algorithms

The block-sparse data structure usually provides efficiency improvements when used with block-based sparse computational kernels [1,37]. Since we already have a point-sparse option in ADETL, a convenient scheme for the user to utilize both the new block-sparse and the original point-sparse options is necessary. The scheme proposed is as follows:

- The user can freely choose between ADscalar <sparse_vector> and ADscalar <block_sparse_vector>.
- Two data types should share a common interface, so that no modification is needed in the client code.

We expect the new block-sparse data structure to have the following properties:

- Well-suited for block-based computations (e.g., accumulation).
- Provide better data locality.
- Compatible with existing ADETL evaluating algorithms:
  - SAXPY (Sparse $Y = aX + Y$) evaluator (primarily for local combination, see Figure 5.7)
    1. Prepare the dense buffer
    2. (+) Prolong product of combining weight c_i and sparse gradient v_i into the buffer
    3. (+) Restrict the buffer to the resulting vector
  - IBTree (Implicit Binary Tree) evaluator (primarily for combination of sparse gradients in different grid blocks, see Figure 5.8)
    1. Bind all arguments to the IBtree
    2. (+) Advance the smallest node into the cache
    3. (+) Assign the cache to the resulting vector

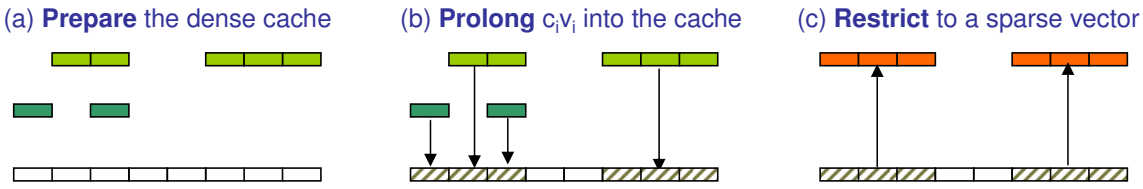


Figure 5.7: Working mechanism of SAXPY evaluator

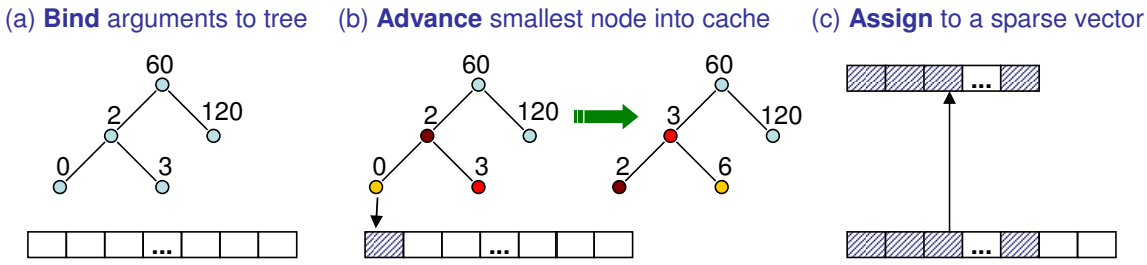


Figure 5.8: Working mechanism of IBTree evaluator

As shown in the list above, the "Prolong" and "Restrict" steps of the SAXPY evaluator, and the "Advance" and "Assign" steps of the IBtree evaluator have potential efficiency benefits from the block sprase data structure.

5.2.2 Case-dependent block size

One important fact about the design of the block-sparse data structure is that it is impossible to select the optimal case-dependent size of the block (usually, $NC + 1$, where NC is the number of components) at compile time because the input file is not read until run time, and NC varies from case to case.

Figure 5.9: Block nonzero entry of two approaches

So the first approach, which uses a case-dependent block size, selects the block size at run time as follows:

- Set block size (static number) to $NC + 1$ after simulation begins.
- Each nonzero entry contains a column and a pointer to the data.
- The data storage is dynamically allocated.
- Block operations are translated into loops from 0 to NC.

The block nonzero entry used in this approach is demonstrated in Figure 5.9(a). This approach seems natural at first sight. However, it implies the following operations:

- Pointer dereferencing (every time the value part of a nonzero entry is visited, the "val" pointer needs to be dereferenced, which is slower than direct access to an element in a built-in double array).
- Dynamic allocation (the value part of each nonzero entry must be dynamically allocated in the construction, or capacity adjustment, of the gradient vector, after setting the block size. This is slower than static allocation at the beginning of the simulation).
- Frequent looping (since the block size is not known at compile time, each block operation needs to be translated into a loop inside a block, which is slower than an unrolled loop, i.e., directly repeating simple instructions instead of using a "for" loop).

These operations result in poor efficiency and considerable overhead compared with the point-sparse data structure. Even though this approach has advantages (e.g. provide better data locality), the overhead will lead to deterioration in the overall performance, causing this approach to spend twice as much time as the point-sparse option on the same expression evaluation according to the preliminary test results.

5.2.3 Case-independent block size

As discussed before, since an optimal block size for each case cannot be selected at compile time, a second approach that uses a case-independent block size is proposed. Its corresponding primary procedures are described as follows:

- Decide on a case-independent block size (e.g., 4) before compile time.
- Each nonzero entry contains a column and a built-in double array.
- The data storage is allocated on the stack, often statically at the beginning.
- Block operations are performed with automatic loop unrolling.

The block nonzero entry used in the second approach is shown in Figure 5.9(b). Its value part is a double array of size four, which is fixed in its size at compile time. As a result, static allocation can be used for the value part of each nonzero entry in the construction, or capacity adjustment, of the gradient vector, which saves a lot of time compared to dynamic allocation. Moreover, in this structure of nonzero entries, the data (four double precision numbers) of the value part is guaranteed to be stored after the column part, which provides even better data locality than the first approach.

As introduced above, the pointer dereferencing, dynamic allocation and frequent loop operations are avoided in this approach. Instead, with a fixed-size built-in data type, static allocation and unrolled loops result in much higher efficiency compared with the case-dependent block-size approach.

Since the case-dependent block size ($NC + 1$) is not used, when $NC + 1$ is not a integral multiple of the fixed block size (e.g., 4) we decided on, the nonzero block entries would not match the positions of actual nonzero elements perfectly so that we have to store extra zero elements and perform more double arithmetic operations than the case-dependent block-size approach. However, the performance benefit of using a fixed case-independent block size (direct access to data instead of pointer dereferencing, static allocation, and unrolled loop) can very well overcome them.

To achieve better efficiency, several linear algebraic functions are designed to replace the original mathematical operations in the evaluating algorithms. For example,

```
s_dense_cache[ col ] += coef * itrV->val();
```

is replaced with:

```
__C::axpy(s_dense_cache[ col ], coef, itrV->val());
```

where "__C" is the templatized gradient data type, which can be either the sparse_vector (point-sparse data type), or the block_sparse_vector (block-sparse data type), and "axpy" represents the algebraic operation $y = a \cdot x + y$, which is implemented for both choices of "__C". For point-sparse option, "axpy" is simply implemented as one arithmetic operation $y = a \cdot x + y$, where x and y are both scalars. As for block-sparse option, "axpy" is implemented as four arithmetic operations: $y_1 = a \cdot x_1 + y_1$, ..., $y_4 = a \cdot x_4 + y_4$, where x and y are both blocks of size four. These two implementations share the same interface, so that they can be called in exactly the same way in the evaluating algorithm.

On the other hand, if we only overload the "+=" and "*" operators for the example above, temporary storage has to be provided for the product of a scalar and a block. In addition, for each block in the source vector, we have to assign the intermediate result (coef * itrV->val()) to the temporary storage first, and then add them to the target vector. Using the new templatized-algebraic-operation approach, we can minimize the extra storage and assignment required by the block arithmetic operations and preserve the performance of point-sparse operations. Further extendability to other data structures is also guaranteed, i.e., we can simply implement the algebraic operation for the new data structure without changing the interface, or the evaluating algorithm.

5.2.4 Block-sparse usage analysis

When the block-sprase and point-sparse options are both available for use as the gradient data type, we might ask ourselves: can the block-sparse option replace the point-sparse option completely? From the results of practical simulation test cases shown in Figure 5.10, we can see that by only using the point- or block- sparse option,

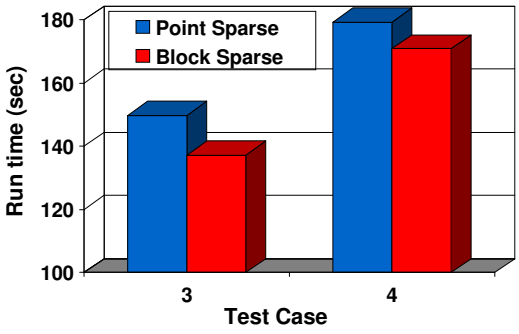


Figure 5.10: Practical simulation cases using only point- or block- sparse option

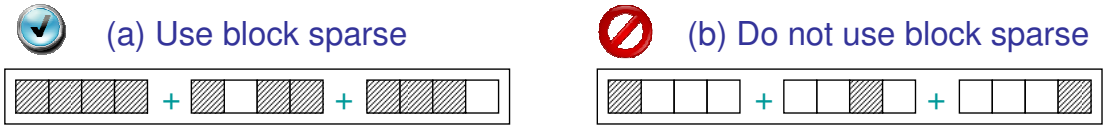


Figure 5.11: Block pattern of different computation schemes

the block-sparse option can outperform the point-sparse option by 5% to 10%, which is an improvement, but far from being very significant.

So, when should we use the block-sparse data structure? Intuitively, if we have some computational scheme as shown in Figure 5.11(a), i.e., the blocks are almost filled with derivatives, we should definitely use the block-sparse option; if we have a scheme like Figure 5.11(b), namely each block contains only one derivative, we should not use the block-sparse option.

We can introduce the apparent usage ratio Ra to describe the sparsity of block nonzero entries, which is defined as:

$$Ra = \frac{nnz}{4 \cdot nnz_b} \times 100\%, \tag{5.2.1}$$

where nnz is the total number of nonzero derivatives, and nnz_b is the number of block nonzero entries (a block entry is nonzero if there is at least one nonzero derivative in it). By definition, we have $25\% \leq Ra \leq 100\%$, and the block-sparse data structure should perform better with larger Ra, which is demonstrated in the benchmarks below.

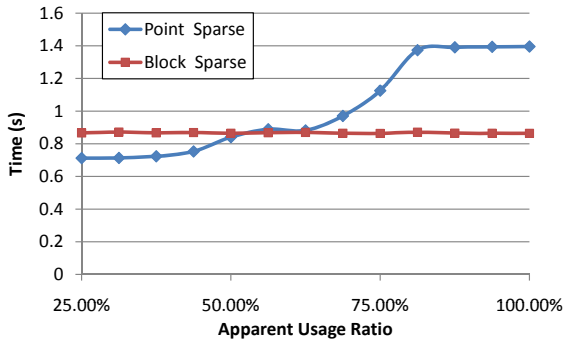


Figure 5.12: Benchmark of SAXPY evaluator with 4(6) arguments

Since the block-sparse option performs differently for different algorithms, and IBtree evaluator uses a special algorithm for expressions with exactly two arguments, the expression-evaluation benchmark is designed for the following three algorithms:

- SAXPY evaluator with 4(6) arguments.
- IBTree evaluator with 2 arguments.
- IBTree evaluator with 4 arguments.

SAXPY evaluator with 4(6) arguments

The following expression is evaluated to examine the performance impact of the block-sparse and point-sparse options for the SAXPY evaluator:

$$W = E_1 + E_2 + E_3 + E_4 + V + V, \tag{5.2.2}$$

where $E_i \quad (i = 1, 2, 3, 4)$ are varying gradient vectors, each of which can have one to four derivatives in one block. By changing the number of derivatives in the E_i parameters, we can have scenarios of different Ra values from 25% to 100%. V is a constant gradient vector, which is added solely for minimizing the optimization bias in certain circumstances in order to achieve a fair comparison. Without V, the monotonicity of the cost with increasing Ra is not satisfied for the point-sparse option, which is counter-intuitive.

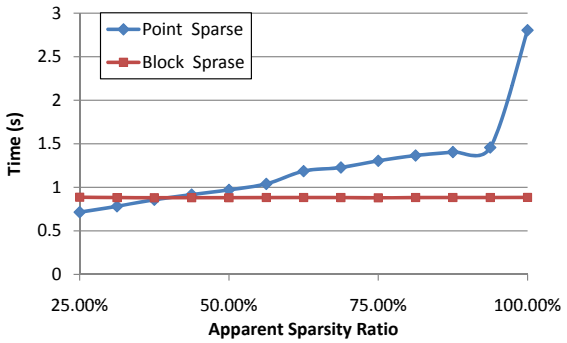


Figure 5.13: Benchmark of IBTree evaluator with 2 arguments

The cost versus apparent usage ratio, Ra, is shown in Figure 5.12, from which we can see that the cost of the point-sparse option increases monotonically with increasing Ra, because there are more entries to be prolonged into the dense buffer. On the other hand, the time of the block-sparse option remains almost the same, since as long as there is one block entry, no matter how many nonzero derivatives are in it, there should be no difference in the cost. These facts indicate that the block-sparse option is slower than the point-sparse option when $Ra \leq 50\%$, and that it outperforms the point-sparse option when $Ra > 50\%$. In the best case with $Ra = 100\%$, the speedup is 1.62 times, while in the worst case with $Ra = 25\%$, the extra cost is 21.7%.

IBTree evaluator with 2 arguments

The following expression is evaluated for the IBTree evaluator with 2 arguments:

$$W = E_1 + E_2, \tag{5.2.3}$$

where $E_1 and E_2$ are varying parameters, each of which can have two to eight derivatives in two blocks.

The comparison results are demonstrated in Figure 5.13. It is shown that the block-sparse option is slower than the point-sparse option when $Ra \leq 31.25\%$ and has better performance for $Ra > 31.25\%$, which is quite a wide range. In the best case

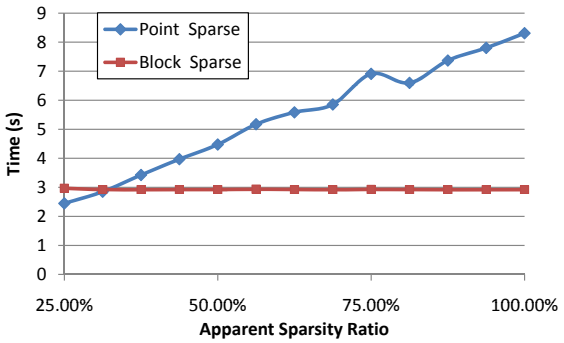


Figure 5.14: Benchmark of IBTree evaluator with 4 arguments

with $Ra = 100\%$, the speedup is 3.17 times, while in the worst case with $Ra = 25\%$, the extra cost is 24.0%.

IBTree evaluator with 4 arguments

The following expression is evaluated for IBTree evaluator with 4 arguments:

$$W = E_1 + E_2 + E_3 + E_4, \tag{5.2.4}$$

where $E_i \quad (i = 1, 2, 3, 4)$ are varying parameters, each of which can have one to four derivatives in one block. The results are shown in Figure 5.14. Similar conclusions as for the IBTree with 2 arguments can be drawn. Specifically, the block-sparse option is slower than the point-sparse option when $Ra \leq 31.25\%$ and has better performance for $Ra > 31.25\%$. In the best case with $Ra = 100\%$, the speedup is 2.85 times, while in the worst case with $Ra = 25\%$, the extra cost is 21.3%.

These test results provide guidance in determining which data structure to use for different scenarios. Generally speaking, all independent variables and simple property computations should use the point-sparse option, while complex property calculations should use the block-sparse option. Ideally, with a flexible and efficient combination scheme (the capability of using both options in a single expression and evaluating from one type of expression to another type of gradient, which can be realized in the future work), the best efficiency can be achieved by combining both point-sparse and

block-sparse options as appropriate.

5.2.5 Concluding remarks

With the motivation to be compatible with block-based computation and provide better data locality, two block-sparse approaches, with case-dependent and case-independent block sizes, are investigated in detail. The selection of the block size, the structure of nonzero entries, the allocation of data storage, the realization of block operations and their impact on the performance are discussed and the two approaches are compared. Based on our findings, the recommended approach is the case-independent scheme with compile-time block size determination.

Then, the results of practical simulation test cases and systematic expression evaluation benchmarks using the point-sparse option and block-sparse option respectively are analyzed for the block-sparse usage. The apparent usage ratio, Ra, is introduced. According to the test results, the block-sparse option should be used with high Ra ($>$50% for SAXPY evaluator and $>$31.25% for IBTree evaluator) and the point-sparse option should be used when Ra is low. Moreover, the optimum solution is to wisely combine point-sparse and block-sparse options, so that both simple and complex properties can be handled in an efficient manner.

Chapter 6

Numerical Examples

Improvements in both flexibility and computational efficiency for practical reservoir simulation are the ultimate goals of this research work. Thus, all the numerical examples in this chapter are compositional simulation cases. The comparison is made among the following simulators:

- Latest **GPRS**: without AD support, version 2.2.3, built Jan. 2009.
- **ADETL[p]**: the prototype ADETL-based simulator [38] with the original common free list allocator and point sparse option.
- **ADETL[b]**: the prototype ADETL-based simulator with the new non-recyclable, coherent allocator, the block sparse option, and other improvements.

We investigated five test cases, including different model sizes, dimensionality, composition, and flow scenario, as listed below:

- Immiscible gas injection (1D, 4 comp.)
- Miscible gas injection (1D, 4 comp.)
- Two wells with a high permeability channel (2D, 9 comp.)
- SPE10 top layer, five-spot pattern (one injector + four producers, 2D, 4 comp.)
- Four wells in the presence of high permeability channels (3D, 4 comp.)

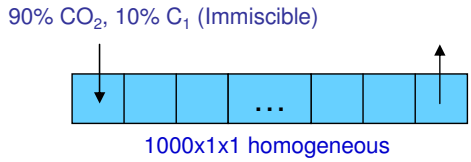


Figure 6.1: Schematic diagram of part II, case 1

6.1 Immsicible Gas Injection

Case Description: (see Figure 6.1)

- 1000x1x1 homogeneous, 1% CO_2, 20% C_1, 29% C_4, 50% C_{10} at 76 bar, 372 K
- One injector at (1,1,1): 90% CO_2, 10% C_1, BHP control at 126 bar
- One producer at (1000,1,1): BHP control at 50 bar
- Run for 400 days with a maximum time step of 5 days

Figure 6.2 shows the results, which include composition, pressure, and saturation profiles at the end of simulation using **ADETL[b]**. The results are consistent with those of **GPRS**. Correctness of the ADETL-based simulator is thus validated. This applies to the four cases discussed next, so that only the solutions using **ADETL[b]** will be shown.

The efficiency comparison among the three simulators is given in Table 6.1. Since the ADETL simulators use slightly different variable sets and formulation from GPRS, the nonlinear behavior is a little different resulting in somewhat different time step numbers, Newton, and linear-solver iterations.

From the fifth column in the table, we can see that the largest improvement (3.54 times speedup) is in the stability and property calculation parts, which include the most local computations. Hence large numbers of expression building, evaluation, and destruction operations are involved, which are exactly the target of our extensions. The discretization part is also improved significantly (1.91 times speedup), but not as much as the stability and property calculation parts, which is due to the fact that the primary operations in discretization are stenciling, which benefit less from the

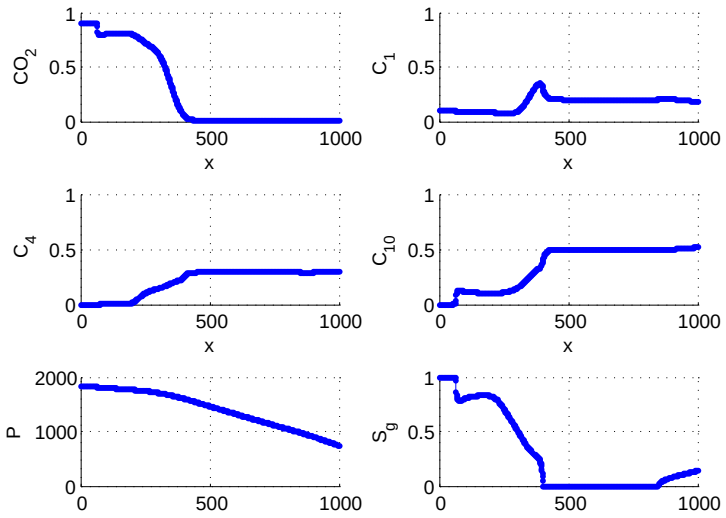


Figure 6.2: ADETL[b] solution of part II, case 1

implemented improvements compared to local computations. And we see no difference in the linear-solver time, because this kernel was not modified in any way. For the total simulation time, compared with **GPRS**, a 43.9% extra cost is consumed by **ADETL[b]**, which is within the reasonable range for a simulation research platform.

6.2 Miscible Gas Injection

Case Description: (see Figure 6.3)

- 1000x1x1 homogeneous, 1% CO_2, 20% C_1, 29% C_4, 50% C_{10} at 76 bar, 372 K
- One injector at (1,1,1): 90% CO_2, 10% C_4, BHP control at 126 bar
- One producer at (1000,1,1): BHP control at 50 bar
- Run for 400 days with maximum time step of 3 days

Figure 6.4 shows the composition, pressure, and saturation profiles at the end of the simulation obtained by **ADETL[b]**. The efficiency comparison is shown in Table 6.2. Again, we can see that the largest improvement (3.64 times) is in the

Table 6.1: Efficiency comparison for part II, case 1

	GPRS	ADETL[p]	ADETL[b]	$\frac{ADETL[p]}{ADETL[b]}$	$\frac{ADETL[b]-GPRS}{GPRS}$
Time step	95	88	88		
Newton Iteration	467	413	413		
Linear Iteration	467	413	413		
Discretization	1.95	4.38	2.30	**1.91**	+17.90%
Linear Solver	3.25	2.35	2.34	1.00	-27.92%
Stability+Property	10.04	53.31	15.06	**3.54**	+50.02%
Total Time	16.44	69.04	23.66	**2.92**	+43.87%

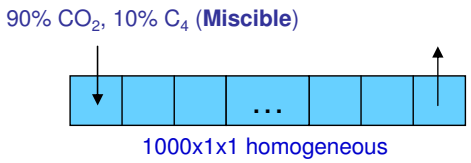


Figure 6.3: Schematic diagram of part II, case 2

stability and property calculation parts with the most local computations, and the second largest improvement (2.09 times) is in the discretization part with stenciling operations.

In this case, **ADETL[b]** uses 33.24% more simulation time than **GPRS**, which is quite encouraging. We saved a lot of time and energy in the development and debugging stages of the simulator by eliminating the human resource that would be spent on generating the Jacobian matrices that correspond to the complex compositional formulation being investigated.

6.3 Two Wells with a High Permeability Channel

Case Description: (see Figure 6.5)

- 50x50x1 with high permeability channel (100x anisotropy), CO_2, C_1, C_2, C_3, NC_4, NC_5, C_6, C_8, C_{10} at 76 bar, 372 K

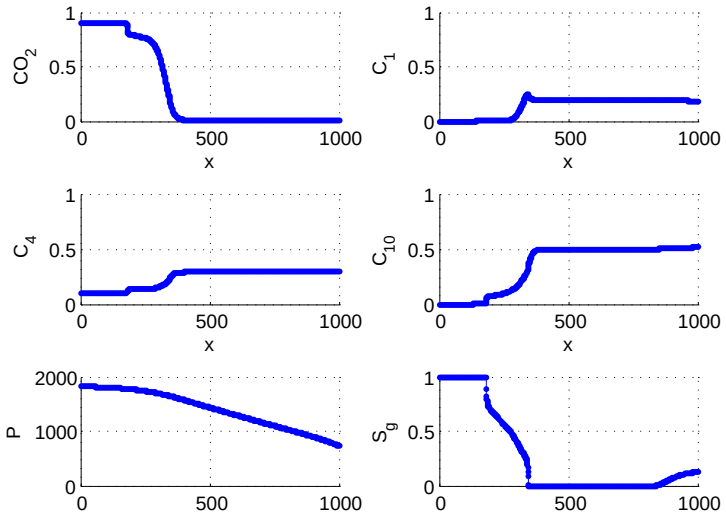


Figure 6.4: ADETL[b] solution of part II, case 2

Table 6.2: Efficiency comparison for part II, case 2

	GPRS	ADETL[p]	ADETL[b]	$\frac{ADETL[p]}{ADETL[b]}$	$\frac{ADETL[b]-GPRS}{GPRS}$
Time step	152	144	144		
Newton Iteration	727	627	627		
Linear Iteration	727	627	627		
Discretization	3.50	6.61	3.17	**2.09**	-9.28%
Linear Solver	4.62	3.50	3.50	1.00	-24.16%
Stability+Property	17.18	85.27	23.42	**3.64**	+36.29%
Total Time	27.05	108.45	36.04	**3.01**	+33.24%

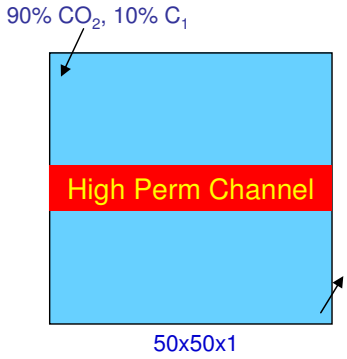


Figure 6.5: Schematic diagram of part II, case 3

- One injector at (1,1,1): 90% CO_2, 10% C_1, BHP control at 140 bar
- One producer at (50, 50,1): BHP control at 30 bar
- Run for 1000 days with maximum time step of 20 days

Figure 6.6 shows the saturation map at the end of the simulation obtained by **ADETL[b]**, from which we can see that the flow scenario is largely affected by the high permeability channel. The efficiency comparison is shown in Table 6.2, from which a similar conclusion can be drawn about the improvement in the different parts.

In this multicomponent case, **ADETL[b]** uses only 5.00% more simulation time than **GPRS**, which is rather encouraging. This overhead is negligible from a practical point of view.

6.4 SPE10 Top Layer, Five-spot Pattern

Case Description: (see Figure 6.7)

- 60x220x1 heterogeneous, 1% CO_2, 20% C_1, 29% C_4, 50% C_{10} at 76 bar, 372K
- One injector at (30,110,1): 90% CO_2, 10% C_1, BHP control at 126 bar
- Four producers at (1,1,1), (60,1,1), (1,220,1) and (60,220,1): BHP control at 30 bar

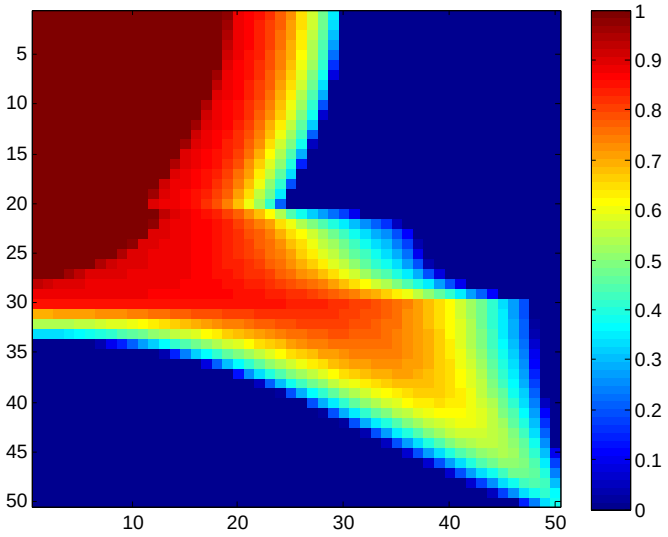


Figure 6.6: ADETL[b] solution of part II, case 3

Table 6.3: Efficiency comparison for part II, case 3

	GPRS	ADETL[p]	ADETL[b]	$\frac{ADETL[p]}{ADETL[b]}$	$\frac{ADETL[b]-GPRS}{GPRS}$
Time step	61	59	59		
Newton Iteration	285	273	273		
Linear Iteration	1074	1063	1063		
Discretization	22.46	45.67	17.23	**2.65**	-23.21%
Linear Solver	31.07	32.44	32.10	1.01	+3.31%
Stability+Property	75.47	365.37	68.73	**5.32**	-8.93%
Total Time	130.60	500.59	137.13	**3.65**	+5.00%

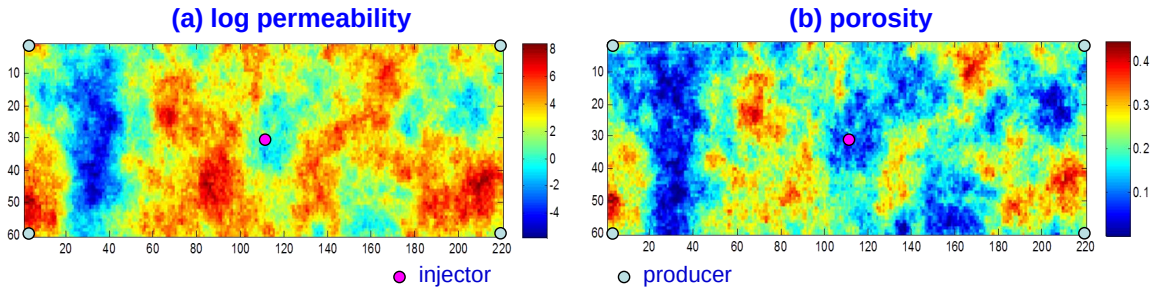


Figure 6.7: Schematic diagram of part II, case 4

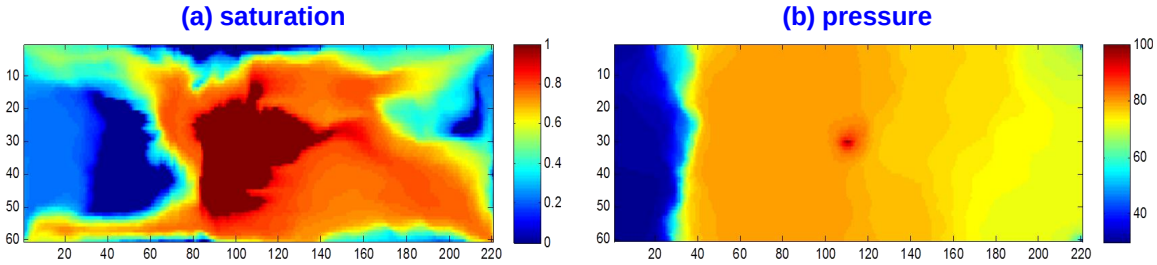


Figure 6.8: ADETL[b] solution of part II, case 4

- Run for 3000 days with maximum time step of 20 days

Figure 6.8 shows the saturation and pressure maps at the end of the simulation obtained by **ADETL[b]**.

The efficiency comparison in Table 6.4 shows similar behavior among the various parts. Notice that **ADETL[b]** spends 121.56% more time in stability and property calculation and 44.9% more time in the entire simulation compared with **GPRS** for this heterogeneous case, which is reasonable but not optimal. This case shows that further improvements of ADETL are needed.

6.5 Four Wells with High Permeability Channels

Case Description: (see Figure 6.9)

- 50x50x5 with high permeability channels (100x anisotropy) and gravity, 1% CO_2, 20% C_1, 29% C_4, 50% C_{10} at 76 bar, 372 K

Table 6.4: Efficiency comparison for part II, case 4

	GPRS	ADETL[p]	ADETL[b]	$\frac{ADETL[p]}{ADETL[b]}$	$\frac{ADETL[b]-GPRS}{GPRS}$
Time step	179	167	167		
Newton Iteration	804	661	661		
Linear Iteration	4333	3567	3567		
Discretization	119.94	235.48	120.57	**1.95**	+0.52%
Linear Solver	179.81	191.33	185.48	1.03	+3.15%
Stability+Property	122.38	853.09	271.14	**3.15**	+121.56%
Total Time	510.57	1610.35	739.84	**2.18**	+44.90%

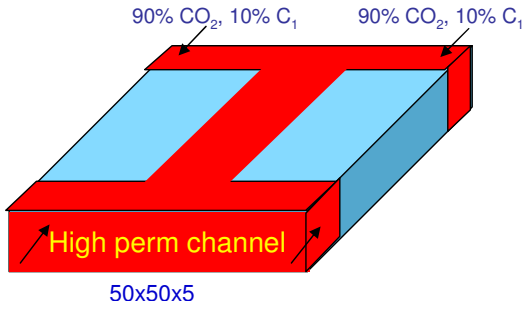


Figure 6.9: Schematic diagram of part II, case 5

- Two injectors at (1,1,1) and (50, 1, 1): 90% CO_2, 10% C_1, BHP control at 126bar
- Two producers at (1,50,5) and (50,50,5): BHP control at 50 bar
- Run for 30 days with maximum time step of 2 days

This case is considered as the most challenging one, since gravity and high permeability channels with large anisotropy are involved in this three-dimensional flow field. The saturation map at the end of the simulation obtained by **ADETL[b]** is shown in Figure 6.10.

From Table 6.5, we can see that **ADETL[b]** uses 3.64% less total time than **GPRS**, which is a quite exciting result! This is not only due to the large improvements in the expression building and evaluation stages, but also due to much better

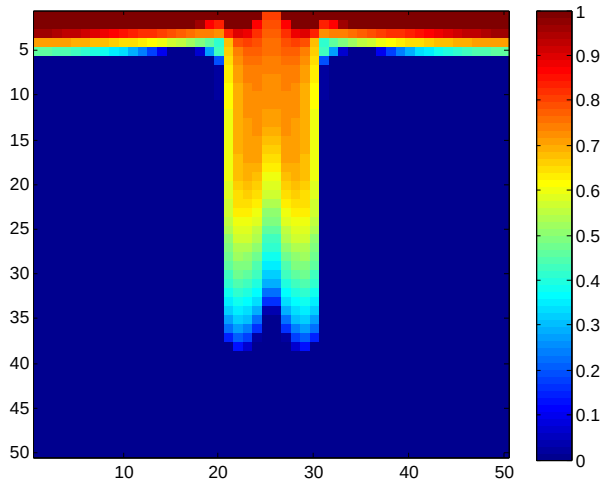


Figure 6.10: ADETL[b] solution of part II, case 5

nonlinear behavior shown in this case. The excellent nonlinear behavior is achieved by using completely rigorous and flexible variable-set switching, which can be much more conveniently realized using the ADETL framework. This is because only independent variable declaration and residual computation need to be coded for the various variable sets, and the derivatives can be automatically generated and assembled into the Jacobian matrix.

Table 6.5: Efficiency comparison for part II, case 5

	GPRS	ADETL[p]	ADETL[b]	$\frac{ADETL[p]}{ADETL[b]}$	$\frac{ADETL[b]-GPRS}{GPRS}$
Time step	35	27	27		
Newton Iteration	168	127	127		
Linear Iteration	634	480	480		
Discretization	22.98	77.23	23.08	**3.35**	+0.40%
Linear Solver	42.91	46.38	46.10	1.01	+7.45%
Stability+Property	109.04	342.81	76.11	**4.50**	-30.20%
Total Time	177.31	523.27	170.85	**3.06**	-3.64%

Chapter 7

Conclusions and Future Work

In the first part (Multistage Preconditioner for Well Groups), the following work has been performed:

- The CPR Multi-stage Preconditioner has been extended to deal with constraints applied on well groups.
- The AIM formulation and well-control switching are now fully supported by the block GMRES solver with adaptive data structures.
- Initial junction-pressure estimation for group-rate control and the junction-pressure constraint with initial rate estimation have been designed and implemented for well groups.
- Large heterogeneous problems with group constraints on multi-segment wells have been demonstrated.

With the extended multi-stage preconditioner for group constraints on multi-segment wells, adaptive data structures, and enhanced well group model, GPRS can run fully coupled reservoir-facilities simulation, including standard, multi-segment wells and well groups, with either FIM or AIM, in an efficient and accurate manner.

Future work for this part may include: (1) further extend the CPR preconditioner to deal with compositional multi-segment wells and group constraints applied on them; (2) extend CPR to thermal-compositional problems either by forming a

pressure-temperature system in the first stage, or by adding a stage to solve the temperature system between the current two stages; (3) analyze the impact of the reduction strategy (true- or quasi- IMPES) on the quality of CPR preconditioning; (4) study the time stepping and implicitness labeling strategy for the AIM formulation in order for it to work better in large heterogeneous cases.

In the second part (Automatic Differentiation for Next Generation GPRS), we first reviewed the automatic differentiation method as an approach to generate Jacobians for reservoir simulation. Then, the ADETL framework was investigated from various aspects including its design objective, usage, working mechanism, and components. Based on that, two recent advances of ADETL are discussed in detail:

- Expression building stage: three customized allocators (specialized freelist, non-recyclable coherent, and recyclable coherent) have been developed based on memory usage patterns. Their advantages and limitations are analyzed with the help of benchmarks of expression building.

- Expression evaluation stage: block sparse data structure with a case-independent block size (e.g., 4) has been designed and implemented. Then suggestions on its usage are discussed based on the results of both practical simulation cases and benchmarks of expression evaluation using different algorithms.

Moreover, the ADETL-based prototype simulator with both the original common free list, point-sparse option, and the new non-recyclable, block-sparse option are tested using challenging compositional simulation problems using 1D, 2D, 3D homogeneous and heterogeneous fields with 4 to 9 components, high permeability channels, and gravity. Compared with the latest GPRS (without AD), the overhead of ADETL simulator ranges from -4% to 45%.

Automatic differentiation is a flexible and effective approach to build complex reservoir simulators. The next generation GPRS will be ADETL-based. ADETL is a modern optimized generic library that is large and growing. It is designed to support multiple data-structures and algorithms with excellent extendability. ADETL enables not only automatic computation of Jacobian behind the scenes but also rapid modification of choice of variables. More effort will be spent on ADETL to make it

more powerful, efficient and flexible. Based on our findings, the next-generation of GPRS will be based on the ADETL library.

The following suggestions can be taken into consideration in the future: (1) design new 'evaluating algorithm' and data structure to support the combined usage of point-sparse and block-sparse vectors; (2) investigate the dense data structure for single-block based computation and direct evaluation capability for short expressions; (3) develop Latex-style expression generation for automatically computed gradients in order to provide debugging and analytic information; (4) establish a general framework, which provides high flexibility, efficiency and extendability, for the next generation GPRS based on ADETL.

Appendix A

A Concrete Example of ADETL

In this appendix, a simple two-phase (gas, oil) black-oil simulator based on ADETL is demonstrated, including the code that calls ADETL and the corresponding Jacobian generation process that takes place behind the scenes.

For simplicity, we use a 2D 6-cell (3x2) model without gravity, capillarity or wells. Given the pressure and gas saturation from the last time step (P^n, S_g^n) and previous Newton iteration for the current time step ($P^{n+1,k}$, $S_g^{n+1,k}$), the residual vector for the current Newton iteration $(n+1, k+1)$ is computed. At the same time, the Jacobian matrix is automatically generated. This process will be analyzed in detail. The problem setting is as follows:

- Reservoir description
  - Reservoir dimensions: Lx = Ly = 2100 ft, Lz = 300 ft
  - Permeability: kx = ky = [200 200 200 300 300 300]
  - Porosity: $\phi = 0.25 \cdot exp(c_R * (P - P_{SC}))$, $c_R = 0$
- Fluid properties
  - $P_{bub} = 3000$ psi, $P_{SC} = 14.7$ psi
  - Oil viscosity, constant: $\mu_o = 2$ cp
  - Gas viscosity: $\mu_g = 3e-10 \cdot P^2 + 1e-6 \cdot P + 0.133$ [cp]

  - Formation volume factor: $Dp = \begin{cases} P_{SC} - P, & \text{if } P < P_{bub} \\ P_{Sc} - P_{bub}, & \text{if } P \geq P_{bub} \end{cases}$
    $b_o = 1/B_o = exp(-(-8.5e-5) \cdot Dp)$
    $b_g = 1/B_g = exp(-4.406e-4 \cdot Dp)$
  - Gas oil ratio: $Rs = \begin{cases} (P/P_{bub})^{1.25}, & \text{if } P \leq P_{bub} \\ 1, & \text{if } P > P_{bub} \end{cases}$
  - Relative permeability: $k_{ro} = (1 - S_g)^{1.5}$, $k_{rg} = S_g^2$

- Conditions given at last time step (n) and last Newton iteration $(n+1, k)$:
  - P^n = [2000 2000 2010 1950 1960 1960]
  - S_g^n = [0.0 0.1 0.2 0.3 0.4 0.5]
  - $P^{n+1,k}$ = [2005 2005 2015 1955 1965 1965]
  - $S_g^{n+1,k}$ = [0.1 0.2 0.3 0.4 0.5 0.6]
  - $\Delta t = 1$ day

Now we analyze the simulator step by step. Prior to the usage of ADETL, we must have the problem properly initialized, which is pretty the same as what we would do in the traditional environment, except that the variables that would have derivatives later (e.g., R, b_o) are declared as ADvector's. Usually the data are read from some input file, whereas they are directly set here for simplicity as follows:

```
// initialization
const int NX = 3, NY = 2, NB = NX * NY;
const int nconn = (NX - 1) * NY + NX * (NY - 1);
const double dx = 700, dy = 1050, dz = 300;
connection_data * conn = new connection_data[nconn];
// ... some code to setup the connection ...
double* volume = new double [NB];
double* poro_ini = new double [NB];
// ... some code to setup initial volume and porosity ...
```

```
const double cR = 0.0, cO = -8.5e-5, cG = 4.406e-4;
// ... some code to set other double constants: vo_A, vo_B, vo_C, vg_A,
vg_B, vg_C, nRs, pSC, pbub, kroo, krgo, no, ng ...

adetl::ADvector R(NB * 2, 0.0); // residual vector
adetl::ADvector poro(NB, 0.0), bo(NB, 0.0), bg(NB, 0.0);
adetl::ADvector rs(NB, 0.0), Ho(NB,0.0), Hg(NB, 0.0);
double * Accu_on = new double[NB];
double * Accu_gn = new double[NB];

// Initial distribution
adetl::ADvector P(NB, 2005.0), Sg(NB, 0.0);
double* Pn = new double[NB];
double* Sgn = new double[NB];
P[2] = 2015; P[3] = 1955; P[4] = 1965; P[5] = 1965;
// ... some code to set other initial distributions: Pn, Sg, Sgn ...
```

We utilize ADETL as described in Section 4.3.2. The first step is to declare the independent variables:

```
// STEP 1: Declare independent variables
for(i = 0; i < NB; ++i) {
    P[i].make_independent(2 * i);
    Sg[i].make_independent(2 * i + 1);
}
```

For each block, we make pressure the $2n^{th}$ independent variable and saturation the $(2n+1)^{th}$ independent variable ($n = 0, 1, ..., NB-1$) so that each of them has only one derivative (of value 1.0) with respect to itself (e.g., P[3] = { 1955, [. . . . . 1.0 . . . . .] }) and the derivative is stored in the user-specified format (either point sparse or block sparse).

In the second step, we write the residual code. This is the core part of the simulator

and can be further divided into several sub-steps as follows:

1. For each time step, first save the information from the last time step

```
// STEP 2: Write residual code
// ... For each time step, do: ...
// Save accumulation term of last time step
for(i = 0; i < NB; ++i) {
    const double Dpn = (Pn[i] < pbub) ? (pSC - Pn[i]) : (pSC - pbub);
    const double poro_n = poro_ini[i] * exp(cR * (Pn[i] - pSC));
    const double Rs_n = (Pn[i] <= pbub) ? pow(Pn[i] / pbub, nRs) : 1.0;
    Accu_on[i] = poro_n * exp(-cO * Dpn) * (1.0 - Sgn[i]);
    Accu_gn[i] = poro_n * exp(-cG * Dpn) * Sgn[i] + Rs_n * Accu_on[i];
}
```

   In this sub-step, the computations are all traditional double operations, since the terms of last time step would have no derivatives at all. Thus this piece of code would be exactly the same as its counterpart in the traditional environment.

2. For each Newton iteration, first calculate the various physical properties
   In this sub-step, various properties, which would be used later for evaluating the residuals, are computed. Here we notice that the conditional branch can be handled well by ADETL. If the pressure is below the bubble point pressure, Dp is set to pSC-P[i], and its derivative is automatically computed as -(P[i])' (e.g., for i = 3, Dp = { 14.7 - 1955, -1.0 * (P[i])' } = { -1940.3, [. . . . . -1.0 . . . . .] }), while if the pressure is above the bubble point, Dp would be set to pSC-pbub, which is a constant so that Dp would have no derivatives.

   Since these properties (porosity (poro), the inverses of the oil and gas formation volume factors (bo, bg), gas-oil ratio (Rs), and the flow parts in the oil and gas transmissibilities (Ho, Hg)) are declared as ADvector's, their derivatives will be automatically computed by ADETL (e.g., for i = 3, bo[i] = { exp(8.5e-5 * (-1940.3)), exp(8.5e-5 * (-1940.3)) * 8.5e-5 * (Dp)' } = { 0.8480, [. . . . .

. -7.2076e-5 . . . . .] }). Note that with this AD based approach, we do not write the code for evaluating the derivatives of these properties, which is required by the traditional approach.

```
// ... For each Newton iteration, do: ...
// Properties calculation for current iteration
for(i = 0; i < NB; ++i) {
    ADscalar Dp;
    if(P[i] < pbub) Dp = pSC - P[i]; else Dp = pSC - pbub;
    poro[i] = poro_ini[i] * exp(cR * (P[i] - pSC));
    bo[i] = exp(-cO * Dp); bg[i] = exp(-cG * Dp);
    if(P[i] <= pbub) Rs[i] = pow(P[i] / pbub, nRs); else Rs[i] = 1.0;

    const ADscalar kro = kroo * pow(1.0 - Sg[i], no);
    const ADscalar krg = krgo * pow(Sg[i], ng);
    const ADscalar vo = vo_A * P[i] * P[i] + vo_B * P[i] + vo_C;
    const ADscalar vg = vg_A * P[i] * P[i] + vg_B * P[i] + vg_C;
    Ho[i] = kro * bo[i] / vo;
    Hg[i] = krg * bg[i] / vg + Rs[i] * Ho[i];
}
```

3. Calculate accumulation terms

```
 // Accumulation Terms
for(i = 0; i < NB; ++i ) {
    const double coef = -volume[i] / (dt * 5.615);
    R[2 * i] = coef * (poro[i] * bo[i] * (1.0 - Sg[i]) - Accu_on[i]);
    R[2 * i + 1] = coef * (poro[i] * (bg[i] * Sg[i] + Rs[i] * bo[i] * (1.0 - Sg[i])) -
Accu_gn[i]);
}
```

In this sub-step, the accumulation terms of the oil and gas equations for each block are computed and saved in the residual vector **R**. The code looks the

same as its corresponding part in the traditional environment. But since **R, poro, bo, bg, Rs** are all ADvector's, the corresponding derivatives of **R** are automatically computed by ADETL. For instance, for i = 3, we have:

$$\begin{aligned} R[2*i] &= \{-3.9270e7*(0.25*0.8480*(1.0-0.4)-0.1485), \\ &\quad -3.9270e7*(0.25*(0.8480*(-1.0)*(Sg[i])'+(1.0-0.4)*(bo[i])'))\} \\ &= \{8.3495e5, 8.3248e6*(Sg[i])'-5.8905e6*(bo[i])'\} \\ &= \{8.3495e5, [\ .\ .\ .\ .\ .\ .\ 424.5636\ 8.3248e6\ .\ .\ .\ .\ ]\}, \end{aligned}$$

where we see that the accumulation term has only local derivatives (with respect to the pressure and saturation of the block itself).

4. Calculate flux terms

```
// Flux Terms
for(i = 0; i < nconn; ++i) {
    const int ia = conn[i].ia, ib = conn[i].ib;
    const int up_i = (P[ia] >= P[ib]) ? ia : ib;
    const ADscalar Oflux = conn[i].trans * Ho[up_i] * (P[ib] - P[ia]);
    R[2 * ia] += Oflux;
    R[2 * ib] -= Oflux;

    const ADscalar Gflux = conn[i].trans * Hg[up_i] * (P[ib] - P[ia]);
    R[2 * ia + 1] += Gflux;
    R[2 * ib + 1] -= Gflux;
}
```

In this sub-step, the flux terms in the oil and gas equations for each connection are computed and added to the corresponding residual elements. We can see that upwinding is treated quite naturally and no error is incurred due to having a conditional branch. Again, the code looks similar to that in a traditional environment, but here the derivatives of **R** are automatically computed and

updated. For instance, the oil equation residual for block #3, R[6], is updated at two connections (3 - 4 in x-direction and 0 - 3 in y-direction) as follows:

$$\begin{aligned}
R[6] &= R[6] + 152.145 * Ho[4] * (P[4] - P[3]) \\
&= \{8.3495e5 + 152.145 * 0.1498 * (1965 - 1955), \\
&(R[6])' + 152.145 * (0.1498 * ((P[4])' - (P[3])') + (1965 - 1955) * (Ho[4])')\} \\
&= \{8.3518e5, (R[6])' + 22.79 * (P[4])' - 22.79 * (P[3])' + 1521.45 * (Ho[4])'\} \\
&= \{8.3518e5, [\ .\ .\ .\ .\ .\ .\ 401.7766\ 8.3248e6\ 22.7676\ -683.6102\ .\ .\ ]\};
\end{aligned}$$

$$\begin{aligned}
R[6] &= R[6] - 54.096 * Ho[0] * (P[3] - P[0]) \\
&= \{8.3518e5 - 54.096 * 0.3605 * (1955 - 2005), \\
&(R[6])' - 54.096 * (0.3605 * ((P[3])' - (P[0])') + (1955 - 2005) * (Ho[0])')\} \\
&= \{8.3616e5, (R[6])' - 19.50 * (P[3])' + 19.50 * (P[0])' + 2704.8 * (Ho[0])'\} \\
&= \{8.3616e5, [\ 19.42\ -1624.97\ .\ .\ .\ .\ 382.27\ 8.3248e6\ 22.77\ -683.61\ .\ .\ ]\},
\end{aligned}$$

from which we can find out that the flux terms add stenciling derivatives (with respect to pressure and saturation of blocks connected to the target block) to the residual elements.

Now, the second step (write residual code) is complete, and no code is needed at all for generating the Jacobian matrix.

The third step is to use the automatically generated gradients. As discussed in chapter 4, the values and gradients of the residual elements are stored in each ADscalar of the ADvector **R**. We can write some code to convert **R** to the format used by the solver (e.g., sparse Jacobian matrix **J** and full double vector **Rv**), solve $\mathbf{J} \cdot \mathbf{Xv} = \mathbf{Rv}$, and update the value part of **P** and **Sg** using **Xv**:

```
// ... some code to convert R into J and Rv ...
// ... some code to solve J * Xv = Rv for Xv ...
for(i = 0; i < NB; ++i) {
    P[i].value() -= Xv[2 * i];
    Sg[i].value() -= Xv[2 * i + 1];
}
```

Up to now, one Newton iteration is complete, and we can repeat this until the system converges, and then proceed to the next time step. More features (e.g., well terms, gravity, capillarity, and three phases) can be added to this simple example by changing the residual code (or declaration of independent variables, if necessary) only. Great flexibility can be achieved and no manual effort by the client code is needed for the Jacobian generation.

Bibliography

[1] Y. Jiang. *Techniques for Modeling Complex Reservoirs and Advanced Wells*. PhD thesis, Stanford University, 2007.

[2] R.M. Younis. *Advances in Modern Computational Methods for Nonlinear Problems; A Generic Efficient Automatic Differentiation Framework, and Nonlinear Solvers That Converge All The Time*. PhD thesis, Stanford University, 2009.

[3] J.R. Wallis. Incomplete gaussian elimination as a preconditioning for generalized conjugate gradient acceleration. In *SPE Reservoir Simulation Symposium*. SPE 12265, 1983.

[4] J.R. Wallis, R.P. Kendall, T.E. Little, and J.S. Nolen. Constrained residual acceleration of conjugate residual methods. In *SPE Reservoir Simulation Symposium*. SPE 13536, 1985.

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

[6] H. Cao, H.A. Tchelepi, J.R. Wallis, and H. Yardumian. Parallel scalable CPR-type linear solver for reservoir simulation. In *SPE Annual Technical Conference and Exhibition*. SPE 96809, October 2005.

[7] W.L. Briggs, V. Henson, and S.F. McCormick. *A Multigrid Tutorial, 2nd Edition*. SIAM, 2000.

[8] Y. Jiang and H. A. Tchelepi. Scalable multi-stage linear solver for coupled systems of multi-segment wells and complex reservoir models. In *SPE Reservoir Simulation Symposium*. SPE 119175, February 2009.

[9] K. Stueben. Algebraic multigrid (AMG): Experiences and comparisons. *Applied mathematics and computation*, 13(3-4):419–451, 1983.

[10] F. Kwok. A block ILU(k) preconditioner for GPRS. Technical report, Stanford University, 2004.

[11] Schlumberger GeoQuest. Eclipse technical description 2005. Technical report.

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

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

[14] G. Acs, S. Doleschall, and E. Farkas. General purpose compositional model. *SPE Journal*, pages 543–553, August 1985.

[15] M.C.H. Chien, H.E. Yardumian, E.Y. Chung, and W.W. Todd. The formulation of a thermal simulation model in a vectorized, general purpose reservoir simulator. In *SPE Reservoir Simulation Symposium*. SPE 18418, February 1989.

[16] P. Quandalle and J.C. Sabathier. Typical features of a multipurpose reservoir simulator. *SPE Reservoir Engineering*, 4(4):475–480, 1989.

[17] P. Moin. *Fundamentals of Engineering Numerical Analysis*. Cambridge University Press, 2001.

[18] T.R. Jensen and B. Toft. *Graph Coloring Problems*. Wiley-Interscience, 1995.

[19] A. H. Gebremedhin, F. Manne, and A. Pothen. What color is your jacobian? graph coloring for computing derivatives. *SIAM Review*, 47(4):629 – 705, December 2005.

[20] A. Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*. Kluwar Academic Publishers, IL, 1990.

[21] Wikipedia, http://en.wikipedia.org/wiki/List_of_differentiation_identities. *List of differentiation identities*.

[22] L.B. Rall. Perspectives on automatic differentiation: Past, present and future. In *Automatic Differentiation: Applications, Theory and Implementations, Lect. Notes in Comp. Sci. and Eng.* Springer, 2005.

[23] H. Fischer. Special problems in automatic differentiation. In A. Griewank and G.F. Corliss, editors, *Automatic Differentiation of Algorithms*. SIAM, PA, 1991.

[24] M. Bcker, G. Corliss, P. Hovland, U. Naumann, and B. Norris. *Automatic Differentiation: Applications, Theory and Implementations, Lect. Notes in Comp. Sci. and Eng.* Springer, 2006.

[25] C.H. Bischof, H.M. Bucker, P. Hovland, U. Naumann, and J. Utke. *Advances in Automatic Differentiation, Lect. Notes in Comp. Sci. and Eng.* Springer, 2008.

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

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

[28] G. Corliss, C. Bischof, A. Griewank, S. Wright, and T. Robey. Automatic differentiation for PDE's: Unsaturated flow case study. In *Advances in Computer Methods for Partial Differential Equations - VII*. IMACS, 1992.

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

[30] C. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic dfiferentiation of fortran 77 programs. *IEEE Computational Science and Engineering*, 3:18–32, 1996.

[31] N. Rostaing, S. Dalmas, and A. Galligo. Automatic differentiation in odyssee. *Tellus*.

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

[33] A. Griewank, D. Juedes, and J. Utke. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, 22:131–167, 1996.

[34] D. Shiriaev. ADOL-F: Automatic differentiation for fortran codes. In M. Berz, C.H. Bischof, G.F. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, 1996.

[35] D. Bulka and D. Mayhew. *Efficient C++: Performance Programming Techniques*. Addison-Wesley, MA, 2000.

[36] S.B. Lippman, J. LaJoie, and B.E. Moo. *C++ Primer (4th Edition)*. Addison-Wesley Professional, 2005.

[37] R. Vuduc, J.W. Demmel, and K.A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16:521+, 2005.

[38] D.V. Voskov, R.M. Younis, and H.A. Tchelepi. General nonlinear solution strategies for multi-phase multi-component EoS based simulation. In *SPE Reservoir Simulation Symposium*. SPE 118996, February 2009.