THE UNIVERSITY OF CHICAGO

FINITE ELEMENT METHOD AUTOMATION FOR NON-NEWTONIAN FLUID
MODELS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

ANDY R. TERREL

CHICAGO, ILLINOIS

AUGUST 2010

To my family, most especially:

my wife, Cheryl,

my mother, Judy, and

my late grandfather, Virgil.

To all the educators who have helped me along my path, most especially:

my high school math teacher, Ms. Swauger, and

chemistry teacher, Ms. Pirtle.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

Both of my advisors, Matthew G. Knepley and L. Ridgway Scott, as well as my previous advisor, Robert C. Kirby, have been a vital part of my graduate education. Without their guidance to the exciting field of automated scientific computing and computational fluids, this thesis would not exist. Graciously, they have not only tutored me through many concepts in these fields but also opened many career opportunities.

I would like to acknowledge the PETSc and FEniCS software developers. Without their wonderful products this work would not be possible. Additionally, Dmitry Karpeev, Anders Logg, Kevin Long, and Garth N. Wells for their guidance in comments and support for this work.

ABSTRACT

Finite element methods (FEMs) are a popular simulation technique for discretizing partial differential equations (PDEs) to an algebraic problem. For well-studied problems, theory shows which methods are optimal and can be applied to a wide variety of applications. Building a FEM model for a new or complex problem can be a labor intensive, error prone task. The difficulties have been somewhat alleviated by automation, but the tools often lag in performance to hand optimized, specialized code. Because of the ease of specifying new models and solver techniques in automated solvers, it is an ideal place to look at challenging problems and evaluate many methods. By developing and using automation techniques, we study a challenging class of problems: complex fluids.

CHAPTER 1

INTRODUCTION

Over the past half century, the pursuit of scientific discoveries has increasingly become dependent on computation. As computers increase in power, scientific simulations increase in size. The ability to run increasingly larger simulation makes a wider ranges of applications feasible. This rapid development of technology has led many in the field to refer to computation as the third tier of science, joining experimentation and theory.

Unfortunately, the cost of increasing the power of the computer is programability. To use state-of-the-art simulation a user must be an expert in large parallel systems. These systems are often incredibly sensitive to the algorithm used for the simulation, often requiring major rethinking to gain scalability from one system to the next. The complexity of the hardware encourages the divide between practice and theory, which further requires expertise to develop large scale codes.

Reducing the programming barrier and increasing the ability of the scientist to use more complex methods is a rich area of study and is being analyzed from many different perspectives. Programming language researchers are developing methods to parallelize codes with little change to existing code. Library writers are constantly creating better interfaces that separate the concerns of the machine and the user. Theoreticians are investigating various approaches to incorporate the attributes of complex algorithms into the existing scalable frameworks. While each of these approaches are able to help lower the barrier of programming, each group tends to be isolated and advances do not always trickle from one discipline to the other.

Automated scientific computing is a growing field that is poised to bridge the gaps between these fields. Automated scientific computing uses mathematical expressiveness at a high level, applies compiler techniques to manipulate the high level representation, and generates low level source designed for specific systems. This process of mechanically applying

the knowledge of computational expert equips the scientist with tools to try more methods and models effortlessly.

In this thesis we present and apply the principles of automated scientific computing to finite element methods (FEMs). There are many variants to the methods. The more difficult the implementation of these methods, which are often required by applications, the less likely numerical comparisons are made. The automation of FEM facilitates the building of application libraries where the underlying models can be changed very easily. The code can simply generate a new low level code using the best method for the particular simulation.

To demonstrate this ability we developed a code, Rheology Application Engine or Rheagen, to simulate non-Newtonian fluid models. We use methods and models that span six decades and are able to generate side by side comparisons of models and numerical methods. Prior to this work, such as study could only compare notes from different work in the literature, which almost never include identical simulations. The hope of this study is to provide the community with a tool for testing new non-Newtonian fluid models and methods.

Additionally we use the generative capabilities of our system to evaluate polynomial solvers. Because many models may not have nice theoretical results proven about them, our hope was to use general polynomial solvers to gain insight to small problems. While current technology is still quite limited in the number of polynomials that can be solved, we provide a small proof of concept with a few contrived examples.

1.1   Finite Element Methods

The first publication of FEM is usually attributed to Courant in 1943 [21] where he summarized the use of Ritz-Rayleigh methods for several other simulation techniques and only outlined a FEM method in an appendix. The first textbooks appeared in the 1970s focusing on a range of applications [67, 75] and numerous popular texts have been produced since.

The heart of the every FEM is assembling a variational formulation of a PDE into alge-

---
Algorithm 1.1 Simple finite element method

 1: Formulate mesh.
 2: Create degrees of freedom (dofs) on mesh.
 3: Evaluate $a(i, j)$ and $L(j)$ for$i, j \in$ dofs.
 4: Apply Dirichlet boundary conditions.
 5: Perform algebraic solve.
---

braic equations and solving them, see Algrothim 1.1. The Ciarlet definition [18] gives the us the first meaning of an element $(\mathcal{K}, \mathcal{P}, \mathcal{N})$:

- $\mathcal{K} \subseteq \mathbb{R}^n$ is the element domain,

- $\mathcal{P}$ is the space of shape functions, and

- $\mathcal{N}$ (the basis or dual of P) is the set of nodal variables.

The domain is "meshed", that is tiled by the element shape $\mathcal{K}$, and each tile is also called an element. On each of these elements, the functions are represented via coefficients of the basis of $\mathcal{P}$, or degrees of freedom (dofs). For each dof, an algebraic equation is formed by the variational form, which consists of the bilinear and linear forms:

$$a(u, v) = L(v) \tag{1.1}$$

where u is the trial function being solved for and v is the trial function. To form these algebraic equations, $a(\phi_i, \psi_j)$ and $L(\psi_j)$ is evaluated for each $\phi_i, \psi_j \in \mathcal{N}$ where the index i correspond to the dofs representing the solution u. Most PDEs require the application of boundary conditions. These are either weak conditions, such as Neumann conditions that are assembled through the variational form, or strong conditions, such as Dirichlet which fix some of the coefficients eliminating the corresponding equation.

Algorithm 1.1 requires an element $(\mathcal{K}, \mathcal{P}, \mathcal{N})$, a variational form $(a(\cdot, \cdot)$ and $L(\cdot))$, a domain, boundary conditions and a algebraic solver to output the solution of the PDE. This view of FEM is highlights some major separations in the field, namely that between

3

Algorithm 1.2 FEM using a reference element

1: **for** each element **do**
2:     Evaluate $a^e(i,j)$ and $L^e(j)$ for $i,j \in$ reference dofs.
3:     Transform $a^e(i,j)$ and $L^e(j)$ to parital result of $a(i,j)$ and $L(j)$
4:     Accumulate with previous partial results for $a(i,j$ and $L(j)$
5: **end for**

---

meshing the domain, representing the functions, formulating solvable PDEs, and solving the algebraic system. This formulation of FEM has been widely successful for theory to define and analyze the different aspects of FEM. Unfortunately, where mathematics will allow the easy substitution of any of these inputs, implementations do not.

For example, the first improvement to Algorithm 1.1 is to transform to a reference space. Thus replacing line 3 with Algorithm 1.2. Here the variational equations are computed on the reference space and transformed to correct element. In the case of a continuous finite elements, the dofs are shared on the intersection of two elements, so the equations of these element must be accumulated. For simple scalar elements such as Lagrange elements, the transform is affine and the accumulation is only summation. For the Raviart-Thomas element, which requires normal vectors, adjacent element normals must be transformed to match directions; it is the same for Nédélec elements of the first kind which use the normal trace. $H(\text{div})$ and $H(\text{curl})$ elements require covariant and contravariant transforms, or Piola transforms, and some elements such as Argyris and Mardal-Tae-Winther do not have a well-known transformation to use with a reference.

Already with the first improvement of our algorithm, we see that new logic must be implemented to assemble different types of elements and interfaces must be kept tight to insure the correct implementation. Being able to substitute different parts of FEM is highly beneficial for tackling difficult problems as we discuss in Chapter 3. For this reason , as we discuss in Chapter 2, the development of *mathematical* interfaces between these four parts of the assembly are necessary to give a researcher the full power of FEM for challenging problems.

1.2   Fluid Models

Modeling fluids with FEM is an error prone task with many hurdles that trap novice researchers. Because fluids couple several functions together, it is common to represent each field with a different finite element, or a mixed element. These mixed systems usually lead to several types of problems and have been the study of many decades. These problems have largely been addressed in Newtonian fluids by correctly picking finite elements, stabilizing the variational forms, and developing specialized algebraic solvers. To test the different variations of FEM requires a numerous simulations which leverages the full power of automating the process.

Non-Newtonian fluids add further complexity to the system because of their representation of the stress as a non-linear relation with the fluid velocity. These fluid models have been theoretically and experimentally derived from small scale models, but often have not been proven well posed. By extending FEM automation to this area, we are able to test many different proposals to overcome the difficulties seen by researchers for the last century.

1.3   Algebraic Solvers

A common problem when working with a difficult and unknown PDE is determining whether it is solvable or not. The most common technique for solving a non-linear PDE with FEM is to produce an approximating linear system and iteratively improve the solution with the Newton-Raphson method. The convergence of this method gives the researcher one solution to the proposed PDE. If the equation truly does have multiple solutions, this complexity is not represented in the converged solution. Even if the method does not converge then it is not clear that the problem is not solvable as the error could come from any other part of the simulation.

To address this problem, we developed an automated FEM assembly process that assem-

bles the full non-linear polynomial system. This system has been coupled with polynomial solvers using homotopy continuation methods. These methods provide all solutions to the polynomial system, giving the researcher the assurance that when it produces one solution, it is the only solution of the given system.

CHAPTER 2

AUTOMATED SCIENTIFIC COMPUTING

Scientific software is complex. Every project must make trade-offs between expressiveness and performance. An example of this trade-off is the use of Matlab versus LAPACK. Matlab is an interpreted, dynamic language with a high level view of linear algebra. This high level view makes programming easy problems trivial and difficult problems tractable, even to the point that a non-expert is able to leverage years of research in linear algebra. LAPACK is an interface for a low-level implementation for linear algebra routines. To use LAPACK effectively, the user must know a large amount of details about their problem, such as what type of matrix they have or how it is laid out in memory. A typical work flow is to develop a prototype in Matlab and when a piece of code must be faster to reimplement using LAPACK. This approach is costly in development and maintenance time, so much so that a new paradigm has begun to emerge: automated scientific computing.

Automated scientific computing uses high level, domain specific languages to define the class of problems being solved and generates low level problem specific code. This model of software engineering takes advantage of the expressiveness provided by the high level language and the performance characteristics of the low level code. While low level optimization, such as loop unrolling and blocking, have become standard practice for producing high performance code, automation opens the realm of possibilities to high level optimization, even choosing different algorithms based on architectural specifications.

The successful development of several automated systems have followed a general pattern. The definition of the problem domain, a well defined interface to divide the problem domain, and optimizations within each subdomain of the problem. Each of the examples, which we discuss later in more detail, define a high level language that will generate a low level code. This process can be thought of as domain specific compiling, where mathematical insights have led to a larger number of optimizations than allowed by general programming languages.

In this chapter, we first we discuss two example of automated scientific computing systems. We then go on to discuss the development of automated scientific systems and observe some general trends. These trends set the course for the next chapters analysis of the development of automated FEM systems and the data structures useful in the process.

2.1   Developing an automated system

In each of the cases described, the pattern for developing the automated system is similar. The problem domain is explored with rigorous mathematical detail, interfaces are developed to join the different mathematical structures and operations on those structures, and the final step is the optimization and code generation.

The first two stages of this development may appear identical to that of building software libraries, but because of the different software goals, the software designer chooses a much different approach. An excellent example is seen in the development of SPIRAL [61] versus FFTW [33]. Because SPIRAL generates code for many linear transforms, the interface takes high level descriptions, such as the transform, architecture, and generation language then produces an optimized library. The development of the FFTW focuses on automatically tuning FFT parameters at runtime after executing a variety of benchmarks. This requires a low level interface of buffers and offsets.

2.1.1   Defining the problem domain.

The problem domain is perhaps the first step of developing the automated system, but it is crucial for a project to use the correct abstractions in the right places. Current state-of-the-art scientific software decomposes simulations into numerous middleware projects. It is not unusual for a code to call numerous independent libraries, such as meshing tools, graph partitioners, linear algebra routines, messaging libraries, and so on. Each tool solving a particular problem necessary for the simulation. On the other hand, this separation of

8

problem domains precludes optimizations between the called library and the code calling it. Automation is able to relieve the block created by calling middleware, but it is still important to decompose the problem into appropriate domains.

Because most scientific simulations start as a mathematical model that is then transposed into code, it is natural to decompose the problem according to its mathematical structure. For example, to numerically integration a function, the representation of the function must be known and then the accuracy of the integration decided. If a library only has one way to represent the function then this task is trivial, but this is rarely the case in software. The functions may be represented as a closed expression, as a discrete object, in a different basis, on a number of processors. The possibilities of the representation are quite numerous as are the different integration rules and optimization techniques. By separating the representation of the function and the integration rules, an optimized integration routine can be produced based on the representation or vice versa.

2.1.2   Building interfaces

After the different problem domains are defined, an interface for each must be created. Where the defining of domains is used to separate concerns of the purpose of optimizing different parts of the code, the interfaces can be thought of a way for each domain to communicate when necessary. Important features of a well designed interface include:

- a robust representation,

- a minimal set of requirements to use the interface, and

- a closed system, or closed as possible.

A good interface drives computational science by allowing users to access deep mathematical algorithms very quickly. Automated scientific computing usually creates several

layers for the user to access. At the high level the user is able to quickly define their problem in simple terms giving an advantage of generating a wide range of experiments. At the lower levels the user is able to finely tune their simulations. The goal is to never require a user to need the lower levels and implement optimizations in the generation.

2.1.3 Optimizations and Code Generation

The optimization techniques are the driving process of creating the automated system. By opening up the system with expressive objects a larger range of optimizations are possible and at very least low level optimizations should be reproducible. Because scientific computing is a relatively refined discipline it is not unusual to implement optimizations that are impractical in a general purpose compiler.

By focusing on code generation, the algorithms are able to switch data structures and low level code for the interest of speed. These data structures are also tuned to the specific parameters of the algorithm. For example, a user of the FEniCSlibrary the default representation of the FEM functions is determined at compile time and specific code is generated based on the which representation is predict to run faster.

2.2   Examples

Two example from scientific computing include the FLAME project for dense linear algebra and SPIRAL project for linear transforms. Both projects produce codes for multiple architectures and have optimizations based on desired algorithms. Here we briefly discuss the problem domains, interfaces, and optimizations used by these projects.

2.2.1   Linear Algebra

Linear algebra is one of the oldest scientific computing disciplines. The vast amount of work in the discipline is applied to problems from all parts of scientific query. Because of its wide use, it is one of the most essential parts of numerical software. In software two interfaces have helped capture the major algorithms used in linear algebra: Basic Linear Algebra Subroutines (BLAS) and Linear Algebra PACKage (LAPACK).

BLAS and LAPACK gave researchers a simple interface use to access numerical method and library writers a target to implement that could be dropped in place of their competitors. Numerous libraries implementing these interfaces are available all with different design principles. For example ATLAS [73] is automatically tuned based on runtime performance of the machine when compiled, GOTO BLAS [38] consists of hand tuned assembly code that consistently provide the fastest runtimes on numerous platforms, SCALAPACK [11] focuses on parallel execution, and the list goes on. A relatively new program for creating linear algebra libraries it the Formal Linear Algebra Methods Environment (FLAME) [40].

FLAME provides an interface that allows the researcher to propose a linear algebra algorithm at a high level through a partitioned matrix equation. The library will handle all indices, memory management, and generate code to be called on a particular linear system. The interface can even generate code for the PLAPACK library, another parallel library implementing LAPACK. FLAME also encourages the library writer to write a formal proof along with the implemented algorithm.

The FLAME interface proved not only to be a good way of implementing BLAS and LAPACK routines, where it often beat the current GOTO BLAS, it also created a language that could be automated. Bientinesi's PhD thesis [8] gives a system for mechanically deriving the algorithms based on loop invariants and the partitioned matrix equations. He uses symbolics to manipulate the matrix blocks and determine appropriated orderings of computation. This work presents a new paradigm of working with linear algebra algorithms

with not only tuning a few algorithms but generating the appropriate algorithm. Further advances to this work could in principle query the system it is running on and generate code for particular architectures.

This story of linear algebra is the most complete in automated scientific computing. Interfaces were created, libraries were automatically tuned, new algorithms discovered, and automated code generated. Its success has sparked the pursuit of automated systems in other parts of scientific computing and continues to be a thriving field of research.

2.2.2   Signal Processing

Another success story in the automation of scientific computing comes from signal processing or more specifically linear transforms. Linear transforms such as the Discrete Fourier Transform (DFT) are a major part of numerous scientific simulations. While generalizing to a matrix-vector product, many algorithms have been discovered to reduce the complexity of specific transforms. While this approach worked well for the best known transforms it did not scale well to all transforms an user might need. The SMART [60] and SPIRAL [61] projects moved the process of finding better algorithms and coding them to an automated process.

The SMART Project focuses on the developing an algebraic theory of signal processing. By searching a transform for symmetries and shifts, the transform is represented as an algebra with specific modules [60]. This gives a high level definition of the transform that can be manipulated automatically for computational purposes.

The SPIRAL Project focuses on automatically generating efficient code for signal processing [61]. By defining the transforms in the high level language given by SMART and abstracting of multiple architectures, the project is able to generate optimal code. The SPIRAL code generator works in multiple stages with feedback between each stage. The user defines the particular transform and architecture. An algorithm for computing the transform

12

is chosen. An implementation of the algorithm is produced and optimized to the architecture. Finally the code is compiled and performance is evaluated. At each stage the feedback generates more possibilities to run through the stages allowing for changes in the algorithms and optimizations that produce the best code. Because signal transforms are often black box and run numerous times, there is a large gain for any improvements produced in this manner.

CHAPTER 3

AUTOMATING FINITE ELEMENT METHODS

The automation of finite element methods requires a highly expressive mathematical language and the generation of high performance code; good interfaces allow the separation of these opposing concerns. The list of FEM packages attempting to bridge this gap is staggering, each taking to heart a different set of abstractions to allow more expressive code. For example both deal.II and libmesh provide automated mesh refinement with a considerable number of elements, hermis provides a hp-adaptive code, Sundance provides automatic differentiation for optimization problems, GetDP focuses on coupling physics and geometric problems, and many more open source and commercial libraries exist.

Each of these example are libraries that a user compiles then links to create a simulation. The FEniCSproject has taken a different approach; it uses language for forms and compiles the form to low level code that evaluates the form on a reference domain. This allows the FEniCSproject to separate the concerns of the local and global properties of the FEM assembly algorithm and use the mathematical representations for each. In this chapter we discuss how the separation of concerns is implemented and the use of the Sieve data structure, which describes and manipulates generic topologies in a way that closely parallels domain-decomposition principles

Two major concerns that have been address by the scientific community, not necessarily related to FEM, are meshing the domain and solving the linear system. While there are many different methods that incorporate these two parts of the FEM algorithm, we leave these topics for others in the community to address. For the purposes of testing our assembly routines, we use off-the-shelf solutions from the mesh generators such at Triangle, Gmsh, or TetGenand use a linear solver codes, such as PETSc, Trilinos, or UMFPACK.

3.1   Automation of FEM Assembly

The mathematical formulation of FEM is usually decomposed into local and global operations related only by a change in basis. For efficiency reasons, FEM codes will often mix these operations, often requiring a large amount of handcoding for different cases and different elements, contradictory to the goal of automating the process. To return to the full generality of the mathematical formulation requires raising the level of abstraction.

The complexity the FEM assembly process can be divided into the local element assembly and the global assembly. Roughly the local assembly is concerned with the forming the local element equations and the global assembly is the process of accumulating these equations correctly. Challenges for the local assembly include defining the finite element in a general fashion and efficiently representing the variational forms to be built. The global assembly then must be able to represent a wide range of functions over possibly a number of parallel processors.

The FEniCSproject is divided up much like FEM theory. This separates the local and global concerns allowing both to be compiled and linked together separately. The initial focus of the project was to provide a simple language for expressing forms and compiling them into optimized assembly routines. This was accomplished with the FIATproject producing the Ciarlet finite element, the FFCproject compiling the form for assembling the reference element, and the DOLFINproject implemented the rest of the FEM algorithm.

By decomposing the assembly of finite element functions into global and local operations, codes can provide a simple interface that allows for arbitrary elements and meshes. This orthogonalization is provided for using concepts from algebric topology embodied by the PETSc Sieve library [50].

3.1.1   Local assembly

We refer to the local assembly process as the formulation of $a^e(\cdot, \cdot) = L^e(\cdot)$. The process is split into several parts. The finite element determines the basis functions used to tabulate the local element. The variational form will determine the proper integrals and necessary coefficient functions. The result is a function that takes local data from the mesh and coefficient functions and produces the system of equations determined by the mesh element. This assembly is divided into several pieces: tabulating the finite element, representing the variational forms, and optimizing the computation of the forms.

Tabulating the finite element

A finite element tabulator is a middleware product that has been developed to give codes access to basis descriptions of hard to form finite elements. Tabulating the finite element space is not a new idea, since every FEM code must do it in some way, but by using some more mathematical operations a tabulator can easily make a large number of elements with arbitrary order.

The FEniCSproject has made two tabulators available one based on a numerical code and a symbolic one, FIAT [43] and SyFi [2], respectively. FIAThas been used in several codes and was created to allow the user to give researchers access to the numerous elements that are used in theory but rarely used in practice. It computes general nodal basis functions as linear combinations of orthogonal polynomials. With recurrence relations that allow stable evaluation of these polynomials to very high orders, and rules describing the degrees of freedom for the families of finite elements, FIATis able to create an effective code for tabulating arbitrary order basis functions for nearly arbitrary finite elements.

The SyFipackage predefines many types of polygonal domains, polynomial spaces, and degrees of freedom as symbolic expressions that are easily manipulated. This makes it easy to define and use finite elements. While the FIATpackage gives an implementer the ability to

define elements in the library easily, a user can only pick from a list of elements and orders implemented. SyFialso implements a form compiler that will generate the local element assembly code as well giving users the abilities to use the generated elements in any code that uses the Unified Form-assembly Code (UFC) interface. The symbolics of SyFicomes at a cost, as computing with symbolics can often cause exponential growth in the number of terms evaluated.

Form evaluation

Possibly one of the largest factors in code expressivity and ability is how one can describe the problems to be solved. If it is the case that one can only solve a list of equations, the software immediately becomes useless as soon as a different equation is needed. On the other hand if one uses a symbolic system that will allow for any equation that can be written down, one risks a very slow implementation that will not be able to handle large problems. Additionally, if one has an optimization or control problem, without differentiation of the solutions methods are limited.

Current FEM software completely disagree with one another on how to describe the problem. For example, Sundance allows for virtually any weak form and uses a symbolics engine to evaluate the equations. FFCuses the unified form language UFLwhich takes in a weak form, but has limits in the types of operators available. FFCprecompiles parts of the transformation on the reference element and generates efficient routines for DOLFINto use in assembly. Finally deal.II leaves the user to input the problem as part of the assembly loop, meaning low-level C++ code, thus it becomes the users responsibility to control how the problem is entered into the simulation.

Perhaps the most interesting achievement with the FEniCSproject is the comparison of an optimized tensor representation for the weak form versus quadrature forms. For simple

problems FFCuses a tensor representation

$$A_i^K = \sum_{\alpha \in \mathcal{A}} A_{i\alpha}^0 (G_K)_\alpha, \quad i \in \mathcal{I}_K,$$

or simply

$$A^K = A^0 : G_K, \tag{3.1}$$

where $\mathcal{I}_K$ is the set of admissible multiindices for the element tensor A^K and $\mathcal{A}$ is the set of admissible multiindices for the geometry tensor G_K. The reference tensor A^0 can be computed at compile-time, and may then be contracted with a G_K to obtain the element tensor A^K for each cell K in the finite element mesh at run-time. This allows the precomputation of terms that do not depend on the geometric features of the element in real space, such as the Jacobian or any coefficients. For example A^K of the weighted Laplacian would be:

$$A_{i_1 i_2}^K = \sum_{\alpha_1=1}^d \sum_{\alpha_2=1}^d \sum_{\alpha_3=1}^n \det F_K' w_{\alpha_3} \sum_{\beta=1}^d \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \int_{K_0} \frac{\partial \Phi_{i_1}}{\partial X_{\alpha_1}} \frac{\partial \Phi_{i_2}}{\partial X_{\alpha_2}} dX \tag{3.2}$$

$$G_k = \det F_K' w_{\alpha_3} \sum_{\beta=1}^d \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \tag{3.3}$$

$$A^0 = \int_{K_0} \frac{\partial \Phi_{i_1}}{\partial X_{\alpha_1}} \frac{\partial \Phi_{i_2}}{\partial X_{\alpha_2}} dX \tag{3.4}$$

Using this representation produces up to one thousand times speed ups for simple problems [46]. This representation also presented new opportunities for optimization which became the FErariproject and will be discussed in the next section.

The major problem associated with the tensor representation of the weak form is complex forms usually require numerous coefficient functions. This situation occurs quite often with non-linear or multiphysics problems. But since the code is generated, changing from the tensor representation to a quadrature representation is a simple change in the compiler flags.

18

The example of the weighted Laplacian becomes:

$$A_{i_1 i_2}^K = \sum_{q=1}^{N} \sum_{\alpha_1=1}^{d} \sum_{\alpha_2=1}^{d} \sum_{\alpha_3=1}^{n} \Phi_{\alpha_3}(X^q) w_{\alpha_3} \sum_{\beta=1}^{d} \frac{\partial X_{\alpha_1}}{\partial x_\beta} \frac{\partial X_{\alpha_2}}{\partial x_\beta} \frac{\partial \Phi_{i_1}(X^q)}{\partial X_{\alpha_1}} \frac{\partial \Phi_{i_2}(X^q)}{\partial X_{\alpha_2}} \det F_K' W^q$$

Using a quadrature representation and applying simple loop optimization techniques, such as unrolling and tiling, Ølgaard and Wells showed a series of forms that were sped up thirty times over the tensor representation and require much smaller amounts of generated code [58].

Optimizing form evaluation

The tensor contraction structure for the computation of the element tensor A^K enables additional optimizations for the form compiler. For typical variational forms, the reference tensor A^0 has significant structure that allows the element tensor A^K to be computed on an arbitrary element in a reduced amount of arithmetic. Reducing the number of operations based on this structure leads naturally to several problems in discrete mathematics.

It is convenient to recast (3.1) in terms of a matrix-vector product:

$$A^0 : G_K \leftrightarrow \tilde{A}^0 \tilde{g}_K. \tag{3.5}$$

The matrix $\tilde{A}$ lies in $\mathbb{R}^{|\mathcal{I}_K| \times |\mathcal{A}|}$, and the vector $\tilde{g}_K$ lies in $\mathbb{R}^{|\mathcal{A}|}$. The resulting matrix-vector product can then be reshaped into the element stiffness matrix A^K. As this computation must occur for each cell K in a finite element mesh, it makes sense to try to make this operation efficient. Note that a corresponding construction is possible also for lower and higher rank element tensors by an appropriate ordering (flattening) of the entries of the element tensor.

Consider the problem of computing

$$y = Ax$$

19

efficiently, where $A = \tilde{A}^0$ is known *a priori* and $x = g_K$ is not. As studied in the FEr-ariproject [44, 45, 48, 74], the structure of A enables a reduction in the amount of arithmetic required to form these products. This structure is discovered through a series of algorithms to find edit distance, zeros, linearity, and higher geometric relations. Take the following example:

$$
A^0 = \left(\begin{array}{cc|cc|cc|cc|cc|cc}
3 & 0 & 0 & -1 & 1 & 1 & -4 & -4 & 0 & 4 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 3 & 1 & 1 & 0 & 0 & 4 & 0 & -4 & -4 \\
1 & 0 & 0 & 1 & 3 & 3 & -4 & 0 & 0 & 0 & 0 & -4 \\
1 & 0 & 0 & 1 & 3 & 3 & -4 & 0 & 0 & 0 & 0 & -4 \\
\hline
-4 & 0 & 0 & 0 & -4 & -4 & 8 & 4 & 0 & -4 & 0 & 4 \\
-4 & 0 & 0 & 0 & 0 & 0 & 4 & 8 & -4 & -8 & 4 & 0 \\
\hline
0 & 0 & 0 & 4 & 0 & 0 & 0 & -4 & 8 & 4 & -8 & -4 \\
4 & 0 & 0 & 0 & 0 & 0 & -4 & -8 & 4 & 8 & -4 & 0 \\
\hline
0 & 0 & 0 & -4 & 0 & 0 & 0 & 4 & -8 & -4 & 8 & 4 \\
0 & 0 & 0 & -4 & -4 & -4 & 4 & 0 & -4 & 0 & 4 & 8
\end{array}\right)
\tag{3.6}
$$

It is possible to apply A to an arbitrary x in fewer operations than a standard matrix-vector multiplication which requires 144 multiply-add pairs. This requires offline analysis of A and special-purpose code generation that applies the particular A to a generic x. For examination of A consider the small subset in (3.7) which would only cost 48 multiply-add pairs but contains all the relations we use to optimize the larger version.
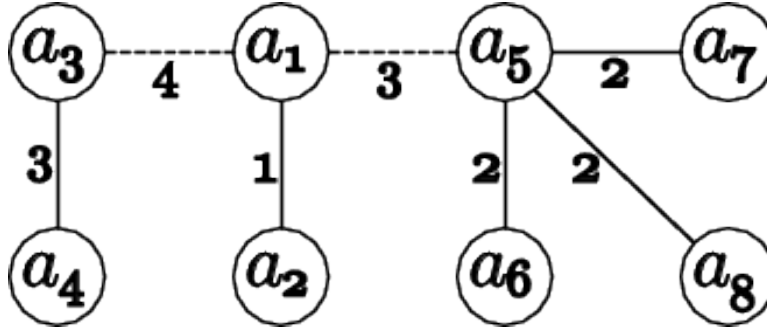
Figure 3.1: Minimum spanning forest for the vectors in Equation 3.7. The dashed edges represent edges that do not reduce the number of operations and thus disconnect the graph.

$$
A = \begin{pmatrix}
a^1 : A^0_{1,3} \\
a^2 : A^0_{1,4} \\
a^3 : A^0_{2,3} \\
a^4 : A^0_{3,3} \\
a^5 : A^0_{4,6} \\
a^6 : A^0_{4,4} \\
a^7 : A^0_{4,5} \\
a^8 : A^0_{5,6} \\
a^9 : A^0_{6,1} \\
a^{10} : A^0_{6,6}
\end{pmatrix}
=
\begin{pmatrix}
1 & 1 & 0 & 0 \\
-4 & -4 & 0 & 0 \\
0 & 0 & 1 & 1 \\
3 & 3 & 3 & 3 \\
0 & 4 & 4 & 0 \\
8 & 4 & 4 & 8 \\
0 & -4 & -4 & -8 \\
-8 & -4 & -4 & 0 \\
0 & 0 & 0 & 0 \\
8 & 4 & 4 & 8
\end{pmatrix}
\tag{3.7}
$$

Using the structure there are several ways to optimize the form evaluation. The first is to put the binary relations in a graph problem. Two nodes are connected by cost of computing one from the other. An optimal path can be found using Dijkstra's algorithm. See the example in Figure fig:kirby4:mst. Higher order relations have been used through a hypergraph model with the edges sets of relations and a greedy algorithm decides which to vectors to compute. Another formulation uses a generating graph where a node is connected if it can be generated by another node.

Empirical results for simple models show reductions of up to four times just using a simple tensor contraction routine. The time to search for the relations and build the graphs is often much larger than that of applying the simple tensor contraction; thus these methods are only useful for operators that are to be applied numerous times. Additionally because they rely heavily on data reuse, they are a promising direction for memory limited architectures.

3.1.2   Global assembly

By automating the local assembly, numerous elements can be represented and optimized accordingly. Putting the local stiffness matrix into the global stiffness matrix is often overlooked in theoretical treatments of FEM, but this process is often not general enough to handle the large number of elements theory provides. Even more difficult is supporting the numerous elements on high performance data structures that scale well. Here we describe the challenges of the global assembly process and in the next section we present the usage of the Sieve parallel data structure to access low level high performance data structures.

The first challenge is separating the local element definitions from the mesh definition. For example, many FEM codes assume triangles or tetrahedrons for two dimensions and are unable to use other geometries often used in applications such as quadrangles or hexagons. Codes also often neglect to include ideas of orientation in the elements and construction of the global interpolant is restricted to elements that do not require any directionally dependent features.

Another issue is mapping the nodes to the mesh in a scalable way. The earliest codes [41] use an associative array that explicitly mapped mesh node to matrix equation; called the degree of freedom map. This data structure becomes a large bottleneck for parallelization of the code. It is required to be updated for any adaptive procedures and distributed before the global assembly takes place. Some approaches to mitigate this problem require not a explicit map but a function to generate the relations. This requires the mesh to be have

some structure and can increase the number of required elements by exponential factors to get the same mesh grading. Our approach is to rethink the mesh to include this data in the connectivity of the elements. This method allows for a seamless integration of the dof map requiring only updates on the appropriate overlap of the elements.

After all the data and mappings are mitigated, the global assembly becomes an accumulation loop inputting each local contribution to the global operator. The advantage of the automation of the process is the flexibility to assembly in numerous ways without changing the data structures or the local elements. In Chapter 7 we show a simple modification to the assembly process to produce polynomials for homotopy solvers. Another commonly used option is to not store the operator but only assemble and apply it when necessary.

3.2   Sieve

We have discussed many advances to the automation of FEM through a global and local decomposition of the code. Codes implementing FEM often disregard this decomposition in the interest of performance. Here we present the use of a parallel domain decomposition data structure, a Sieve, that naturally allows the proposed FEM structure.

A Sieve is a model for describing and manipulating generic topologies in a way that closely parallels domain-decomposition principles. The basic idea is to break up a function into pieces over a collection of subdomains, and then assemble a global function over the entire domain using information from the overlap of the subdomains. The PETSc Sieve library [4, 51] provides an implementation of these concepts.

A *sieve* captures the hierarchy inherent in topological constructions with a single covering relation. It consists of *points*, and *arrows* connecting a point to any point which it covers. For example a point could represent any piece of a mesh, such as a vertex, edge, or face, and a triangular cell would consist of a face point with three arrows to edge points with two arrows each to vertex points, see Fig. 3.2. For some typical operations for selecting points

Table 3.1: Some Typical Operations on a sieve

cone(p)	the sequence of points which cover a given point p
closure(p)	transitive closure of cone
support(p)	the sequence of points which are covered by a given point p
star(p)	transitive closure of support
meet(p,q)	minimal separator of closure(p) and closure(q)
join(p,q)	minimal separator of star(p) and star(q)
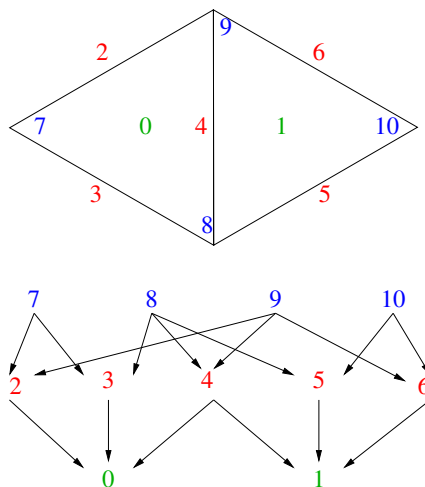
Figure 3.2: Sieve representation of a small triangular mesh

on a sieve based on covering relations, see Table 3.1.

An *overlap* is introduced to link several sieves when representing a complex manifold. Since the overlap is another covering relation it can be implemented as another type of sieve. Thus the sieve construction can express hierarchy at several levels, wherever it arises. A common application of the overlap is in a parallel setting to identify points in sieves on different processors.

A *section* is a mapping from sieve points to a vector of values. This is the analgoue of the vector bundle construction from mathematics, which associates a vector space with each point of an underlying base manifold. The section may map different spaces or "fibers" to different points. The principle interface to values in a section is the `restrict()` operation that returns all the values on a given subdomain. The dual operation is `update()`, injecting

subdomain values into the global section. The *completion* operation is used to enforce coherence between cones (or supports) of points identified through an overlap. In our section implementation, we use contiguous storage for the values. In order to record fiber dimensions and offsets into this array, we use another section which we call an *atlas*.

This framework can describe almost any topology, however we concentrate on cell complexes which are suitable for representing FEM meshes. For more complete background and terminology, see [50].

3.2.1   Reference Finite Element Method

An operation necessary for some elements is a mapping of the local basis on the reference element into a basis for the subdomain which is compatible with the global basis for the entire domain. For example, the derivative degrees of freedom for Hermite elements transform as J^{-T} where J is the element Jacobian. The Raviart-Thomas element has degrees of freedom which are normal derivatives on the mesh edges. The reference basis on an element always uses the outward normals to the element edges, however the global basis must pick one of the two outward normals from neighboring triangles. This requires that each sieve arrow be oriented, but this orientation can still be calculated locally.

Our main advantage is the formulation of a clear divide between global, topological operations and local, analytical operations, mediated through the sieve interface for hierarchy management and its dual for functions over such a hierarchy. This allows us to orthogonalize the FEM assembly and completion routines from the local integration routines. We easily treat the mesh in parallel, as well as the underlying linear algebra. Moreover, we use the FIATsystem [43] to specify elements in a declarative fashion, allowing easy and intuitive definition of elements by the user, whereas imperative code for elements is extremely hard to understand or maintain.

3.2.2   Functions over a Mesh

FEM define functions in $\mathcal{P}$ on the local element and links them together over the entire mesh, creating a global interpolant, using an assembly process. Locally, a function is defined by its coefficients in the basis $\mathcal{P}$. Since each basis vector maps by duality to a given dual basis vector, and each dual basis vector is assoicated with a sieve point, we can define a section over the element in which each basis coefficient covers the point associated to the appropriate dual vector. We may take a collection of points and choose a unique dual basis over them, which allows us to transform local coefficients into coefficients in this extended basis. Therefore, we can extend our section over a subdomain, or group of points larger than a cell.

Globally we may relate subdomains using an overlap, which identifies points in the two sieves. If we have chosen the same dual basis vector for both related points, the section values can be identified as well. More generally, we can introduce a transformation which will carry a coefficient from one section to the other. Notice here that since the subdomains are arbitrary, we could have introduced a section for each cell, meaning the overlap would consist of all interior faces, and the overlap transformation would be precisely that one between the local and global bases. This is the totally unassembled view of the given function.

Sieves encode topology and sections functions supported on the topology, which make sections dual objects to sieves. Sieves define local covering relations and are then patched together using an overlap, which relates points in two sieves. The analogous clutching construction for two sections defined over different sieves is the completion operation. An obvious example of such an operation is the transformation relating functions on two overlapping coordinate patches of a manifold.

Globally, we must relate sections defined on different sieves, which we accomplish using the *completion* mechanism [49]. We may split completion into two operations, *communication* and *fusion*. The communication phase restricts each section to the overlap, and then

communicates this overlap section to the other processes. The fusion phase then operates on the set of overlap sections to produce a single section replacing the original values. For example, in finite element accumulation, the fusion operation is merely addition.

Clearly, fusion operations with special properties permit certain optimizations. Accumulation allows received values to be added directly to existing values, preventing some copies. Nothing in the interface precludes these optimizations, and in fact the current templated interface enables some interesting algebraic optimizations based upon C++ concepts [34]. In that sense, we are not sacrificing efficiency for generality.

3.2.3   Local Function Definition

We begin to define a section by specifying the dimension of the fiber space attached to each point in the sieve. This task is simplified when using a reference element, due to symmetry requirements, to the specification of a fiber dimension for each kind of topological element, e.g. vertex, face, cell. FIAT provides this information for the given element.

Next, we classify degrees of freedom of the element by the way they transform when mapped to the reference element. Scalar degrees of freedom, such as those in Lagrange elements, remain invariant under the transformation, as we expect. Vector degrees of freedom, such as derivatives in Hermite elements or gradients in the Laplacian weak form, transform using J^{-T}. Integral moments transform as the Jacobian determinant. FIAT now provides a classifier for each degree of freedom so that an appropriate transformation may be automatically constructed.

A further difficulty arises from variables defined as outward normal vectors or normal derivatives. In the global basis we must choose one orientation, and this some element must reverse the sign of their contribution. However, this can be handled using the induced orientation of each sieve point. The orientation of the arrow from the cell to the face with the normal determines the sign.

3.2.4 Global update

A section may be restricted to any given cell to obtain the local function representation, and updated from a modified local representation. The restriction and update operations rely on an ordering for the closure of the element, provided by the underlying sieve. This allows us to use purely local names for all sieve points, since an overlap allows identified points to have different names in two different local sieves. Therefore, a sieve mesh may scale to any size given a fixed local memory and overlap. The sieve may also need to supply an orientation for each arrow, but this is also scalable.

3.2.5 Implementation Issues

Many practical problems arise when implementing any large scale code. FEM with Sieve is no different and has required specific solutions to many general problems. For example the orientation of an element is in coded by the order of the points returned by cone of the cell. Elements are transformed by the selection of the correct rules or coded by the user using geometric data. Any geometry is allowed, as long as a mesh can be supplied, but the user is still required to give methods for computing geometric data such as the Jacobian of a cell. These specific problems may handle small user added code to run but allow the user to make use of the broader system without much thought.

Perhaps more interesting implementation details are the handling of section allocation, integration rules, and boundary conditions. These details must be handled generally and thus require more planning.

Section Allocation

Sections map sieve points to values, but have no other information about the differential structure. Thus, for section allocation we need only know how many values are associated

with each sieve point. From the element description, we know how many degrees of freedom are associated with points of a given depth in the reference cell. By symmetry, we extend this description to the entire mesh. If we have several fields, we simply sum the degrees of freedom for each depth. The sizes for each field are also recorded separately so that we may later split, or *fibrate*, the section into its component parts.

It is common to define a field only on a subset of the given domain. For instance, a coupled neutronics and gas flow simulation for a nuclear reactor core would define the neutron flux over the entire domain, but the gas momentum, pressure, and energy only over the open, non-solid, portions. We can indicate the set of excluded sieve points, and procede with the procedure above. In the implementation, we use a label whose base is the excluded set.

Constraints are represented just as another field, having support on a set of sieve points, with associated values being the set of constrained indices on each point. In practice, we constrain a set of fields on each point, and thus keep the total constraint size and constraint size for each field on a point. Constrained values are stored in the normal value storage. This allows contiguous access to both unconstrained and constrained unknowns through the `restrict()` operation, exactly as we expect. The `update()` operation now only changes unconstrained unknowns. In addition, an `updateBC()` and `updateAll()` are provided to allow update of the constrained variables. The entire operation of section allocation occurs in the `Mesh::setupField()` method.

Integration Rules

In order to assemble a weak form, we integrate over each cell in turn. The first step in integration is to compute the cell geometry, in particular an affine map to the reference cell, as our quadrature rules are defined there. We retrieve from the `Discretization` objects, the basis function and derivative, or jet, tabulation. From this, we can construct the value of the weak form at any quadrature point, which are accumulated to produce the integral.

An entry exists in the element vector for each test function space, which in turn matched each basis function space, or each `Discretization` object. For the element matrix, we take the tensor product. The `Discretization` store indices into the element vector order for each degree of freedom of that element. The `Mesh` determines this order by consulting all discretizations in the `calculateIndices()` method.

The integration rules are produced by FIAT. The `quadrature.fiat` file contains: an element (usually a family and degree) defined by FIAT and a quadrature rule. It is then run automatically by **make**, or independently by the user. It can take arguments –**element_family** and –**element_order**, or **make** takes variables **ELEMENT** and **ORDER**. Then `make` produces `quadrature.h` with:

- Quadrature points and weights

- Basis function and derivative evaluations at the quadrature points

- Integration against dual basis functions over the cell

- Local dofs for Section allocation

Boundary Conditions

Boundary conditions are a difficult part of any scientific code, sometimes the most difficult part. There are two basic boundary conditions with FEM: Neumann and Dirichlet. We advocate considering the boundary conditions during the assembly process rather than the usual post-processing that is done in many codes.

Neumann conditions may be expressed as

$$\nabla u \cdot \hat{n}|_\Gamma = h$$

and implemented by explicit integration along the boundary. The user only needs to provide

a weak form. The code pro

Dirichlet conditions may be expressed as

$$u|_\Gamma = g$$

and implemented by constraints on dofs in a Section. The user provides a function and the code is able to eliminate the constraints during assembly.

3.2.6 Examples

The provided example, Algorithm 3.1, gives a generic version of FEM generated with Sieve code. The mesh is already given by the Sieve code and FIAThas generated the quadrature rules for integration. The provided code assumes scalar affine nodes but can be adapted by changing the transform and adding a loop for each vector field. The user is able to simple supply with the appropriate boundary functions, right hand side, linear form, and residual calculations. While the code may seem very short, it is through the powerful local/global decomposition that it is able to provide such expressiveness.

Algorithm 3.1 General sieve code implementing FEM

```
cells = mesh->heightStratum(0);
for(c = cells->begin(); c != cells->end(); ++c) \{
  // Compute cell geometry
  coords = mesh->restrict(coordinates, c);
  v0, J, invJ, detJ = computeGeometry(coords);
  // Retrieve values from input vector
  inputVec = mesh->restrict(U, c);
  for (q = 0; q < numQuadPoints; ++q) \{
    // Transform coordinates
    realCoords = J*refCoords[q] + v0;
    for(f = 0; f < numBasisFuncs; ++f) \{
      // Constant term
      elemVec[f] += basis[q,f]*rhsFunc(realCoords);
      // Linear Term
      for(d = 0; d < dim; ++d) \{
        for(g = 0; g < numBasisFuncs; ++g) \{
          for(d = 0; d < dim; ++d) \{
            // Compute a(u, v) and store in approriate elemVec
            elemMat[f,g] += /* user defined */
            elemVec[f] += elemMat[f,g]*inputVec[g];
          \}
        \}
      \}
      // Add Nonlinear residual terms to elemVec
      elemVec[f] +=/* user defined */
    // Multiply by quadrature point weight and scale to reference
    elemVec[f] *= weight[q]*detJ;
    \}
  \}
  //Update output vector
  mesh->updateAdd(F, c, elemVec);
\}
// Aggregate updates
Distribution<Mesh>::completeSection(mesh, F);
```

CHAPTER 4

RHEOLOGY APPLICATION ENGINE: NEWTONIAN FLUIDS

Despite the numerous FEM libraries, computational fluid dynamics still remains an experts–only field. Furthermore, when a complex model of a fluid is required, there is not a consensus on the best mathematical models. Even the simplest non-Newtonian fluids have many numerical challenges for creating robust simulations. The Rheology Application Engine, or Rheagen, is a software library that bridges the gaps between proposing mathematical models, applying stable numerical techniques, and giving an intuitive non-expert interface for testing different models.

Rheagen uses the FEniCS automation tools to generate and manage many different simulations. A researcher defines the weak form, then chooses between numerous element types and stabilization methods in the high level UFL language. From this definition of the form, FFC generates low-level C++ code for assembling the form on a reference element. Rheagen uses the DOLFIN library to assemble the system and manages solving the system of equations. A user is able to define a geometry and boundary conditions using DOLFIN and pass this definition to the fluid model.

This design creates three distinct boundaries of separation: the definition of a mathematical model, the discretizing and solving of an FEM system, and the application of a fluid model to a specific geometry. Through this separation we are able to implement many models, apply simulation techniques uniformly on the different models, and provide a large sample of test cases. The code is open source and has been used by several users; one user has even contributed a power law Stokes fluid model.

The current focus of Rheagen is mixed method FEM for planar flows at low Reynolds number. Even with this somewhat stringent requirement on the fluid, there are numerous complex models that have yet to be fully compared with one another on all but a few test problems. To give a fuller description of the Rheagen library, we first discuss the basic

equations of FEM fluid modeling with the Newtonian model. In the two subsequent chapters, we discuss the application of other methods to solve more difficult models.

4.1   The Newtonian fluid model

Incompressible fluid dynamics uses three basic equations: the conservation of momentum, the conservation of mass, and a constitutive equation. For each fluid model presented in this work will use the same equations for the two conservation laws,

$$\nabla \cdot \boldsymbol{u} = 0 \tag{4.1}$$

$$\rho \frac{\mathrm{D}\boldsymbol{u}}{\mathrm{D}t} \equiv \rho \left(\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} \right) = -\nabla p + \nabla \cdot \boldsymbol{T} + \boldsymbol{f} \tag{4.2}$$

where ρ denotes the constant density, D/Dt the material time derivatives, p the pressure, $\boldsymbol{T}$ the extra stress and $\boldsymbol{f}$ a general volumetric force. Each different models uses a different constitutive equation relating the stress with the rate of strain tensor or fluid velocity.

The first model to analyze is Newtonian fluids where the constitutive equation is a linear relation.

$$\boldsymbol{T} = 2\eta \boldsymbol{D} \qquad \text{where } \boldsymbol{D} = \frac{1}{2}\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T \tag{4.3}$$

$\boldsymbol{D}$ is the rate of strain tensor and η the viscosity, sometimes referred to as the solvent viscosity. Each non-Newtonian fluid model adds non-linear terms to this constitutive equation, and for numerical solutions man uses the linear equation as the starting solution. This is because in the limit of small viscosity they must all reduce to this model. To understanding the complexities of non-Newtonian models, we start with some background on solving the Newtonian model.

The above three equations above are usually known as the Navier-Stokes equations. There is a rich history of computing the solution to these equations, but we focus on two distinct

aspects which will affect the solution of each non-Newtonian fluid model. Discretizing the linear model, which corresponds to steady-state Stokes flow, the limit of zero Reynolds number ($Re \sim \rho|\boldsymbol{u}|/\eta$) commonly referred to as creeping flow. The constitutive equation and conservation of momentum reduce to a relation between the velocity and pressure:

$$-\Delta \boldsymbol{u} + \nabla p = \boldsymbol{f} \tag{4.4}$$

$$\nabla \cdot \boldsymbol{u} = 0. \tag{4.5}$$

4.1.1 Numerical Formulation: Discretization

Discretizing the Stokes equations has been well studied and is documented in several books [13, 14]. It is a challenging problem for a few reasons. First, due to the coupling between the velocity and pressure results in a saddle point in the mixed variational form. The matrix equations will be indefinite which proves to be quite taxing on iterative linear solvers. Second, the divergence free requirement on the velocity requires special care for the numerical discretization to preserve this property; it is often violated in common cases.

One natural way to solve the system is to create a mixed system, with blocks corresponding to fields, and each field using a different element. The variational form of this mixed system is as follows:

Let $V = H^1(\Omega)^n$ and $\Pi = \{q \in L^2(\Omega) : \int_\Omega q dx = 0\}$. Given $F \in V$, find functions $\mathbf{u} \in V$ and $p \in \Pi$ such that

$$\begin{aligned} a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) &= F(\mathbf{v}) \quad \forall v \in V \\ b(\mathbf{u}, q) &= 0 \quad \forall q \in \Pi, \end{aligned} \tag{4.6}$$

35

(a) P_3 for V (b) P_2 for Π
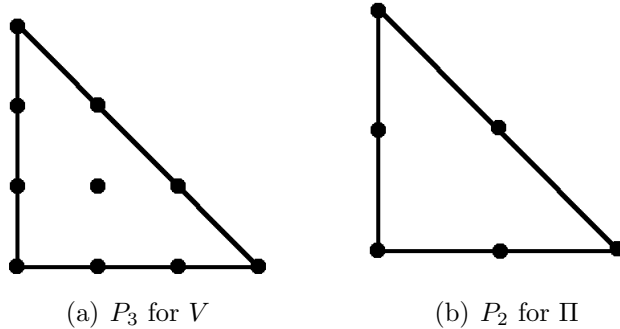
Figure 4.1: Taylor-Hood Elements

where

$$
\begin{aligned}
a(\mathbf{u}, \mathbf{v}) &:= \int_\Omega \nabla \mathbf{u} \cdot \nabla \mathbf{v} \, dx, \\
b(\mathbf{v}, q) &:= \int_\Omega (\nabla \cdot \mathbf{v}) q \, dx.
\end{aligned}
$$

This mixed method formulation uses two discrete spaces V and Π. Developing different finite element spaces for this system is quite challenging due to the Ladyzhenskaya-Babuška-Brezzi compatibility condition which is necessary for stability, see Brezzi-Fortin for more details [14]. Using the most straight forward schemes, such as equal order continuous element spaces for both the pressure and velocity, leads to an over-determined system of equations. Thus a range of schemes, each varying in its ability to represent the solution space accurately, have been developed. Here we presents some well known elements and evaluate the numerical consequences.

The Taylor-Hood element [12, 69] is one of the most widely used elements for solving Stokes flow. It consists of a P_k element for the velocity space and P_{k-1} for the pressure space (see Figure 4.1). Because of the simplicity of using Lagrangian elements, it can easily be extended to higher orders. This element produces a continuous pressure space, but the order of the pressure convergence is lower than that for the velocity.

The Crouzeix-Raviart Element [23] is a non-conforming element that is it does not con-

(a)   Crouzeix-Raviart
for V

(b) P_0 for Π
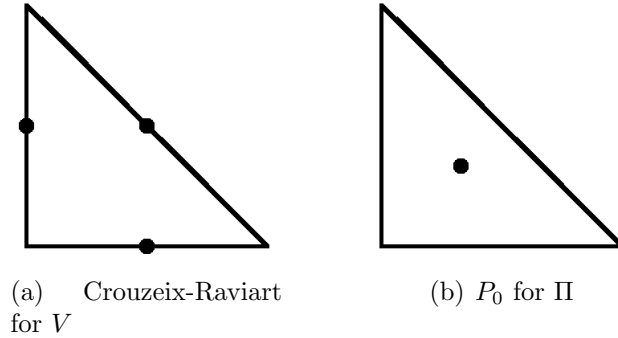
Figure 4.2: Crouzeix-Raviart Elements

verge to a proper subspace of the approximating space. It uses integral moments over the element edges as a basis for the velocity and a discontinuous pressure space, P_0 (see Figure 4.2). For the low order case, the velocity edge moments are equivalent to evaluating the basis functions at the center of each edge.

Another possibility is just to use the high degree Lagrange element for velocity but use a discontinuous element two orders lower in the pressure space, what we loosely call CD. Brezzi and Fortin discuss the $P_2 - P_0$ case [14], but for completeness we include higher order versions as well. For the low orders, the pressure is poorly approximated and the element loses a degree of convergence. This space does not satisfy the LBB condition but is commonly used with a stabilization parameter or enriched with bubble functions which are not discussed here.

To get around the difficulties of the LBB condition, there has been work on stabilization of the discrete problem. Stabilization attempts to convert the discrete problem from a saddle point to a positive definite matrix, for a more complete discussion see Donea and Huerta [25]. The most successful method has been the streamline upwind/Petrov-Galerkin (SUPG) method of Brookes and Hughes [15]. For our Stokes problem test, using only the pressure stabilization from SUPG, the variational problem becomes:

$$a(\mathbf{u}, \mathbf{v}) + b(\mathbf{v}, p) + (\delta \cdot \nabla q, \nabla p) = (\mathbf{v}, \mathbf{f}) + (\delta \cdot \nabla q, \mathbf{f})$$

$$b(\mathbf{u}, q) = 0$$

(4.7)

where δ is a stabilization parameter. For our tests, it was set to 0.2 times the circumference of the mesh cell squared. The tests stabilize the simple element that uses continuous elements of the same order for both the pressure and velocity spaces, which we call STAB. Additionally we apply the stabilization parameter the Taylor-Hood element, TH_STAB, and get a improved convergence rate.

Other strategies to avoid the LBB condition and the saddle point problem are the Uzawa iteration method and penalty methods. A combination of these two ingredients results in the iterated penalty method, or Scott and Vogelius [64]. Let $r, \rho \in \mathbb{R}$ and $\rho > 0$ and define $\mathbf{u}^n$ and $\mathbf{w}^n$ by

$$a(\mathbf{u}^n, \mathbf{v}) + r(\nabla \cdot \mathbf{u^n}, \nabla \cdot \mathbf{v}) = F(\mathbf{v}) - (\nabla \cdot \mathbf{v}, \nabla \cdot \mathbf{w^n}) \tag{4.8}$$

$$\mathbf{w}^{n+1} = \mathbf{w}^n + \rho \mathbf{u}^n \tag{4.9}$$

The pressure may be recovered from the auxiliary $\mathbf{w}$ field, $p = \nabla \cdot \mathbf{w} - \mathbf{C}$ were C is a constant, we use the average of $\nabla \cdot \mathbf{w}$ to center around zero. The algorithm assumes $\mathbf{w^0} = \mathbf{0}$, solves the first equation updating $\mathbf{w}$ every step it finds a fixed point, $||u^{n+1} - u^n|| < \epsilon$. This method uses only one space, but requires a higher order continuous element [65] and it solves the divergence free criteria exactly. The iteration count and accuracy is dependent upon the penalty coefficients ρ and r. For our experiments we use $\rho = -r = 1.0e3$.

4.2 Implementation

In order to evaluate these methods, we will compare mesh sizes and element orders on a test problem with an analytic solution and also compare the lid driven cavity solutions as well.
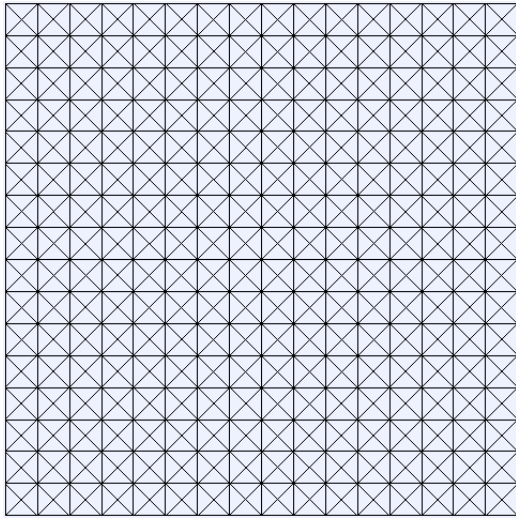
(a) Unit square mesh 16x16 with cross pattern
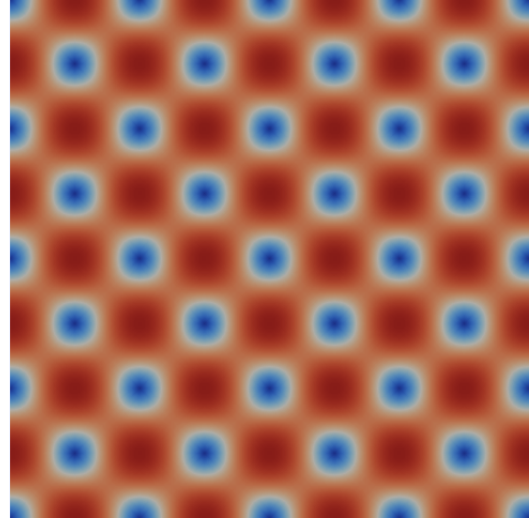
(b) Velocity magnitude of test problem
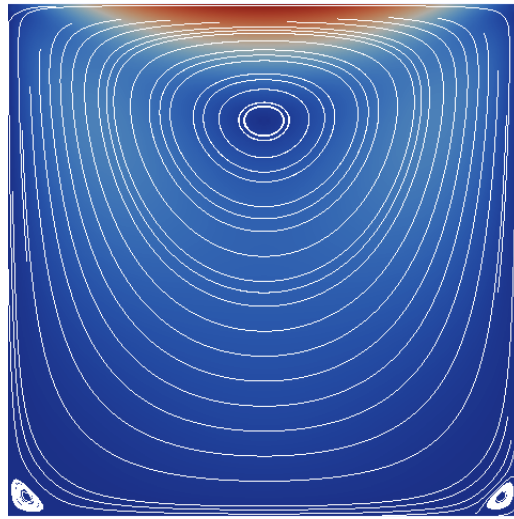
(c) Velocity magnitude of lid driven cavity with streamlines

Figure 4.3: Test mesh and solution plots

The simulations are implemented in Rheagenusing the LU solver from the SuiteSparsepackage, UMFPACK [24]. Because we are using direct linear algegra solver, the number of degrees of freedom is a good measure of the total work the method, this is not true for iterative solvers for such studies we recommend the book by Elman, Silvester and Wathen[29]. The number of the degrees of freedom as a function of mesh size and element is shown in Table 4.2.

Table 4.1: Element variables defining the different mixed methods

	Crouzeix-Raviart	CD	Taylor-Hood	STAB	TH_STAB
V_element	"Crouzeix-Raviart"	"CG"	"CG"	"CG"	"CG"
V_order	1	p	p	p	p
P_element	"DG"	"DG"	"CG"	"CG"	"CG"
P_order	0	$p-2$	$p-1$	p	$p-1$
stabilized	False	False	False	True	True

4.2.1 Simulation Setup

For our tests, we use a $N \times N$ unit square mesh with a cross pattern, see Figure 4.3 and use the following analytic solution:

$$\mathbf{f} = \begin{bmatrix} 28\pi^2 \sin(4\pi x)\cos(4\pi y) \\ -36\pi^2 \cos(4\pi x)\sin(4\pi y) \end{bmatrix}, \tag{4.10}$$

$$\mathbf{u} = \begin{bmatrix} \sin(4\pi x)\cos(4\pi y) \\ -\cos(4\pi x)\sin(4\pi y) \end{bmatrix}, \text{ and} \tag{4.11}$$

$$p = \pi * \cos(4\pi x)\cos(4\pi y) \tag{4.12}$$

To aid the linear solver we additionally "pinpoint" one pressure degree of freedom to zero. Because the pressure term is only determined up to a constant, this process sets that constant and insures the space is not singular. See Figures A.1 and A.2 for the definition of this domain in the Python interface. The extension form the test problem to the lid driven cavity is only a change in the boundary conditions functions and right hand side equation.

Given a domain, we can define the variational problem over it, see Figure A.4. For the mixed methods, we define each element by element family and order. Additionally when defining the form we specify the stabilization terms. Rheagenwill use FFCand FIATto generate the correct elements and forms automatically, allowing for one function to test all the methods. See Table 4.1 for a reference to the names of each pair. The iterated penalty method use can be defined generally over the various orders but because the function space

Table 4.2: A comparison of the degrees of freedom for each element organized by velocity order (p) and number of mesh divisions per dimension (m). A '-' indicate that the order for that particular element is not stable or undefined.

p	n	CR	STAB	CD	TH	IP
1	8	1056	435	-	-	-
	16	4160	1635	-	-	-
	32	16512	6339	-	-	-
2	8	-	1635	1346	1235	-
	16	-	6339	5250	4771	-
	32	-	24963	20738	18755	-
3	8	-	3603	3170	2947	-
	16	-	14115	12482	11523	-
	32	-	55875	49538	45571	-
4	8	-	6339	5762	5427	4226
	16	-	24963	22786	21347	16642
	32	-	99075	90626	84675	66050
5	8	-	9843	5762	8675	6562
	16	-	38883	36162	34243	25922
	32	-	154563	144002	136067	103042

and solver is significantly different we show the code in Figure A.5.

To determine the error for the analytic problems, we compile a higher order function space and project the solutions to that space. Then we are able to integrate over the domain, see Figure A.6.

Table 4.3: Convergence rates of the velocity L^2 error for the different elements

p	CR	STAB	CD	TH	TH_STAB	IP
1	$2.01 \pm 1\text{E-}2$	$1.93 \pm 6\text{E-}2$	-	-	-	-
2	-	$3.02 \pm 2\text{E-}2$	$2.15 \pm 1\text{E-}1$	$3.02 \pm 2\text{E-}2$	$3.06 \pm 2\text{E-}2$	-
3	-	$4.00 \pm 1\text{E-}2$	$3.11 \pm 2\text{E-}2$	$3.98 \pm 1\text{E-}2$	$4.00 \pm 4\text{E-}4$	-
4	-	$4.99 \pm 4\text{E-}3$	$4.07 \pm 1\text{E-}2$	$4.99 \pm 1\text{E-}3$	$4.99 \pm 2\text{E-}3$	$5.00 \pm 2\text{E-}3$
5	-	$5.98 \pm 1\text{E-}2$	$5.1 \pm 5\text{E-}2$	$5.97 \pm 1\text{E-}1$	$5.97 \pm 1\text{E-}2$	$5.97 \pm 1\text{E-}1$

4.2.2 Results

In evaluating the method performance, a few features were notable. Table 4.3 displays the calculated order of convergence for each method and order, as calculated from a series of refined meshes with n in $\{8, 16, 32, 64\}$. The optimal error should be $p + 1$ in the L^2 norm, notice the CD element loses one order of convergence due to poor pressure estimation. The difference in mass balance between the divergence free elements and the continuous elements is clearly demonstrated in Figure A.8(a). Runtimes for the mixed elements parallel the number of degrees of freedom; however, the iterated penalty method has better properties for iterative solvers making its increased runtime for the small problems tested less important. All runtimes were computed on a 2.6 GHz Intel Xeon with the timing the assembly and solve in the Python code, assuming all code generation to be compile time.

CHAPTER 5

GRADE TWO FLUIDS

The next fluid model we implement is the grade two or second order fluid, formulated by Rivlin and Ericksen [63]. The extra stress is formulated by a recurrence relation based on the gradient of the velocity.

$$T = \eta \boldsymbol{A}_1 + \alpha_1 \boldsymbol{A}_2 + \alpha_2 \boldsymbol{A}_1 \tag{5.1}$$

$$A_1 = \nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T \tag{5.2}$$

$$A_j = \frac{\mathrm{D}\boldsymbol{A}_{j-1}}{\mathrm{Dt}} + \boldsymbol{A}_{j-1}\nabla \boldsymbol{u} + \nabla \boldsymbol{u}^T \boldsymbol{A}_{j-1} \tag{5.3}$$

This model does add a nonlinear term, but since the stress is only based on the velocity it requires less degrees of freedom than more complex constitutive relations. A typical assumption, based on thermodynamical arguments, is $\alpha_1 + \alpha_2 = 0$, $\alpha_1 > 0$, and $\eta > 0$.

The model has been the subject of many studies [26, 27, 54], often referred to as the second order fluid. The Tanner-Giesekus Theorem [35, 68] has showed this model produces equivalent velocity fields as the Stokes equations in the realm of creeping planar flows, which our study focuses on. Coleman and Noll modified the model for simple fluids which is suitable for slow flow and slightly elastic [20]. Additionally the problem has been applied to areas such as ice flows [54] and turbulence models [16, 17].

5.1 Numerical Formulations

The constitutive equations have numerous formulations but the third order constitutive equations makes it difficult to discretize with standard FEM. We focus on the planar formulation

of Ouazar [19] which Scott and Girault [36, 37] prove to be unique and stable.

$$-\eta\Delta\boldsymbol{u} + \nabla p + \mathbf{z} \times \boldsymbol{u} = \boldsymbol{f} \tag{5.4}$$

$$\nabla \cdot \boldsymbol{u} = 0 \text{ in } \Omega \tag{5.5}$$

$$\eta\mathbf{z} + \alpha\boldsymbol{u} \cdot \nabla\mathbf{z} = \alpha\mathbf{curl}(\boldsymbol{f}) + \eta\mathbf{curl}(\boldsymbol{u}) \tag{5.6}$$

$$\boldsymbol{u} \cdot \boldsymbol{n} = \mathbf{0} \text{ on } \partial\Omega \tag{5.7}$$

with $\alpha = \alpha_1$ and substituting $\alpha_2 = -\alpha_1$.

The solver strategy is to iterate between the mass and momentum equations, Equations 5.4 and 5.5), and the transport equation, Equation 5.6. Only the boundary conditions, Equation 5.7, allowed Scott and Girault to prove existence and uniqueness; nevertheless, we implement the system without strictly adhering to this condition.

The hyperbolicity of the transport equation requires special treatment to solve the system. Scott and Girault also propose five different numerical schemes for discretizing the transport equations. Amara et al use a regularization term. We use a scheme based on Brooks and Hughes SUPG method [15], which is a streamline diffusion method from Scott and Girault with the parameters determined by the local velocity. This large number of numerical schemes is what attracted this problem to us as a nice candidate for using automation methods to compare.

For comparisons of the different schemes we implemented five different formulations of the transport equation: standard Galerkin, symmetrized , streamline diffusion, streamline symmetric, and regularized. Each of these schemes can be implemented with augmentations of the variational forms under our automation model of FEM. Our solver provided the ability to change the underlying Stokes discretizations. For the results we use the Scott-Vogelius methods for methods outlined by Scott and Girault and Amara et al but the Taylor-Hood method for SUPG method. Our experience did not show significant difference in the

convergence and residual when changing the Stokes discretization, but only with the change of the Reynolds number and transport equation.

The following variational forms are coupled to the Stokes system. For example using the Scott-Vogelius method:

$$\forall \boldsymbol{v} \in V, \eta(\nabla \boldsymbol{u}, \nabla \boldsymbol{v}) + (\boldsymbol{z} \times \boldsymbol{u}, \boldsymbol{v}) = (\boldsymbol{f}, \boldsymbol{v}) \tag{5.8}$$

where V is continuous vector elements of degree greater than or equal to four and Z is a continuous element. Using Scott and Girault's notation where $\boldsymbol{z} = (0, 0, z)$. Standard Galerkin:

$$\forall \theta \in Z, \eta(z, \theta) + \alpha(\boldsymbol{u} \cdot \nabla z, \theta) = \eta(\mathbf{curl}(\boldsymbol{u}), \theta) + \alpha(\mathbf{curl}(\boldsymbol{f}), \theta) \tag{5.9}$$

Symmetrized:

$$\forall \theta \in Z, \eta(z, \theta) + \alpha(\boldsymbol{u} \cdot \nabla z, \theta) + \tfrac{\alpha}{2}((\nabla \cdot \boldsymbol{u})z, \theta)$$
$$= \eta(\mathbf{curl}(\boldsymbol{u}), \theta) + \alpha(\mathbf{curl}(\boldsymbol{f}), \theta) \tag{5.10}$$

Streamline, with a scalar parameter δ:

$$\forall \theta \in Z, \eta(z, \theta + \delta \boldsymbol{u} \cdot \nabla \theta) + \alpha(\boldsymbol{u} \cdot \nabla z, \theta + \delta \boldsymbol{u} \cdot \nabla \theta)$$
$$= \eta(\mathbf{curl}(\boldsymbol{u}), \theta + \delta \boldsymbol{u} \cdot \nabla \theta) + \alpha(\mathbf{curl}(\boldsymbol{f}), \theta \delta \boldsymbol{u} \cdot \nabla \theta) \tag{5.11}$$

Streamline Symmetric:

$$\forall \theta \in Z, \eta(z, \theta + \delta \boldsymbol{u} \cdot \nabla \theta) + \alpha(\boldsymbol{u} \cdot \nabla z, \theta + \delta \boldsymbol{u} \cdot \nabla \theta) + \tfrac{\alpha}{2}((\nabla \cdot \boldsymbol{u})z, \theta + \delta \boldsymbol{u} \cdot \nabla \theta)$$
$$= \eta(\mathbf{curl}(\boldsymbol{u}), \theta + \delta \boldsymbol{u} \cdot \nabla \theta) + \alpha(\mathbf{curl}(\boldsymbol{f}), \theta \delta \boldsymbol{u} \cdot \nabla \theta) \tag{5.12}$$

Regularized, with a scalar parameter ϵ:

$$\forall \theta \in Z, \epsilon(\nabla z, \nabla \theta) + \eta(z, \theta) + \alpha(\boldsymbol{u} \cdot \nabla z, \theta)$$
$$= \eta(\mathbf{curl}(\boldsymbol{u}), \theta) + \alpha(\mathbf{curl}(\boldsymbol{f}), \theta) \tag{5.13}$$

Table 5.1: Grade two theoretical estimates. X indicates unknown, ⟵ indicates estimate is derived from entry to the left and thus the same.

	Taylor-Hood	T-H ($W^2 \in \Pi$)	Scott-Vogelius
Standard Galerkin	X	X	quasi-optimal
Symmetric Galerkin	sub-optimal	X	quasi-optimal
Streamline diffusion	X	quasi-optimal	⟵
Streamline symmetric	X	quasi-optimal	⟵
Discontinuous Galerkin	quasi-optimal	⟵	⟵
Regularized	X	X	X
SU/PG	X	X	X

SUPG uses the same variational form as the streamline with the following the specified scalar:

$$\delta = \beta \frac{\boldsymbol{u} \cdot \boldsymbol{u}}{h^2} \text{for} 5.11 \tag{5.14}$$

The study of Stokes systems have produced numerous discretization for velocity and pressure spaces, but we focus our implementations on the Taylor-Hood and Scott-Vogelius elements. Taylor-Hood provides a simple discretization of low order but does not preserve the divergence free property of the velocity. The Scott-Vogelius elements do provide the divergence free property but require an order of four for the velocity. For non-divergence free spaces, theory suggests the transport term must satisfy the following condition:

$$\int_\Omega (\nabla \cdot \boldsymbol{u}) z w dx = 0 \qquad \text{for any } z, w \in Z \tag{5.15}$$

Which for low order Taylor-Hood elements would require piecewise linear or constant elements for W, but in our experiments we used discretized the same element as the velocity with no problems. See Table 5.1 for a listing of the known theoretical estimates for each discretization and stabilization estimates from Scott and Girault.

All the different discretizations suggested by Scott and Girault worked well for small α and low Reynolds numbers, as in the algorithm converged quickly. As either α or the Reynolds number increased, the algorithms tended to not converge at all. Amara et al created a

error estimator using the regularized discretization to help this problem. Unfortunately, the algorithm produced poor convergence rates for increase of α. In our experiments we use Hughes-Brookes SUPG to add the streamline terms. This method was able to increase α for small Reynolds number (Re << 1) but suffered the same convergence rates as the Amara work when Reynolds number was increased.

5.2   Implementation

Just as in the Stoke equation Rheagenimplements each numerical form and the user must input the boundary conditions and body forces as defined in DOLFIN. The iteration between the transport equation and Stokes system is implemented in the `Grade2Solver`. The class manages which element is being called through the parameter system and calls the correct generated simulation.

See Figures B.1– B.7 for the implementation of each variational form for the different numerical methods of solving the equations. Using the same implementation of the domain of the lid driven cavity from the Stokes equation, we are able to pass to the particular simulation, see Figure B.8.

Using the lid driven cavity defined in the previous chapter, Figure A.2, we show a small sample of the different parameters under this method, see Tables B.1 and B.2. All methods produced very similar responses but the Taylor-Hood element with SUPG resulted in a much smaller absolute residual than the other transport discretizations.

CHAPTER 6

OLDROYD-B TYPE FLUIDS

Olyroyd-B type fluids are models similar to the equations derived by Oldroyd's seminal work [57]. Such models are derived from a Maxwell solid representation of the polymers. The constitutive models with the following type are considered:

$$\lambda \overset{\nabla}{\boldsymbol{t}} + \boldsymbol{t} - 2\eta \boldsymbol{D} + \boldsymbol{g}(\boldsymbol{t}) = \boldsymbol{0}, \tag{6.1}$$

$$\overset{\nabla}{\boldsymbol{t}} \equiv \mathrm{d}t\boldsymbol{t} + \boldsymbol{u} \cdot \nabla\boldsymbol{t} - (\nabla\boldsymbol{u})^T \cdot \boldsymbol{t} - \boldsymbol{t} \cdot \nabla\boldsymbol{u}, \tag{6.2}$$

where $\boldsymbol{g}$ is a nonlinear relations specific to different models and $\overset{\nabla}{\boldsymbol{t}}$ is the upper convected stress. Over the years numerous models have been developed with the different values of $\boldsymbol{g}$ fitting to different characteristics of fluids.

For each of the Oldroyd-B models a common problem is numerically stabling simulating high Weissenberg number, $Wi = \lambda |\boldsymbol{u}|$. A number of numerical methods have been tried to reach higher Wi but often each numerical method is tried on a few models without much comparison on other models. A review by Baaijens [3] give a long literary review of the numerical methods applied to the Upper Convect Maxwell model but these papers only report on a few benchmarks. By using the automated methods of FEniCS, we have included numerous models and numerical methods into Rheagen; thereby, giving a researcher a means to test the models and methods side by side.

First we give some common models that are reported in numerous papers on different numerical problems. Next we discuss the various numerical discretizations and stabilization techniques for the models. Finally we give details on the implementations of the algorithms inside Rheagento solve the different models.

6.1 Common Models

Here we list three different models discussed in literature. These models build on each other allowing for different parameters to reduce one to the other. From a numerical standpoint these extra parameters often act to regularize the problem making the more complex looking equations often easier to solve. The Oldroyd-B and Upper convected model is know to blow up in many situations whereas the Phan Thein Tanner and Grmela models are reported to be considerably more stable.

6.1.1 Oldroyd-B

The Oldroyd-B equations uses a single relaxation time λ for the polymer system and splits the extra-stress into a solvent (Newtonian part) and polymer part:

$$T = T_p + T_s \qquad T_p + \lambda \overset{\nabla}{T_p} = 2\eta \boldsymbol{D} \qquad T_s = 2\eta_s \boldsymbol{D} \tag{6.3}$$

where η and η_s are polymer viscosity parameters and ∇ denotes the upper-convected time derivative:

$$\overset{\nabla}{\boldsymbol{T}} \equiv \frac{\partial \boldsymbol{T}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{T} - (\nabla \boldsymbol{u})^T \cdot \boldsymbol{T} - \boldsymbol{T} \cdot \nabla \boldsymbol{u} \tag{6.4}$$

6.1.2 Upper convected Maxwell

The upper-convected Maxwell model (UCM), the simplest Oldroyd-B model, is formulated by setting $\eta_s = 0$,(eqn 6.3)

$$\lambda \left(\frac{\partial \boldsymbol{T}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{T} - (\nabla \boldsymbol{u})^T \cdot \boldsymbol{T} - \boldsymbol{T} \cdot \nabla \boldsymbol{u} \right) + \boldsymbol{T} - 2\eta \boldsymbol{D} = \boldsymbol{0} \tag{6.5}$$

6.1.3 Phan Thein-Tanner

The Phan Thein-Tanner (PTT) model is derived from a Lodge network model [70].

$$\boldsymbol{T} + \frac{\lambda\epsilon}{\eta}tr(\boldsymbol{T})\boldsymbol{T} + \lambda \overset{\nabla}{\boldsymbol{T}} = 2\eta\boldsymbol{D} \tag{6.6}$$

Where ϵ is a dimensionless material parameter to fitted to experimental data. When $\epsilon = 0$ the model reduces to the UCM model and when $\lambda = 0$ it reduces to the Newtonian case.

6.2 Numerical Formulations

The numerical formulation of the Oldroyd-B type fluid models has proven to be much more difficult than the Newtonian case. Mixed formulations must model the stress tensor which requires stable elements to couple the extra fields. The larger Wiessenberg number turns the nature of the equations from elliptic to hyperbolic thus requiring stabilization. Furthermore almost every solver in practice uses continuation methods on λ to successfully solve higher Wiessenberg numbers. Here we review different numerical formulations for the first two aspects before describing the implementation in the next chapter.

6.2.1 MIX

The simplest formulation from the UCM constitutive model, Eq.(6.5) is the mixed method (MIX) of Kawahara and Takeuchi [42]. Find $\boldsymbol{T}$, $\boldsymbol{u}$, and p such that for all $\boldsymbol{S}$, $\boldsymbol{v}$, and q:

$$(\boldsymbol{S}, \lambda \overset{\nabla}{\boldsymbol{T}} + \boldsymbol{T} - 2\eta\boldsymbol{D}) = 0, \tag{6.7}$$

$$((\nabla\boldsymbol{v})^{T}, 2\eta_e\boldsymbol{D} + \boldsymbol{T}) - (\nabla \cdot \boldsymbol{v}, p) = 0, \tag{6.8}$$

$$(q, \nabla \cdot \boldsymbol{u}) = 0 \tag{6.9}$$

Kawahara and Takeuchi used a quadratic element for the velocity and linear for the pressure and stress. Using this model, they were able to compute several simulations with low Reynolds number and Wiessenberg numbers, around 80 and 1 respectively. While producing results from such a simple model is promising criticism comes quickly from other studies trying to reproduce the work [22]. Kawahara and Takeuchi used a very course mesh and when replicated we found to that the Newton-Raphson method does converge as suggested, but once the mesh is coursed at all numerical artifacts tend to blow up the solutions.

6.2.2 Marchal Crochet

With a series of experiments Marchal and Crochet [55] show that Kawahara and Takeuchi have an unstable discretization for the velocity stress coupling. They produce two alternative discretizations to deal with the instabilities. The first is to use a Hermite elements [55]. The difficulty in implementing such elements and the large number of dofs limited the use of the Hermite element in further research. The second discretization proposed a 4X4 bilinear subelement for each stress component [56].

Additionally Marchal and Crochet apply stabilization parameters to handle the hyperbolic nature of the higher Wiessenberg number. The first method was to use SUPG method which adds an upwind stabilization parameter to stress.

$$(\boldsymbol{S} + \alpha \boldsymbol{u} \cdot \nabla \boldsymbol{S}, \lambda \stackrel{\nabla}{\boldsymbol{T}} + \boldsymbol{T} - 2\eta \boldsymbol{D}) = 0, \tag{6.10}$$

where α is a scalar parameter, the most common being the characteristic element length over the scaling of the velocity. The SUPG formulations failed near singularities due to oscillations, to correct this they produce formulate a non-consistent stabilization method by

only applying the upwinding term to the convective term of the constitutative equation:

$$(\boldsymbol{S}, \lambda \overset{\nabla}{\boldsymbol{T}} + \boldsymbol{T} - 2\eta\boldsymbol{D}) + (\alpha\boldsymbol{u} \cdot \nabla\boldsymbol{S}, \alpha\boldsymbol{u} \cdot \nabla\boldsymbol{T}) = 0, = 0, \qquad (6.11)$$

This techniques allow for the larger Weissenburg numbers.

6.2.3   EVSS and DEVSS

Further work on the stability of coupling mixed spaces has developed other workable elements or methods such as discontinuous Galerkin methods [31, 32]. This work has led to a generalized version of the LBB condition with three criteria:

1. The velocity–pressure space much satisfy the usual LBB condition,

2. if a discontinuous space is used for $\boldsymbol{T}$ then the rate of strain tensor $\boldsymbol{D}$ must also be in the same space, and

3. if a continuous space is used for $\boldsymbol{T}$ then the number of internal nodes must be larger than the number of nodes on the side of the element for the velocity.

The third criteria is satisfied by Marchal-Crochet's macro element and the discontinuous Galerkin methods have been successful at satisfying the second criteria. But these elements are still quite limiting for the range of possible elements. Baranger and Sandri later show that the requirement of this condition is not valid if there is a viscoelastic contribution ($\eta_e \neq 0$) [5].

The result of Baranger and Sandri open for a number of method that focused on retaining the elliptic contribution of the momentum equations, $(\nabla\boldsymbol{v}^T, \boldsymbol{D})$. The first of these methods, the elastic viscous stress splitting (EVSS) formulation [7, 59], uses a change of variables.

$$\boldsymbol{\Sigma} = \boldsymbol{T} - 2\eta\boldsymbol{D} \qquad (6.12)$$

This changes the UCM model to the following system:

$$(\boldsymbol{S}, \lambda \overset{\nabla}{\boldsymbol{\Sigma}} + \boldsymbol{\Sigma} + 2\eta \overset{\nabla}{\boldsymbol{D}}) = 0, \tag{6.13}$$

$$((\nabla \boldsymbol{v})^T, 2(\eta + \eta_e)\boldsymbol{D} + \boldsymbol{T}) + (\nabla \cdot \boldsymbol{v}, p) = 0, \tag{6.14}$$

This change of variables is not available for all constitutive equations and furthermore adds convected terms on the rate of strain tensor. Many solutions to the problem have been proposed but the most general approach was proposed by Guénette and Fortin [39], known as discrete EVSS (DEVSS).

The DEVSS method augments the change of variable:

$$\boldsymbol{\Sigma} = \boldsymbol{T} - 2\alpha \overline{\boldsymbol{D}} \tag{6.15}$$

where α is any positive constant (often set to η) and $\overline{\boldsymbol{D}}$ is the projection of the rate of strain tensor into the stress space, which is the same as the pressure space. Taking the divergence of the change of variable shows no need to introduce the variable $\boldsymbol{\Sigma}$, and the momentum equation becomes

$$-2\alpha\nabla \cdot \boldsymbol{D} + \nabla p = \nabla \cdot T - 2\alpha\nabla \cdot \overline{\boldsymbol{D}} \tag{6.16}$$

For the discretization Guénette and Fortin used P2 for the velocity, P1 for each stress element, and a discontinuous P1 for the pressure.

This formulation with the SU method has been widely used and is very successful at high Weissenburg numbers.

6.3 Implementation

The combination of methods to stabilize the hyperbolicity of the constitutive equations and retain the elliptic contributions has created a large number of available methods for solv-

Table 6.1: Stabilization versus elliptic operator discretization. D represents a change in element discretization and V represents the change in variational form from the reference MIX method.

	None	SUPG	SU
Marchal-Crochet	D	D+V	D+V
DG	D	D+V	D+V
EVSS	V	V	V
DEVSS	V	V	V

Table 6.2: Element wise operation count in Rheagen

	MIX	MIX/SUPG	MIX/SU
Oldroyd-B	29543	90870	67230
UCM	29495	90794	67154
PTT	53750	91603	60197
	DEVSS	DEVSS/SUPG	DEVSS/SU
Oldroyd-B	29553	90880	58904
UCM	29505	90804	58828
PTT	53750	91603	60197

ing the models, see Table 6.1. Each of these models require changing either the underlying discretization or the variational form. Using the FEniCStools, Rheagengenerates the formulations from the high level UFLlanguage and generates low level C code. This allows for models to be added easily and generated with a large range formulations. The current version of Rheagenhas implemented a subset with a simple object model for implementing further models 6.2. See Figures C.1– C.6 for the implementation of each variational form for the different numerical formulations. It should be noted that the UFL stadard is to set the residual formulation for the Newton-Raphson as the variable L and a as the Jacobian.

Rheagenuses the Newton-Raphson method with a simple predictor/corrector continuation method to solve each model and formulation. Baaijens reports the standard method is to stepping along the values of λ using each iteration as the next solution method. For Rheagenwe implemented a more robust predictor corrector method [66]. Because computing the Hessian of the system is quite expensive, we adjust the time step by halving the step size of λ for every iteration the Newton-Raphson method fails to converge, and increasing it

(a) Planar contraction mesh

(b) Velocity magnitude of planar with stream-lines
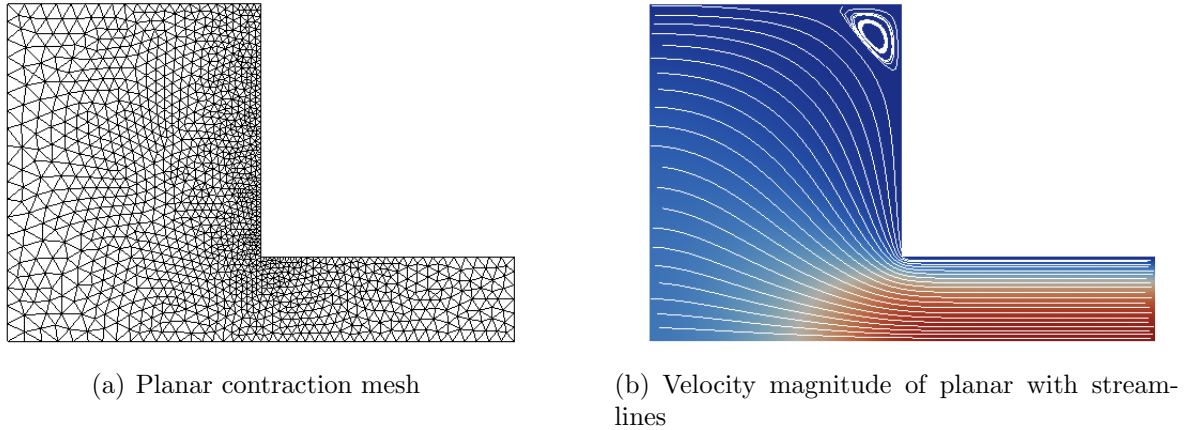
Figure 6.1: Planar contraction mesh and solution plot

by a factor $2^{1/4}$ thus doubling the time step every four steps.

We give two test problems to give a sense of the ability to test different simulation scenarios. The first is the lid driven cavity already defined, see Figure A.2, and the second is a four to one planar contraction problem (about a line of symmetry). Both of these problems have a Reynolds number of one divided by the viscosity, or *eta*, and pressure singularities that make them require more iterations than simpler simulations. See the definition of the planar contraction in Figure C.8 and plots in Figure 6.1. To give some view of the different methods, we give the number of iterations in the Newton-Raphson process, see Tables C.1–C.7 for a range of parameters on the given models using the lid driven cavity problem, A mark of 'X' indicates that the given method did not converge in 5000 iterations. The different methods did better for different parameters.

From the large range of parameter space it is eash to see the need for a tool to compare methods side by side. The simulations could lower the Reynolds number with higher viscosity (η) and the Wiessenberg number (λ) to get more less linear effects. Perhaps the most significant deviation from Baaijens is the affect of η_e on the convergence. Most literature set this parameter to zero and solved a higher Wiessenberg number, but few actually used the full Oldroyd-B model, $\eta_e \neq 0$. Our numbers are consistent with these results, except for the

SUPG methods. It is not surprising that SUPG has more problems as when Marchal and Bezy introduced it, they also had difficulties with it on the Newtonian case.

CHAPTER 7

POLYNOMIAL SOLVERS

Perhaps the least discussed but equally important aspect of solving non-Newtonian fluids is the use of a robust nonlinear solver. Existence and uniqueness has not been proven for many models so when a researcher finds a method not converging it is difficult to know if it is the implementation or the method that is at fault. Furthermore, a single answer to the model may not be correct if there is a bifurcation point that had not been predicted. Rheagenuses continuation methods to help find good initial guesses for a Newton-Raphson solver, but for even for some simulations this does not result in convergence. There has been some work to make a fast Newton-Raphson method of solving the resulting sets of equations [3], but this work only solves the system faster and not robustly.

Discovering new features in these non-linear models is very difficult and often goes decades without results on the matter. In this chapter we propose using the new advances of polynomial solvers using homotopy continuation to create robust solvers that give all solutions to a problem. This method will formulate a polynomial system based on the FEM discretization that has been subsequently automated. Because we are using an automation system, we are able to call the polynomial solver rather than Newton-Raphson solver with very little overhead.

The hope with this work was to give an experimental toolkit for proposing small problems and solving them. Additionally one would be able to build new solver techniques for these problems using ideas such as Full Approximation System or nonlinear Gauss-Seidel methods. The system we describe here produces the systems and can use the polynomial solvers for simple problems but are unfortunately solving systems of notable size in a timely manner. Nonetheless there is a great deal of potential for these systems because they are very amenable to course grain parallelism. The small data share between processes make them a prime candidate for heterogeneous architectures that have recently received so much attention.

In this chapter we give some details on the advances relevant for nonlinear FEM problems in polynomial solvers. Formulate how we build the polynomials systems from our automation framework. Take a simple examples and show the potential for building robust systems on top of them.

7.1   Polynomial Solvers using Homotopy Continuation

Polynomial solvers using homotopy continuation methods are a set of robust methods capable of solving arbitrary polynomial systems. The focus is on obtaining all solutions instead of solving for a single solution quickly. This method also features a great deal of coarse grain parallelism. Its robustness make it an ideal candidate for numerical experimentation on difficult problems.

The method computes all finite solutions of a polynomial system $P(x) = \mathbf{0}$ where $P = (p_1, p_2, \ldots, p_n)^T$ for n unknowns $x = (x_1, x_2, \ldots, x_n)$. The solve begins by defining a trivial start system, $Q(x) = 0$ and then follows the path of a homotopy $\mathcal{H}$ defined between Q and P

$$\mathcal{H}(x, t) = (1 - t)Q(x) + tP(x) \tag{7.1}$$

The solutions of $\mathcal{H}$ at $t = 0$ are traced to $t = 1$. Many numerical techniques have been developed to insure that a trivial start system Q exists, that paths from $t = 0$ to $t = 1$ are smooth, and finally that every solution at $H(x, 1)$ is accessible from a solution at $H(x, 0)$.

Original work on these polynomial solvers used Bézout's theorem to determine the number of paths whic have to be tracked, the product of the degrees of each polynomial. For deficient polynomial systems, which have many less solutions, the tracking of extraneous paths was wasteful. Current codes, such as PHCpack [72] instead use the Bernshtein theorem to estimate the number of actual roots of the system, giving a much smaller number of paths. This main result, along with numerous smaller improvements to the continuation

algorithms, have made polynomial solvers much more efficient and more suited to the types of problems which arise from PDEs. PHCpack is one of several open-source polynomial solvers available using these new techniques, along with Bertini [6].

7.2   Nonlinear finite element assembly

Algorithm 7.1 Simple finite element method, revisited

1: Formulate mesh.
2: Create degrees of freedom (dofs) on mesh.
3: **for** each element **do**
4:     Evaluate $a^e(i, j)$ and $L^e(j)$ for $i, j \in$ reference dofs.
5:     Transform $a^e(i, j)$ and $L^e(j)$ to paritial result of $a(i, j)$ and $L(j)$
6:     Accumulate with previous partial results for $a(i, j$ and $L(j)$ in matrix equaitons
7: **end for**
8: Apply Dirichlet boundary conditions.
9: Perform algebraic solve.

The heart of the FEM is assembling a variational formulation of a PDE into algebraic equations and solving them, see Algrothim 7.1. The domain is meshed by the element shape $\mathcal{K}$. On each of these elements, the functions are represented via coefficients of the basis functions in $\mathcal{P}$ on degrees of freedom (dofs) $\mathcal{N}$. For each dof, an algebraic equation is formed by the variational form:

$$a(u; v) = L(v) \tag{7.2}$$

where u is the trial function being solved for and v is the test function. For Galerkin methods both these spaces are the same space.

To form these algebraic equations the local variational equation, $a^e(\phi_i, \psi_j)$ and $L^e(\psi_j)$, is evaluated for each $\phi_i, \psi_j \in \mathcal{P}$ where the index i correspond to the dofs representing the solution u. Because some shared between multiple elements, the global variational equations is accumulated in a global matrix. Most PDEs require the application of boundary conditions. These are either weak conditions, such as Neumann conditions that are assembled

through the variational form, or strong conditions, such as Dirichlet which fix some of the coefficients eliminating the corresponding equation.

For nonlinear problems, the standard technique is to linearize the problem around a previous solution then use an iterative method to update the solution. This technique works well with well-posed problems where the solution space has only one minimum. Many problems do not have these desirable properties and practice has developed heuristics, such as continuation and regularization, to accommodate the problems. Even with these heuristics iterative methods only finds a single solution to a problem and for many problems the iterative method does not converge at all.

Algorithm 7.2 Finite element method for generating nonlinear polynomials

---
1: Formulate mesh.
2: Create degrees of freedom (dofs) on mesh.
3: **for** each element **do**
4:     **for** each variational term, a **do**
5:         Evaluate a^e for reference dof
6:         Transform a^e to partial result of a
7:         Accumulate with previous partial result in tensor
8:     **end for**
9: **end for**
10: Apply Dirichlet boundary conditions elminating constrained dofs.
11: Generate symbolic nonlinear polynomial equations in text file.
12: Use PHCpack to solve polynomial equations.

To manage the nonlinear FEM assembly for the polynomial solvers, there are several ways we augment the standard assembly process, see Algorithm 7.2. The most notable change is instead of creating a single matrix equation, the nonlinear tensor equations will have higher ranks. We use a sparse tensor data structure to store the intermediate coefficients while the FEM integrals are evaluated and accumulated.

The second change is to eliminate all constrained dofs. Many FEM codes merely set the matrix entries to represent the constrained value, but this did not work with PHCpack. This may seem a small issue but it is one example of how slightly different FEM assembly

algorithms are generated differently.

This process of counting the number of roots is still based on degrees of the polynomial and is a symbolic calculation, thus the generated polynomials must be converted from coefficients to symbolic equations. For our purposes we generated the text file because it was nice for archival purposes but since the core of PHCpack is written in Ada and our tools written in Python it was the simplest place to generate the polynomials. The system then calls PHCpack on the generated text file.

The storage needed for the full nonlinear FEM problem is quite large. The FEM matrix is usually the number of unknowns times the bandwidth, our structure adds additional terms that is the number of unknowns time the bandwidth to the degree of nonlinearity. For our prototype we have not tried to optimize the storage in any way, but have focused on small proof-of-concept problems.

The polynomial solver will do more work than just tracing one solution path, but it will also use less communications. Since every path is tracked independently, the solver can scale to parallel architectures trivially, whereas the iterative solutions require communication for each iteration and update. Because of this communication pattern, we are hopeful that the speed of polynomials solvers will increase with newer architectures and the number of unknowns are able to grow. Additionally many solutions may be unphysical or unstable, so for practical use one must pick a solution based desired criteria.

While the issue of storage and run time still present a problem for our simple prototype, this work is still in its infant stage. Already though use of these will give some comfort to researchers who are working with difficult problems and want some sense of the solution space.

7.3 Case studies

Our prototype began with two prototype planar problems, the first is a simple problem with numerous solutions and the second is the well known Navier-Stokes equations that has one solution but because of high Reynolds number requires special care. We propose future working with problems that have few theoretical results, such as the Oldroyd-B fluids, but choose these problems because they have simple discretizations and demonstrate the ability to use FEM to generate the polynomial systems.

7.3.1 Finding Multiple Solutions

The simple problem we give is the Poisson equation with an added nonlinear term u^2:

$$u^2 - \Delta u = f \tag{7.3}$$

The domain was the unit square with Dirichlet boundary conditions, $u(x, y) = 1.0 + x + y$, and $f = (1.0 + x + y)^2$. Adding the nonlinear term turns this problem from a well-posed problem with one solution to a problem with two solutions per degree of freedom. The unit square is meshed into NXN squares each divided into two triangles.

We divide the variational problem into the quadratic, bilinear, and linear forms, b, a, L:

$$b(u, w; v) = \int_\Omega u \cdot w \cdot v d\Omega \tag{7.4}$$

$$a(u; v) = \int_\Omega \nabla u \cdot \nabla v d\Omega \tag{7.5}$$

$$l(v) = \int_\Omega f \cdot v d\Omega \tag{7.6}$$

Where v is the test function and u is split into two trial functions u, w for the quadratic term. For our simulations we used Lagrange polynomials tabulated by FIAT. The coefficients were then summed into the tensors, B, A, and L. From these tensors, for each degree of freedom

i a polynomial is generated:

$$\sum_{j,k} B[i,j,k] \cdot u_j \cdot u_k + \sum_{j} A[i,j] \cdot u_j = L[i] \tag{7.7}$$

Once the polynomial equations are generated, PHCpack is called and returns the solutions to the equations.

7.3.2   A Robust Navier-Stokes Solver

The second problem we look at is the steady state Navier-Stokes equation:

$$\boldsymbol{u} \cdot \nabla \boldsymbol{u} - \mu \Delta \boldsymbol{u} + \nabla p = \boldsymbol{f} \tag{7.8}$$

$$\nabla \cdot \boldsymbol{u} = 0 \tag{7.9}$$

Once again we use the same unit square mesh with NXN squares each divided into two triangles. We defined a number of exact solutions to test the method then set Dirichlet boundary conditions and f accordingly.

The variational problem can be divided into three parts as before but because we have a $\boldsymbol{u}$ as a vector field and p scalar, we use a mixed method.

$$b(\boldsymbol{u}, \boldsymbol{w}; \boldsymbol{v}) = \int_{\Omega} \boldsymbol{u} \cdot \nabla \boldsymbol{w} \cdot \boldsymbol{v} d\Omega \tag{7.10}$$

$$a(\boldsymbol{u}, p; \boldsymbol{v}, q) = \int_{\Omega} (\mu \nabla \boldsymbol{u} \cdot \nabla \boldsymbol{v} - p \nabla \cdot \boldsymbol{v} + q \nabla \cdot \boldsymbol{u}) d\Omega \tag{7.11}$$

$$l(\boldsymbol{v}) = \int_{\Omega} \boldsymbol{f} \cdot \boldsymbol{v} d\Omega \tag{7.12}$$

We using the Taylor-Hood space for the function space definition. Thus each component of $\boldsymbol{u}$ and $\boldsymbol{w}$ is a Lagrange polynomial of degree two and p is one.

	dofs	mixed volume	solutions found	time
TH P2/P1 5X5	122	1	1	70ms
TH P2/P1 6X6	197	1	1	95ms
TH P2/P1 8X8	401	1	1	930ms

Table 7.1: PHC data for stokes problem

	dofs	mixed volume	solutions found	time
P1 4X4	4	16	16	60ms
P1 6X6	16	65536	65536	5318s

Table 7.2: PHC data for nonlinear Laplacian problem

7.4 Results

To first test the method we were using we constructed the Stokes problem by leaving out the nonlinear form in the Navier-Stokes example. This produces a linear problem and should be easy to solve as it has only one solution by both the mixed volume and Bézout methods. PHC solved these small problems rather quickly but had problems with large problems, see Table 7.1.

The first nonlinear problem was solvable using PHCpack and Bertini for small problem size (< 20 dof). While there were only two real solutions the number of complex solutions is consistent with Bézout's theorem that is 2^N, see Figure 7.2. The large number of solutions to track already made small problems unfeasible. This is somewhat expect for a problem that has a full set of solutions under the Bézout theorem, but since planar Navier-Stokes is known to be unique the hope is the mixed volume will reduce the number of solutions to track.

Navier-Stokes systems proved not to be feasible using PHCpack or Bertini. The smallest grid without being completely constrained is the 3X3 grid and produces 116 equations. PHCpack does report that the mixed volume is much less than Bézout's theorem, 2^{24} versus 2^{116}. But the continuation tracking provided by PHCpack still takes much longer than expected because as the simple predictor corrector methods will reduce the continuation

step to only very small steps, around $1e-8$ for the 3X3 grid. The fact that we are tracking millions of solutions at such a small continuation step required runtimes of weeks. Because of these results we did not continue to code up the complex fluids as they would produce more nonlinear equations which have been harder to track than the Navier-Stokes system.

CHAPTER 8

CONCLUSIONS

This thesis has presented and applied automated scientific computing as a paradigm for lowering the barriers of entry for scientists to use state of the art computational methods. Posing the simulation with high level mathematical abstractions allow for optimizations that are not available to the low level code compilers. Using these representations we are able to implement many theoretical methods quickly allowing comparisons in the same code. As our application to complex fluids has shown, comparing numerous models and methods is able to evaluate methods for further analysis. Additionally we were able to use these programming techniques to evaluate a different type of equation solver.

Our studies are certainly only the tip of the possibilities. The FEniCS Application Gallery is a place where users are providing numerous application codes written in this same manner. New architecture models, such as general purpose GPUs, are becoming popular allowing our general methods to generate different low level code based on the computer. These opportunities only increase the need for higher level algorithmic methods for computation as the expertise is further increased to achieve top performance.

We close with a proposal for researching the automation of the assembly algorithm itself. By automating this procedure, we are able to leverage many different FEM methods that have not been discussed here. Since the assembly routines for the FEniCS code still remains a dominant step, it is a piece that will further close the gap on theory and practice.

8.1   Future Directions: FEM Assembly Automation

Finite element assembly is a simple loop summing integrals and inserting values into a matrix. But anyone who has implemented a finite element code will tell you, this is often the most error prone part of the simulation. Even though other parts of finite element methods (FEM)

have been automated, the assembly algorithm usually remains hand coded and a major bottleneck. Handling extra features that are necessary for large scale multiphysics problems, such as hanging nodes for adaptivity or coupling of several fields, increases complexity. In the end, this seemingly simple algorithm often drives libraries to limit the range of elements they support, encouraging the divide between application and theory.

By viewing assembly at a high level, it can be formally derived and efficient low level code can be generated. This method of generating libraries allows tuning to given architectures – from the multicore laptop to the world's largest supercomputer with modern accelerator technology. Automated program generation has been successfully applied to low level linear algebra libraries [9, 40], and its performance rivals state-of-the-art hand tuned codes on numerous architectures [10, 62, 71]. Here we outline techniques for extending this paradigm to finite element assembly. The complexity of FEM codes and their ubiquitous use in scientific computation for solving partial differential equations make it a prime candidate for automation.

Background

Consider Figure 8.1 with three linear elements A, B, and C. The first step of assembly is to compute the local element matrices, usually done by evaluating a user defined weak form. The FEniCS project [28, 53] is a set of software packages and interface specifications for automating FEM. With it the user inputs a weak form in a high level language, and efficient low level code is generated to compute these matrices. Different representations of the weak form radically speed the code [46, 58], and a near optimal reuse of computed entries can be computed with discrete optimization techniques [47, 74]. These advances have pushed the major bottleneck from the element computation to inserting them into the global matrix.

In this step, the unassembled element matrices are inserted into a global system by reducing the connected degrees of freedom (dofs). In Figure 8.1, dofs of connected nodes

such as 1 and 7 or 2 and 5 are summed. For the FEniCS project, the runtime of this step is as much as ten times that of computing the element matrices. This step is difficult for a few reasons. First, the code must have a mapping from local to global dofs to know which are to be reduced. This mapping is a large data structure that must be distributed. By using highly regular meshes, applications are able to mitigate the amount of information stored but require an exponentially larger number of elements to achieve the same accuracy as unstructured meshes. Second, for more complex elements the connection is not enough information and geometric data must be analysed as well. This geometric data comes as input from the mesh and reduces opportunities for pre-computing parts of the element matrices. Typical information might include the orientation of vector dofs or the direction of integration for dofs representing moments, each type of dof requiring different reduction operations.

Other features, such as adaptive mesh refinement which can lead to hanging nodes, add additional complexity. In the case of Figure 8.1, the element C uniquely defines the space along its edge, the hanging dofs 3 and 4 need to be eliminated and the system rewritten without their equations. This is one type of equation elimination but others exist and are commonly used. One common practice for high order elements, called static condensation, is to perform Gaussian elimination on the unassembled matrices before insertion and is another piece of the algorithm that can be generated.

Current work

The example above is a simple demonstration of the assembly algorithm. Multiphysics problems assemble several fields at once which often have non-obvious dependencies. For example, the non-Newtonian fluid solvers must couple eleven fields for the two dimensional equations and apply extra numerical stabilization parameters that couple over half the fields. To solve such problems, state-of-the-art codes use specialized solvers tuned to the individual problems parameters and the computer architecture. Creating codes that derive the assembly
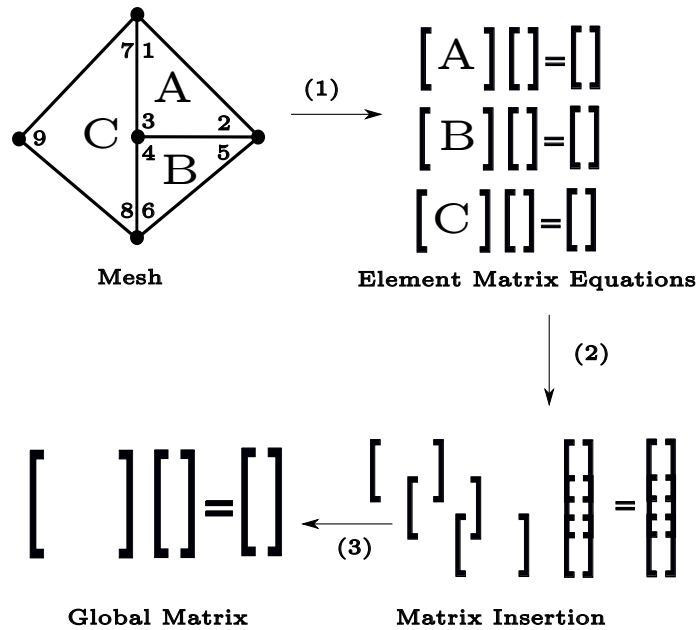
Figure 8.1: Assembly algorithm 1: (1) Compute element matrices, (2) Reduce connected nodes, and (3) Constrain hanging node.

algorithms may be able to extract the dependencies between fields and optimize for these dependencies. Furthermore, the code generation process will allow a user to specify a wide variety of complex elements with the same technology that generates simple elements.

Inspiration for this work comes from the FLAME [40] and SPIRAL [61] projects. Both of these libraries begin deriving algorithms from a high level description of the problem, using a mechanical process similar to that used by experts implementing optimized libraries. These projects generate optimized libraries for a wide range of algorithms, often out performing codes hand tuned by teams of experts. In principle this high level process can leverage knowledge of computational costs, memory footprint, and actual performance measurements to optimize the code. Combining the high level model with new architecture details can produce a highly optimized library for a new platform without extensive rewriting of the code.

Research Objectives

The goal is to generate code for FEM assembly of large scale multiphysics problems which directly challenge state-of-the-art solvers. These solvers are used in grand challenge problems in numerous fields and drive many technological advances [52].

Task 1: Develop an assembly generator library. Using the different steps of assembly algorithms, the library will generate code for the given problems. The first target will be standard assembly schemes adding simple optimizations such as static condensation. Further work will include assembly for matrix-free methods [41], discrete optimization techniques, and data flow analysis to generate code for specific meshes.

Task 2: Build community code interfacing with FEniCS abstractions. Currently the element matrix is generated by the FEniCS Form Compiler and then used by a hand coded assembly routine. This project would take that output and automatically build the assembly algorithm. The code will conform to the FEniCS assembly application program interface (API) [1], allowing the use of other form compilers and element libraries. It will also provide a simple, documented API and be distributed over the web.

Task 3: Automatically tune to architecture dependent models. The underlying architecture has a large effect on the proper assembly technique. The library will incorporate the specific architecture into the optimized code.

Task 4: Target large scale multiphysics simulations. The project will use large scale multiphysics problems as its testbed. Additionally, this project makes it possible to derive assembly for some advanced multiphysics techniques. Two techniques that will be tried are finite element tearing and interconnecting [30] and physics based preconditioning [52]. Both can be scheduled to match the hardware model, assembling different fields in appropriate blocks and scheduling solves based on the field interactions.

Assembly Step	Eliminated	Size
(1) Compute all element matrices		9 eqs
(2) Reduce connected nodes	4, 5, 7, 8	5 eqs
(3) Constrain hanging node	3	4 eqs

(a) Assembly Algorithm 1

Assembly Step	Eliminated	Size
(1) Compute all element matrices		9 eqs
(2) Constrain hanging nodes	3, 4	7 eqs
(3) Reduce connected nodes	5, 7, 8	4 eqs

(b) Assembly Algorithm 2

Assembly Step	Eliminated	Size
(1) Compute matrices A and B		6 eqs
(2) Reduce connected node	4, 5	4 eqs
(3) Compute matrix C		7 eqs
(4) Reduce connected nodes	7, 8	5 eqs
(5) Constrain hanging nodes	3	4 eqs

(c) Assembly Algorithm 3

Figure 8.2: Three variants of the assembly algorithm from Figure 8.1 using loop rearrangement.

Methods

In the simple example in Figure 8.1, the assembly algorithm reduces the nine dofs to four. A number of algorithms can be used to achieve this goal, (see Figure 8.2). Algorithm 1 reduces all connected nodes first then constrains the hanging nodes, and Algorithm 2 does the opposite. Algorithm 3 intertwines the computation of the element matrices with reductions and applies constraints last. Since Algorithm 2 constrains first, it uses one less floating point operation for the following reduction, but if the cells are on separate processes, Algorithm 1 will require less communication. Algorithm 3 has the advantage of temporarily storing fewer equations. This small example shows that three algorithms arise from the order of assembling the dofs and that the performance is dependent on underlying architecture.

Goal-oriented algorithm derivation: Just as the FLAME project used loop invariants to rearrange linear algebra algorithms, formal derivation of assembly can rearrange loops to describe the range of available algorithms. As shown in Figure 8.2, alternative

71

assembly algorithms can be generated by reducing and eliminating dofs while producing the same global system of equations.

Discrete optimization: Techniques such as graph max flow and optimal traversals enable assembly algorithms to be analyzed for optimal flops, memory accesses, and/or data dependence. These techniques have been used by the FEniCS project for element matrices and can be extended to the entire assembly algorithm.

Code generation techniques: Code generation allows high level domain specific abstractions to benefit from the runtime benefits of low level codes. By generating code and optimized library calls, the assembly algorithms will be applicable to large problems. Furthermore, by generating a simulation for a specific mesh, it is possible to avoid the need for the large dof map data structure.

Impact

Automatic code generation lowers the barriers of entry to large simulation inviting a larger pool of researchers to attempt solutions to grand challenge problems. Researchers will have access to optimized code that can run on a variety of architectures without needing to understand the underlying implementation details. Their focus will remain on formulating correct models, not coding details.

Also because developing optimized libraries is such a labor intensive process, code generation tools will allow the expert to exploit new optimizations and regenerate libraries with much less effort. Using high level specifications and code generation reduces the huge problem of maintaining code in the face of new architectures and operating systems.

APPENDIX A

CODE AND RESULTS FROM STOKES EQUATIONS

```python
# Define the boundary domains
class NoSlipDomain(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary

class PinPoint(SubDomain):
    def inside(self, x, on_boundary):
        return x[0] < DOLFIN_EPS and x[1] < DOLFIN_EPS

# Define mesh
mesh = UnitSquare(h_num, h_num, "crossed")

# Test problem specification
prob = {'f' : ("28*pi*pi*sin(4*pi*x[0])*cos(4*pi*x[1])",
                "-36*pi*pi*cos(4*pi*x[0])*sin(4*pi*x[1])"),
        'u' : ("sin(4*pi*x[0])*cos(4*pi*x[1])",
                "-cos(4*pi*x[0])*sin(4*pi*x[1])"),
        'p' : "pi*cos(4*pi*x[0])*cos(4*pi*x[1])"
    }

# Instantiate the boundary conditions
noslip_domain = NoSlipDomain()
noslip = Expression(prob['u'], V=V)
pinpoint = PinPoint()
pin_val = Expression(prob['p'], V=Q)
bc0 = DirichletBC(W.sub(0), noslip, noslip_domain)
bc1 = DirichletBC(W.sub(1), pin_val, pinpoint, "pointwise")
bc = [bc0, bc1]

# Define the RHS
f = Expression(prob['f'], V=V)
```

Figure A.1: Code for defining the test domain

```cpp
// Boundary Subdomain marking walls
class Wall : public SubDomain
{
public:
    bool inside (const double *x, bool on_boundary) const
  {
    return on_boundary;
  }
};

// Boundary Subdomain marking lid
class Lid : public SubDomain
{
public:
    bool inside (const double *x, bool on_boundary) const
  {
    return on_boundary && (x[1] > 0.75 - DOLFIN_EPS);
  }
};

// Function for top flow for velocity
class TopFlow : public Expression
{
public:
  TopFlow() :
    Expression(2)
  {}
  void eval (double* values, const std::vector<double>& x) const
  {
    values[0] = 4.0/1.0 * x[0] * (1.0 - x[0]);
    values[1] = 0.0;
  }
};

// Function for no-slip boundary condition for velocity
class Noslip : public Expression
{
public:
  Noslip() :
    Expression(2)
  {}
  void eval(double* values, const std::vector<double>& x) const
  {
    values[0] = 0.0;
    values[1] = 0.0;
  }
};
```

Figure A.2: Code for defining the lid domain

```
// Create a unit square
UnitSquare mesh(32, 32, "crossed");
Wall wall;
Lid lid;

// No-slip boundary condition for velocity
Noslip noslip;

// Lid boundary condition for velocity
TopFlow topflow;

// Create a mesh function, marking boundaries by the
// position the bc function is in the bc func array.
MeshFunction<unsigned int> vel_subdomains(mesh, mesh.topology().dim() -
1);
vel_subdomains = 2;
wall.mark(vel_subdomains, 0);
lid.mark(vel_subdomains, 1);
std::vector< GenericFunction* > vel_bc_funcs;
vel_bc_funcs.push_back(&noslip);
vel_bc_funcs.push_back(&topflow);

// Create fluid and solvers
Fluid fluid(mesh, vel_subdomains, vel_bc_funcs);
Constant zero(2, 0.0);
```

Figure A.3: Code for defining the lid domain continued

```
# Define function spaces
V = VectorFunctionSpace(mesh, V_element, V_order)
Q = FunctionSpace(mesh, P_element, P_order)
W = V + Q

# Define test and trial functions
(v, q) = TestFunctions(W)
(u, p) = TrialFunctions(W)

# Define the variational problems
a = (inner(grad(v), grad(u)) - div(v)*p + q*div(u))*dx
L = inner(v, f)*dx
if stabilized:
    h = CellSize(mesh)
    beta = 0.2
    delta = beta*h*h
    a += delta*inner(grad(q), grad(p))*dx
    L += inner(delta*grad(q), f)*dx
pde = VariationalProblem(a, L, bc)

# Assemble and solve
U = pde.solve()
```

Figure A.4: Code for defining the various mixed methods, see Table 4.1

```python
# Define function space
V = VectorFunctionSpace(mesh, "CG", V_order)

# Define test and trial functions
v = TestFunction(V)
u = TrialFunction(V)

# Define auxilary function and parameters
w = Function(V);   r = 1e3

# Define the variational problem
a = (inner(grad(v), grad(u)) - r*div(v)*div(u))*dx
L = (inner(v, f) + inner(div(v), div(w)))*dx
pde = VariationalProblem(a, L, bc0)

# Iterate to fix point
iters = 0; max_iters = 100; U_m_u = 1
while iters < max_iters and U_m_u > 1e-8:
    U = pde.solve()
    w.vector().axpy(c, U.vector())
    if iters != 0:
        U_m_u = (U.vector() - u_old_vec).norm('l2')
    u_old_vec = U.vector().copy()
    iters += 1
```

Figure A.5: Code for defining the iterated penalty methods

```python
VP10 = VectorFunctionSpace(mesh, "CG", 10)
P10 = FunctionSpace(mesh, "CG", 10)
(u, p) = U.split(True)
u_ex = Expression(prob['u'], V=VP10)
M = inner((u_ex - u),(u_ex - u))*dx
v_err = assemble(M, mesh=mesh)
p_ex = Expression(prob['p'], V=P10)
Mp = (p_ex - p)*(p_ex - p)*dx
p_err = assemble(Mp, mesh=mesh)
Mdiv = div(u)*div(u)*dx
v_div = assemble(Mdiv, mesh=mesh)
```

Figure A.6: Code for determining error of analytic problem.

Test Case Velocity Error

(a)

Test Case Pressure Error
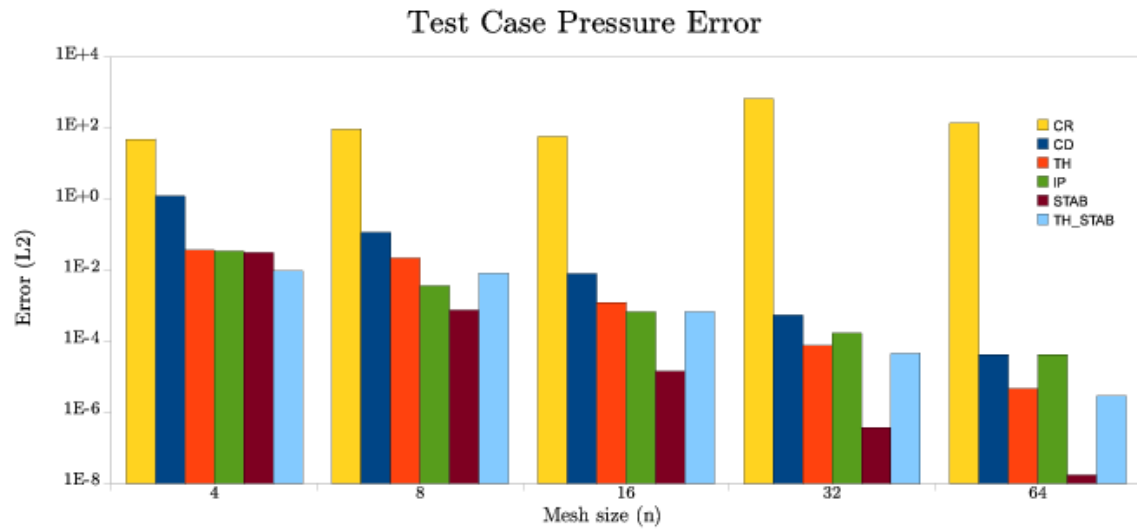
(b)

Figure A.7: Comparison of 4th Order methods on analytic test case. Crouzeix-Raviart on a finer mesh to have equivalent number of DOFs),TH is Taylor-Hood, IP is Iterated Penalty
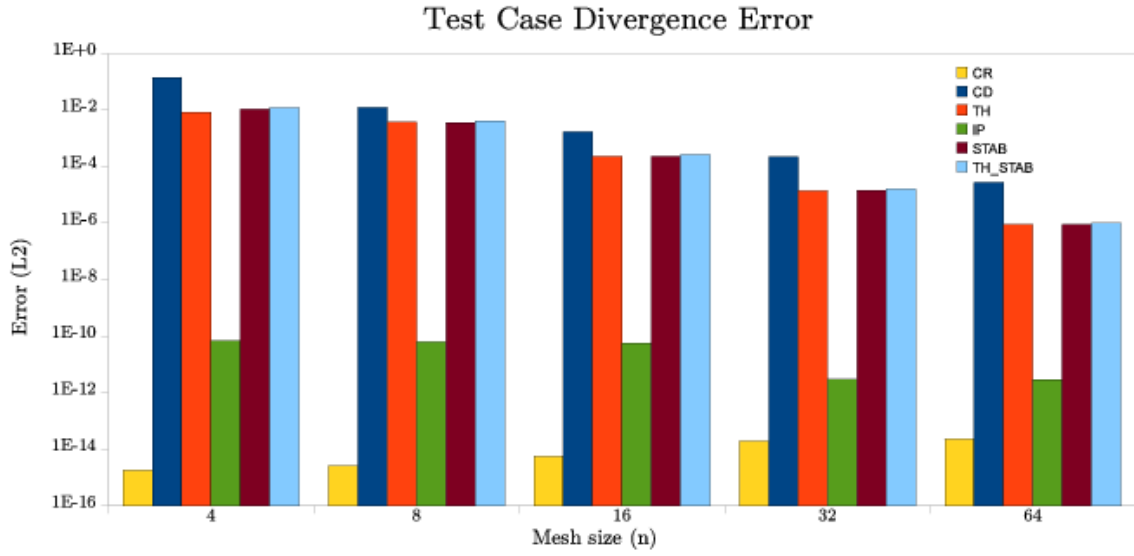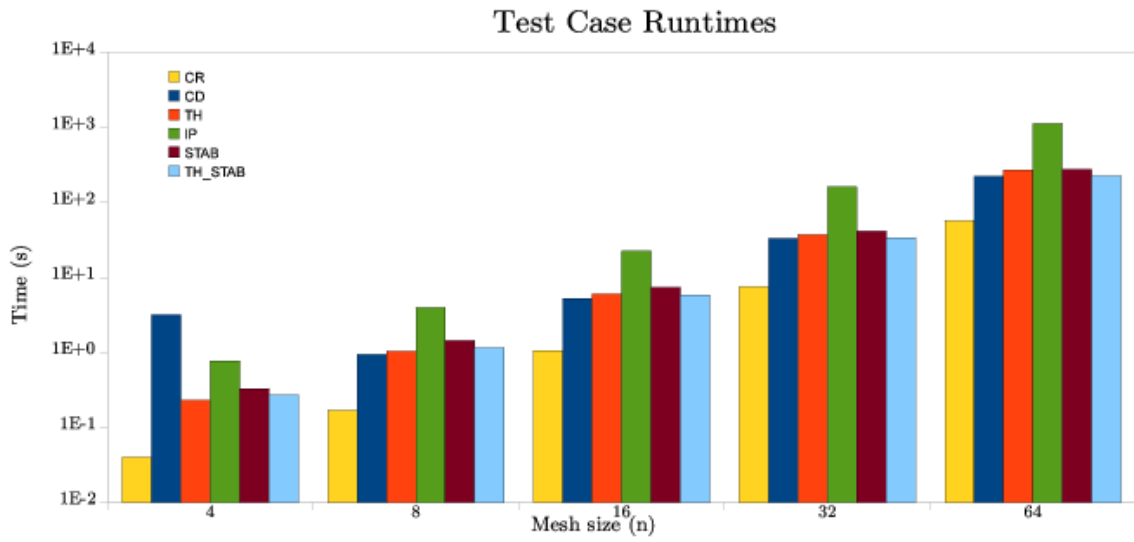
Figure A.8: Comparison of 4th Order methods on analytic test case, as in Figure A.7

APPENDIX B

CODE AND RESULTS FROM GRADE TWO EQUATIONS

```
# Discretization parameters
family = "Lagrange"
shape = "triangle"
vel_order = 2
pres_order = 1

# Element Definitions
VelocityElement = VectorElement(family,shape,vel_order)
PressureElement = FiniteElement(family,shape,pres_order)
MixedElement = VelocityElement + PressureElement
TransportElement = FiniteElement(family,shape,vel_order)

# Test and Trial Functions
(v, q) = TestFunctions(MixedElement)
(u, p) = TrialFunctions(MixedElement)

# Equation Parameters
z = Function(TransportElement)
eta = Constant(shape)
f = Function(VelocityElement)

# Bilinear and Linear Form Definitions
a = (eta*inner(grad(v), grad(u)) + z*v[1]*u[0] - z*v[0]*u[1] - div(v)*p \
  - q*div(u))*dx
L = dot(v, f)*dx
```

Figure B.1: Code for Stokes system in grade two fluid.

```
family = "Lagrange"
shape = "triangle"
order = 4

VelocityElement = VectorElement(family,shape,order)
TransportElement = FiniteElement(family,shape,order)

v = TestFunction(TransportElement)
z = TrialFunction(TransportElement)

u = Function(VelocityElement)
eta = Constant(shape)
alpha = Constant(shape)
f = Function(VelocityElement)

def curl2d(w):
    return D(w[1],0) - D(w[0],1)

a = (eta*v*z + alpha*v*dot(u,grad(z)))*dx
L = v*(alpha*curl2d(f) + eta*curl2d(u))*dx
```

Figure B.2: Form definition of grade two with standard Galerkin transport.

```
a = (eta*v*z + alpha*v*dot(u,grad(z)) + 0.5*v*div(u)*z)*dx
L = v*(alpha*curl2d(f) + eta*curl2d(u))*dx
```

Figure B.3: Form definition of grade two with symmetric transport.

```
delta = Constant(shape)
U = v + delta*dot(u,grad(v))
a = (eta*U*z + alpha*U*dot(u,grad(z)))*dx
L = U*(eta*curl2d(f) + alpha*curl2d(u))*dx
```

Figure B.4: Form definition of grade two with streamline transport.

```
a = (eta*U*z + alpha*U*dot(u,grad(z)) + 0.5*v*div(u)*z)*dx
L = U*(alpha*curl2d(f) + eta*curl2d(u))*dx
```

Figure B.5: Form definition of grade two with symmetric streamline transport.

```
reg_param = Constant(shape)
n = FacetNormal(shape)
h = Function(TransportElement)
e = reg_param*h
a = (e*dot(grad(z),grad(v)) + eta*v*z + alpha*v*dot(u,grad(z)))*dx +
e*v*dot(grad(z),n)*ds
L = v*(alpha*curl2d(f) + eta*curl2d(u))*dx
```

Figure B.6: Form definition of grade two with regularized transport.

```
a_galerkin = (eta*v*z + alpha*v*dot(u,grad(z)))*dx
L_galerkin = v*(alpha*curl2d(f) + eta*curl2d(u))*dx

# SUPG Regularization Terms
res0 = eta*z + alpha*dot(u, grad(z))
res1 = alpha*curl2d(f) + eta*curl2d(u)
supg = tau*alpha*dot(u, grad(v))
a_supg = supg*res0*dx
L_supg = supg*res1*dx

# Bilinear and Linear Form Definitions
a = a_galerkin + a_supg
L = L_galerkin + L_supg
```

Figure B.7: Form definition of grade two with SUPG transport.

```
Grade2Solver grade2;
grade2.parameters["max_iter"] = 15;
grade2.parameters["alpha"] = 0.1;
grade2.parameters["mu"] = 10.0;
grade2.solve(fluid, zero);
fluid.toFile("grade2");
```

Figure B.8: Calling `Grade2Solver` class

Table B.1: Grade two data from lid driven cavity

velocity discretization	transport stabilization	alpha	mu	iterations	residual
Scott-Vogelius	Galerkin	0.1	1.0	6	1.77e-05
Scott-Vogelius	Galerkin	1.0	1.0	5	1.77e-05
Scott-Vogelius	Galerkin	10.0	1.0	5	1.77e-05
Scott-Vogelius	Galerkin	100.0	1.0	7	1.77e-05
Scott-Vogelius	Galerkin	0.1	0.1	9	2.51e-07
Scott-Vogelius	Galerkin	1.0	0.1	8	2.47e-07
Scott-Vogelius	Galerkin	10.0	0.1	10	2.47e-07
Scott-Vogelius	Galerkin	100.0	0.1	10	2.46e-07
Scott-Vogelius	symmetric	0.1	1.0	6	1.77e-05
Scott-Vogelius	symmetric	1.0	1.0	5	1.77e-05
Scott-Vogelius	symmetric	10.0	1.0	5	1.77e-05
Scott-Vogelius	symmetric	100.0	1.0	5	1.77e-05
Scott-Vogelius	symmetric	0.1	0.1	9	2.51e-07
Scott-Vogelius	symmetric	1.0	0.1	8	2.47e-07
Scott-Vogelius	symmetric	10.0	0.1	10	2.47e-07
Scott-Vogelius	symmetric	100.0	0.1	10	2.46e-07
Scott-Vogelius	streamline	0.1	1.0	4	1.77e-05
Scott-Vogelius	streamline	1.0	1.0	5	1.77e-05
Scott-Vogelius	streamline	10.0	1.0	8	1.78e-05
Scott-Vogelius	streamline	100.0	1.0	10	3.43e-05
Scott-Vogelius	streamline	0.1	0.1	9	2.51e-07
Scott-Vogelius	streamline	1.0	0.1	10	1.06e-05
Scott-Vogelius	streamline	10.0	0.1	10	1.78e-03
Scott-Vogelius	streamline	100.0	0.1	10	2.15e-03
Scott-Vogelius	symmetric streamline	0.1	1.0	6	1.77e-05
Scott-Vogelius	symmetric streamline	1.0	1.0	5	1.77e-05
Scott-Vogelius	symmetric streamline	10.0	1.0	5	1.77e-05
Scott-Vogelius	symmetric streamline	100.0	1.0	5	1.77e-05
Scott-Vogelius	symmetric streamline	0.1	0.1	9	2.51e-07
Scott-Vogelius	symmetric streamline	1.0	0.1	8	2.47e-07
Scott-Vogelius	symmetric streamline	10.0	0.1	7	2.46e-07
Scott-Vogelius	symmetric streamline	100.0	0.1	7	2.46e-07

Table B.2: Grade two data from lid driven cavity, continued

velocity discretization	transport stabilization	alpha	mu	iterations	residual
Scott-Vogelius	regularized	0.1	1.0	5	1.77e-05
Scott-Vogelius	regularized	1.0	1.0	5	1.77e-05
Scott-Vogelius	regularized	10.0	1.0	5	1.77e-05
Scott-Vogelius	regularized	100.0	1.0	4	1.77e-05
Scott-Vogelius	regularized	0.1	0.1	5	2.44e-07
Scott-Vogelius	regularized	1.0	0.1	5	2.44e-07
Scott-Vogelius	regularized	10.0	0.1	5	2.44e-07
Scott-Vogelius	regularized	100.0	0.1	4	2.44e-07
Taylor-Hood	SUPG	0.1	1.0	6	3.95e-13
Taylor-Hood	SUPG	1.0	1.0	5	1.19e-11
Taylor-Hood	SUPG	10.0	1.0	5	7.97e-13
Taylor-Hood	SUPG	100.0	1.0	5	1.11e-13
Taylor-Hood	SUPG	0.1	0.1	9	9.06e-12
Taylor-Hood	SUPG	1.0	0.1	8	5.10e-12
Taylor-Hood	SUPG	10.0	0.1	7	5.17e-12
Taylor-Hood	SUPG	100.0	0.1	7	4.36e-12

APPENDIX C

CODE AND RESULTS FROM OLDROYD-B EQUATIONS

```python
# Discretization parameters
family = "Lagrange"; dfamily = "Discontinuous Lagrange";
shape = "triangle"; order = 2

# Element Definitions
dim = 2
stress = VectorElement(dfamily, shape, order-1, 3)
velocity = VectorElement(family, shape, order)
pressure = FiniteElement(family, shape, order-1)
mixed = MixedElement([velocity,pressure,stress])

# Test and Trial function definitions
v, q, phi_vec = TestFunctions(mixed)
phi = as_matrix([[phi_vec[0], phi_vec[1]],[phi_vec[1], phi_vec[2]]])
dsol = TrialFunction(mixed)

# Current Solution
sol = Function(mixed)
u, p, sigma_vec = split(sol)
sigma = as_matrix([[sigma_vec[0], sigma_vec[1]],
                   [sigma_vec[1], sigma_vec[2]]])

# Previous solution
sol0 = Function(mixed)
u0, p0, sigma0_vec = split(sol0)
sigma0 = as_matrix([[sigma0_vec[0], sigma0_vec[1]],
                    [sigma0_vec[1], sigma0_vec[2]]])

# Equation parameters
dt = Constant(shape)     # time step
lam = Constant(shape)    # relaxation time
eta = Constant(shape)    # viscosity
eta_e = Constant(shape) # effective viscosity
alpha = eta_e # DEVSS change of variable parameter. Just use eta_e
f = Function(velocity)  # Body forces

# The projected rate-of-strain
D_proj_vec = Function(stress)
D_proj = as_matrix([[D_proj_vec[0], D_proj_vec[1]],
                    [D_proj_vec[1], D_proj_vec[2]]])
```

Figure C.1: Boilerplate code for setting up model

```
# Some useful functions
def tgrad (w):
    """Returns transpose gradient"""
    return transpose(grad(w))

def D (w):
    """Returns the rate of strain tensor"""
    return (grad(w) + tgrad(w))/2

# Conservation equations (Stokes-like system)
lhs_stokes = ( 2*eta*inner(grad(v), D(u)) - inner(p, div(v)) \
                + inner(q, div(u)) + inner(grad(v), sigma) )*dx
rhs_stokes = inner(v,f)*dx

# Constitutive equations: Oldroyd-B
sigma_uc = 1.0/dt*sigma + dot(u, grad(sigma)) \
           - dot(grad(u), sigma) - dot(sigma, tgrad(u))

lhs_con = inner(phi, lam*sigma_uc + sigma - 2*eta_e*D(u))*dx
rhs_con = (lam/dt * inner(phi, sigma0) )*dx
```

Figure C.2: Boilerplate code for setting up model, continued

```
from OB_THD_Base import *

# Full bilinear form and residual
L = lhs_con + lhs_stokes - rhs_con - rhs_stokes
a = derivative(L, sol, dsol)
```

Figure C.3: Code for specifying Oldroyd MIX system

```
from OB_THD_Base import *

lhs_con += inner(alpha*dot(u0, grad(phi)), dot(u, grad(sigma)))*dx

# Full bilinear form and residual
L = lhs_con + lhs_stokes - rhs_con - rhs_stokes
a = derivative(L, sol, dsol)
```

Figure C.4: Code for specifying Oldroyd MIX system with SU stabilization

```
from OB_THD_Base import *

lhs_con += inner(alpha*dot(u0, grad(phi)), sigma_uc + sigma -
2*eta*D(u))*dx

L = lhs_con + lhs_stokes - rhs_con - rhs_stokes
a = derivative(L, sol, dsol)
```

Figure C.5: Code for specifying Oldroyd MIX system with SUPG stabilization

```
from OB_THD_Base import *

lhs_stokes += 2*eta*inner(D(v), D(u))*dx
rhs_stokes += 2*(eta_e + eta)*inner(D(v), D_proj)*dx

# Full bilinear form and residual
L = lhs_con + lhs_stokes - rhs_con - rhs_stokes
a = derivative(L, sol, dsol)
```

Figure C.6: Code for specifying UCM DEVSS system

```
OldroydBSolver oldroydb;
// polymer viscosity
oldroydb.parameters["eta"] = 100.0;
// effective viscosity, UCM sets this to 0
oldroydb.parameters["eta_e"] = 0.1;
// relaxation time parameter
oldroydb.parameters["lam"] = 1.0;
// initial continuation step
oldroydb.parameters["d_lam"] = 1.0e-2;
//Possibilities: MIX, DEVSS
oldroydb.parameters["discretization"] = "MIX";
//Possibilities: None, SU, SUPG
oldroydb.parameters["stabilization"] = "SU";

NewtonSolver& ns = oldroydb.newton_solver();
ns.parameters["maximum_iterations"] = 50;
oldroydb.solve(fluid, zero);
fluid.toFile("oldroydb", true, true);
```

Figure C.7: Code for calling Oldroyd-B class

```cpp
class NoSlipBC : public Expression
{
public:
  NoSlipBC() : Expression(2) {}
  void eval (double* values, const std::vector<double>& input) const
  {
      values[0] = 0;
      values[1] = 0;
  }
};

// Boundary Condition for inflow
class Inflow : public Expression
{
public:
  Inflow() : Expression(2) {}
  void eval (double* values, const std::vector<double>& input) const
  {
    double y = input[1];
    values[0] = 1.0/64.0 * 0.01 * (4.0 - y) * (4.0 + y);
    values[1] = 0.0;
  }
};

class Outflow : public Expression
{
 public:
  Outflow() : Expression(2) {}
  void eval (double* values, const std::vector<double>& input) const
  {
    double y = input[1];
    values[0] = .01 * (1.0 - y) * (1.0 + y);
    values[1] = 0.0;
  }
};
```

Figure C.8: Code for defining planar contraction problem

```cpp
// Boundary Subdomain
class Inlet : public SubDomain
{
 public:
  Inlet () : SubDomain () {}
  virtual bool inside (const double* pt, bool on_boundary) const
  {
    return on_boundary && pt[0] < DOLFIN_SQRT_EPS;
  }
};

class Outlet : public SubDomain
{
 public:
  Outlet () : SubDomain () {}
  virtual bool inside (const double* pt, bool on_boundary) const
  {
    return on_boundary && pt[0] > 6.0 - DOLFIN_SQRT_EPS;
  }
};

class TopWall : public SubDomain
{
public:
  TopWall() : SubDomain () {}
  virtual bool inside (const double* pt, bool on_boundary) const
  {
    double x = pt[0], y = pt[1];
    return on_boundary &&
      ( ( x > 3.0 - DOLFIN_SQRT_EPS && y > 1.0 - DOLFIN_SQRT_EPS ) ||
        ( x < 3.0 + DOLFIN_SQRT_EPS && y > 4.0 - DOLFIN_SQRT_EPS));
  }
};

class SymmetryLine: public SubDomain
{
public:
  SymmetryLine() : SubDomain () {}
  virtual bool inside (const double* pt, bool on_boundary) const
  {
    return on_boundary && pt[1] < DOLFIN_SQRT_EPS;
  }
};
```

Figure C.9: Code continued for defining planar contraction problem

Table C.1: UCM method data from lid driven cavity

discretization	stabilization	eta	lam	Newton iterations
MIX	SU	1.0	1.0	18
MIX	SUPG	1.0	1.0	X
MIX	None	1.0	1.0	18
DEVSS	SU	1.0	1.0	20
DEVSS	SUPG	1.0	1.0	X
DEVSS	None	1.0	1.0	20
MIX	SU	100.0	1.0	18
MIX	SUPG	100.0	1.0	X
MIX	None	100.0	1.0	18
DEVSS	SU	100.0	1.0	20
DEVSS	SUPG	100.0	1.0	X
DEVSS	None	100.0	1.0	20
MIX	SU	100.0	100.0	44
MIX	SUPG	100.0	100.0	X
MIX	None	100.0	100.0	44
DEVSS	SU	100.0	100.0	46
DEVSS	SUPG	100.0	100.0	X
DEVSS	None	100.0	100.0	46

Table C.2: Oldroyd-B method data from lid driven cavity

discretization	stabilization	eta	eta_e	lam	Newton iterations
MIX	SU	1.0	0.1	1.0	X
MIX	SUPG	1.0	0.1	1.0	X
MIX	None	1.0	0.1	1.0	X
DEVSS	SU	1.0	0.1	1.0	X
DEVSS	SUPG	1.0	0.1	1.0	X
DEVSS	None	1.0	0.1	1.0	X
MIX	SU	100.0	0.1	1.0	21
MIX	SUPG	100.0	0.1	1.0	22
MIX	None	100.0	0.1	1.0	X
DEVSS	SU	100.0	0.1	1.0	X
DEVSS	SUPG	100.0	0.1	1.0	X
DEVSS	None	100.0	0.1	1.0	X
MIX	SU	100.0	0.1	0.1	10
MIX	SUPG	100.0	0.1	0.1	X
MIX	None	100.0	0.1	0.1	7
DEVSS	SU	100.0	0.1	0.1	X
DEVSS	SUPG	100.0	0.1	0.1	X
DEVSS	None	100.0	0.1	0.1	1753
MIX	SU	100.0	0.1	100.0	X
MIX	SUPG	100.0	0.1	100.0	X
MIX	None	100.0	0.1	100.0	X
DEVSS	SU	100.0	0.1	100.0	X
DEVSS	SUPG	100.0	0.1	100.0	X
DEVSS	None	100.0	0.1	100.0	X

Table C.3: PTT method data from lid driven cavity

discretization	stabilization	eta	eta_e	lam	Newton iterations
MIX	SU	1.0	0.0	1.0	18
MIX	SUPG	1.0	0.0	1.0	X
MIX	None	1.0	0.0	1.0	18
DEVSS	SU	1.0	0.0	1.0	20
DEVSS	SUPG	1.0	0.0	1.0	X
DEVSS	None	1.0	0.0	1.0	20
MIX	SU	100.0	0.0	1.0	18
MIX	SUPG	100.0	0.0	1.0	X
MIX	None	100.0	0.0	1.0	18
DEVSS	SU	100.0	0.0	1.0	20
DEVSS	SUPG	100.0	0.0	1.0	X
DEVSS	None	100.0	0.0	1.0	20
MIX	SU	100.0	0.0	100.0	44
MIX	SUPG	100.0	0.0	100.0	X
MIX	None	100.0	0.0	100.0	44
DEVSS	SU	100.0	0.0	100.0	46
DEVSS	SUPG	100.0	0.0	100.0	X
DEVSS	None	100.0	0.0	100.0	46

Table C.4: PTT method data from lid driven cavity, continued

discretization	stabilization	eta	eta_e	lam	Newton iterations
MIX	SU	1.0	0.1	1.0	458
MIX	SUPG	1.0	0.1	1.0	X
MIX	None	1.0	0.1	1.0	X
DEVSS	SU	1.0	0.1	1.0	X
DEVSS	SUPG	1.0	0.1	1.0	X
DEVSS	None	1.0	0.1	1.0	X
MIX	SU	100.0	0.1	1.0	X
MIX	SUPG	100.0	0.1	1.0	X
MIX	None	100.0	0.1	1.0	X
DEVSS	SU	100.0	0.1	1.0	X
DEVSS	SUPG	100.0	0.1	1.0	X
DEVSS	None	100.0	0.1	1.0	X
MIX	SU	100.0	0.1	0.1	9
MIX	SUPG	100.0	0.1	0.1	X
MIX	None	100.0	0.1	0.1	7
DEVSS	SU	100.0	0.1	0.1	X
DEVSS	SUPG	100.0	0.1	0.1	X
DEVSS	None	100.0	0.1	0.1	X
MIX	SU	100.0	0.1	100.0	X
MIX	SUPG	100.0	0.1	100.0	X
MIX	None	100.0	0.1	100.0	X
DEVSS	SU	100.0	0.1	100.0	X
DEVSS	SUPG	100.0	0.1	100.0	X
DEVSS	None	100.0	0.1	100.0	X

Table C.5: UCM method data from planar contraction

discretization	stabilization	eta	lam	Newton iterations
MIX	SU	1.0	1.0	44
MIX	SUPG	1.0	1.0	X
MIX	None	1.0	1.0	44
DEVSS	SU	1.0	1.0	46
DEVSS	SUPG	1.0	1.0	X
DEVSS	None	1.0	1.0	46
MIX	SU	100.0	1.0	44
MIX	SUPG	100.0	1.0	X
MIX	None	100.0	1.0	44
DEVSS	SU	100.0	1.0	46
DEVSS	SUPG	100.0	1.0	X
DEVSS	None	100.0	1.0	46
MIX	SU	100.0	100.0	71
MIX	SUPG	100.0	100.0	X
MIX	None	100.0	100.0	71
DEVSS	SU	100.0	100.0	73
DEVSS	SUPG	100.0	100.0	X
DEVSS	None	100.0	100.0	73

Table C.6: Oldroyd-B method data from planar contraction

discretization	stabilization	eta	eta_e	lam	Newton iterations
MIX	SU	1.0	0.1	1.0	46
MIX	SUPG	1.0	0.1	1.0	X
MIX	None	1.0	0.1	1.0	44
DEVSS	SU	1.0	0.1	1.0	68
DEVSS	SUPG	1.0	0.1	1.0	58
DEVSS	None	1.0	0.1	1.0	44
MIX	SU	100.0	0.1	1.0	46
MIX	SUPG	100.0	0.1	1.0	X
MIX	None	100.0	0.1	1.0	44
DEVSS	SU	100.0	0.1	1.0	66
DEVSS	SUPG	100.0	0.1	1.0	X
DEVSS	None	100.0	0.1	1.0	52
MIX	SU	100.0	0.1	0.1	33
MIX	SUPG	100.0	0.1	0.1	X
MIX	None	100.0	0.1	0.1	31
DEVSS	SU	100.0	0.1	0.1	53
DEVSS	SUPG	100.0	0.1	0.1	X
DEVSS	None	100.0	0.1	0.1	33
MIX	SU	100.0	0.1	100.0	73
MIX	SUPG	100.0	0.1	100.0	X
MIX	None	100.0	0.1	100.0	73
DEVSS	SU	100.0	0.1	100.0	93
DEVSS	SUPG	100.0	0.1	100.0	X
DEVSS	None	100.0	0.1	100.0	136

Table C.7: PTT method data from planar contraction

discretization	stabilization	eta	eta_e	lam	Newton iterations
MIX	SU	1.0	0.0	1.0	18
MIX	SUPG	1.0	0.0	1.0	X
MIX	None	1.0	0.0	1.0	18
DEVSS	SU	1.0	0.0	1.0	20
DEVSS	SUPG	1.0	0.0	1.0	X
DEVSS	None	1.0	0.0	1.0	20
MIX	SU	100.0	0.0	1.0	44
MIX	SUPG	100.0	0.0	1.0	X
MIX	None	100.0	0.0	1.0	44
DEVSS	SU	100.0	0.0	1.0	46
DEVSS	SUPG	100.0	0.0	1.0	X
DEVSS	None	100.0	0.0	1.0	46
MIX	SU	100.0	0.0	100.0	71
MIX	SUPG	100.0	0.0	100.0	X
MIX	None	100.0	0.0	100.0	71
DEVSS	SU	100.0	0.0	100.0	73
DEVSS	SUPG	100.0	0.0	100.0	X
DEVSS	None	100.0	0.0	100.0	73

Table C.8: PTT method data from planar contraction, continued

MIX	SU	1.0	0.1	1.0	18
MIX	SUPG	1.0	0.1	1.0	X
MIX	None	1.0	0.1	1.0	18
DEVSS	SU	1.0	0.1	1.0	
DEVSS	SUPG	1.0	0.1	1.0	
DEVSS	None	1.0	0.1	1.0	
MIX	SU	100.0	0.1	1.0	46
MIX	SUPG	100.0	0.1	1.0	X
MIX	None	100.0	0.1	1.0	44
DEVSS	SU	100.0	0.1	1.0	X
DEVSS	SUPG	100.0	0.1	1.0	X
DEVSS	None	100.0	0.1	1.0	X
MIX	SU	100.0	0.1	0.1	33
MIX	SUPG	100.0	0.1	0.1	X
MIX	None	100.0	0.1	0.1	31
DEVSS	SU	100.0	0.1	0.1	X
DEVSS	SUPG	100.0	0.1	0.1	X
DEVSS	None	100.0	0.1	0.1	31
MIX	SU	100.0	0.1	100.0	73
MIX	SUPG	100.0	0.1	100.0	X
MIX	None	100.0	0.1	100.0	71
DEVSS	SU	100.0	0.1	100.0	X
DEVSS	SUPG	100.0	0.1	100.0	X
DEVSS	None	100.0	0.1	100.0	X

REFERENCES

[1] M. Alnæs, H. P. Langtangen, A.Logg, K.-A. Mardal, and O. Skavhaug, *UFC Specification and User Manual*, 2007. URL: `http://www.fenics.org/ufc/`.

[2] M. S. Alnæs and K.-A. Mardal, *SyFi user manual*, 2009. `http://www.fenics.org/syfi/`.

[3] Frank P. T. Baaijens, *Mixed finite element discretizations for viscoelastic flow analysis: a review*, Journal of Non-Newtonian Fluid Mechanics, 79 (1998), pp. 361–385.

[4] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang, *PETSc Web page.* `http://www.mcs.anl.gov/petsc`, 2007.

[5] J. Baranger and D. Sandri, *A formulation of stoke's problem and the linear elasticity equations suggested by the oldroyd model for viscoelastic flow*, Math. Modelling Numer. Anal., 26 (1992).

[6] Daniel J. Bates, Jonathan D. Hauenstein, Charles W. Wampler, , and A. J. Sommese, *Bertini: Software for numerical algebraic geometry*, 2010.

[7] A. Beris, Robert C. Armstrong, and Robert A. Brown, *Perturbation theory for viscoelastic eccentric rotating cylinders fluids between*, Journal of Non-Newtonian Fluid Mechanics, 13 (1983), pp. 109– 148.

[8] Paolo Bientinesi, *Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms*, PhD thesis, Department of Computer Sciences, University of Texas at Austin, August 2006.

[9] Paolo Bientinesi, John a. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert a. Van De Geijn, *The science of deriving dense linear algebra algorithms*, ACM Transactions on Mathematical Software, 31 (2005), pp. 1–26.

[10] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn, *Representing linear algebra algorithms in code: the FLAME application program interfaces*, ACM Trans. Math. Softw., 31 (2005), pp. 27–59.

[11] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, SIAM, Philadelphia, PA, 1997.

[12] Daniele Boffi, *Three-dimensional finite element methods for the Stokes problem*, SIAM J. Numer. Anal., 34 (1997), pp. 664–670.

[13] SUSANNE C. BRENNER AND L. RIDGWAY SCOTT, *The mathematical theory of finite element methods*, vol. 15 of Texts in Applied Mathematics, Springer, New York, third ed., 2008.

[14] FRANCO BREZZI AND MICHEL FORTIN, *Mixed and hybrid finite element methods*, vol. 15 of Springer Series in Computational Mathematics, Springer-Verlag, New York, 1991.

[15] ALEXANDER N. BROOKS AND THOMAS J. R. HUGHES, *Streamline upwind/petrov-galerkin formulations for convection dominated flows with particular emphasis on the incompressible navier-stokes equations*, Computer Methods in Applied Mechanics and Engineering, 32 (1982), pp. 199–259.

[16] R. CAMASSA AND D. D. HOLM, *Thermodynamics, stability, and boundedness of fluids of complexity two and fluids of second grade*, Phys. Rev. Lett., 71 (1993), pp. 1661 – 1664.

[17] S. CHEN, C. FOIAS, D. D. HOLM, E. OLSON, E. TITI, AND S. WYNNE, *The camassa-holm equations as a closure model for turbulent and pipe flow*, Phys. Rev. Lett., 81 (1993), pp. 5338 – 5341.

[18] P.G. CIARLET, *Lectures on the finite element method*, Lectures on Mathematics and Physics, Tata Institute of Fundamental Research, Bombay, 49 (1975).

[19] D. CIORANESCU AND E. H. OUAZAR, *Existence and uniqueness for fluids of second grade*, in Nonlinear Partial Dierential Equations and their Applications, College de France Seminar, Pitman: Boston, 1984, pp. 178–197.

[20] COLEMAN AND NOLL, *Foundations of linear viscoelasticity*, Arch. Rat. Mech. Anal., 6 (1961), p. 355.

[21] R. COURANT, *Variational methods for the solution of problems of equilibrium and vibrations*, Bull. Amer. Math. Soc., (1943), pp. 1–23.

[22] M. J. CROCHET AND M. BEZY, *Numerical solution for the flow of viscoelastic fluids*, Journal of Non-Newtonian Fluid Mechanics, 5 (1979), pp. 201–218.

[23] M. CROUZEIX AND P A RAVIART, *Conforming and non-conforming finite element methods for solving the stationary Stokes equations*, R.A.I.R.O Anal. Numer., 7 (1973), pp. 33–76.

[24] T.A. DAVIS, *Algorithm 832: UMFPACK V4. 3an unsymmetric-pattern multifrontal method*, ACM Transactions on Mathematical Software, 30 (2004), pp. 196–199.

[25] JEAN DONEA AND ANTONIO HUERTA, *Finite Element Methods for Flow Problems*, Wilet, 2003.

[26] J.E. Dunn and K.R. Rajagopal, *Fluids of differential type: Critical review and thermodynamic analysis*, Internat. J. Engrg. Sci., 33 (1995), pp. 689–729.

[27] J. E. Dunn and R. L. Fosdick, *Thermodynamics, stability, and boundedness of uids of complexity two and fluids of second grade*, Archives for Rational Mechanics and Analysis, 56 (1974), pp. 191–252.

[28] T. Dupont, J. Hoffman, C. Johnson, R. C. Kirby, M. G. Larson, A. Logg, and L. R. Scott, *The FEniCS project*, Tech. Report 11, Chalmers Finite Element Center Preprint Series, 2003.

[29] Howard Elman, David Silvester, and Andy Wathen, *Finite Elements and Fast Iterative Solvers*, Oxford Science Publications, 2005.

[30] Charbel Farhat and Francois-Xavier Roux, *A method of finite element tearing and interconnecting and its parallel solution algorithm*, International Journal for Numerical Methods in Engineering, 32 (1991), pp. 1205–1227.

[31] Michel Fortin and André Fortin, *A new approach for the FEM simulation of viscoelastic flows*, Journal of Non-Newtonian Fluid Mechanics, 32 (1989), pp. 295–310.

[32] M Fortin and R Pierre, *the convergence of the mixed method of Crochet and Marchal for viscoelastic flows, Comput*, Methods Appl. Mech. Engrg., 73 (1989), pp. 341–350.

[33] Matteo Frigo and Steven G. Johnson, *The design and implementation of FFTW3*, in Proceedings of the IEEE, vol. 93, 2005, pp. 216–231.

[34] Jens Gerlach, Peter Gottschling, and Uwe Der, *A generic c++ framework for parallel mesh based scientific applications*, in 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments, San Francisco, 2001.

[35] Hanswalter Giesekus, *Die elastizität von flüssigkeiten*, Rheologica Acta, 5 (1966), pp. 29–35.

[36] Vivett Girault and L. Ridgway Scott, *Finite element discretizations of a two-dimensional grade-two fluid model*, Mathematical Modelling And Numerical Analysis, 35 (2001), pp. 1007–1053.

[37] V. Girault and L. Ridgway Scott, *Upwind discretization of a steady grade-two fluid model in two-dimensions*, Studies in Math. and Its Appl., 31 (2002), pp. 393–414.

[38] K. Goto, *Gotoblas*, 2010.

[39] Robert Guénette and Michel Fortin, *A new mixed finite element method for computing viscoelastic flows*, Journal of Non-Newtonian Fluid Mechanics, 60 (1995), pp. 27–52.

[40] JOHN A. GUNNELS, FRED G. GUSTAVSON, GREG M. HENRY, AND ROBERT A. VAN DE GEIJN, *FLAME: Formal linear algebra methods environment*, Transactions on Mathematical Software, 27 (2001), pp. 422–455.

[41] T. J. R. HUGHES, *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Prentice-Hall, 1987.

[42] MUTSUTO KAWAHARA AND NORIO TAKEUCHI, *Mixed finite element method for analysis of viscoelastic fluid flow*, Computers and Fluids, 5 (1977), pp. 33–45.

[43] R. C. KIRBY, *FIAT: A new paradigm for computing finite element basis functions*, ACM Trans. Math. Software, 30 (2004), pp. 502–516.

[44] ROBERT C. KIRBY, MATTHEW G. KNEPLEY, ANDERS LOGG, AND L. RIDGWAY SCOTT, *Optimizing the evaluation of finite element matrices*, SIAM J. Sci. Comput., 27 (2005), pp. 741–758.

[45] R. C. KIRBY, M. G. KNEPLEY, AND L. R. SCOTT, *Evaluation of the action of finite element operators*, Tech. Report TR–2004–07, University of Chicago, Department of Computer Science, 2004.

[46] ROBERT C. KIRBY AND ANDERS LOGG, *A compiler for variational forms*, ACM Transactions on Mathematical Software, 32 (2006), pp. 417–444.

[47] ROBERT C. KIRBY, ANDERS LOGG, L. RIDGWAY SCOTT, AND ANDY R. TERREL, *Topological optimization of the evaluation of finite element matrices*, SIAM J. Sci. Comput., 28 (2006), pp. 224–240.

[48] ROBERT C. KIRBY AND L. RIDGWAY SCOTT, *Geometric optimization of the evaluation of finite element matrices*, to appear in SIAM J. Sci. Comput., (2007).

[49] MATTHEW G. KNEPLEY AND DMITRY A. KARPEEV, *Mesh algorithms for PDE with Sieve I: Mesh distribution*, Technical Report ANL/MCS-P1455-0907, Argonne National Laboratory, February 2007.

[50] ——, *Sieve concepts and interfaces*, Technical Report ANL/MCS-P????-????, Argonne National Laboratory, February 2007.

[51] ——, *Mesh algorithms for PDE with Sieve I: Mesh distribution*, Scientific Programming, 17 (2009), pp. 215–230.

[52] D. A. KNOLL AND D. E. KEYES, *Jacobian-free NewtonKrylov methods: a survey of approaches and applications*, Journal of Computational Physics, 193 (2004), pp. 357–397.

[53] ANDERS LOGG, *An overview of the fenics project*, in 21st Nordic Seminar on Computational Mechanics, 2008. submitted September 2008.

[54] C-S. Man, *Nonsteady channel flow of ice as a modified second-order fluid with power-law viscosity*, Archives for Rational Mechanics and Analysis, 119 (1992), pp. 35–57.

[55] J. M. Marchal and M. J. Crochet, *Hermitian finite elements for calculating viscoelastic flow*, Journal of Non-Newtonian Fluid Mechanics, 20 (1986), pp. 187–207.

[56] ———, *A new mixed finite element for calculating viscoelastic flow*, Journal of Non-Newtonian Fluid Mechanics, 26 (1987), pp. 77–114.

[57] J. G. Oldroyd, *On the formulation of rheological equations of state*, Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences, 200 (1950), pp. 523–541.

[58] K. B. Ølgaard and G. N. Wells, *Optimisations for quadrature representations of finite element tensors through automated code generation*, ACM Transactions on Mathematical Software, 37 (2010).

[59] M. G. N. Perera and K. Walters, *Long range memory effects in flows involving abrupt changes in geometry : Part 2: the expansion/contraction/expansion problem*, Journal of Non-Newtonian Fluid Mechanics, 2 (1977), pp. 191 – 204.

[60] M. Puschel and J.M.F. Moura, *The Algebraic Structure in Signal Processing: Time and Space*, in 2006 IEEE International Conference on Acoustics Speed and Signal Processing Proceedings, IEEE, 2006, pp. V–997–V–1000.

[61] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo, *SPIRAL: Code generation for DSP transforms*, Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation", 93 (2005), pp. 232–275.

[62] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. van de Geijn, Field G. Van Zee, and Ernie Chan, *Programming matrix algorithms-by-blocks for thread-level parallelism*, ACM Transactions on Mathematical Software, 36 (2009), pp. 14:1–14:26.

[63] R. S. Rivlin and J. L. Ericksen, *Stress-deformation relations for isotropic materials*, Rat. Mech. Anal., 3 (1955), pp. 323–702.

[64] L. Ridgway Scott and Micheal Vogelius, *Conforming finite element methods for incompressible and nearly incompressible continua*, in Large Scale Computations in Fluid Mechanics, B. E. Engquist, *et al.*, eds., vol. 22 (Part 2), Providence: AMS, 1985, pp. 221–244.

[65] ———, *Norm estimates for a maximal right inverse of the divergence operator in spaces of piecewise polynomials*, M^2AN (formerly R.A.I.R.O. Analyse Numérique), 19 (1985), pp. 111–143.

[66] A. J. Sommese and C. W. Wampler, *Numerical Solution of Polynomial Systems Arising in Engineering and Science*, World Scientific, Singapore, 2005.

[67] G. Strang and G. J. Fix, *An Analysis of the Finite Element Method*, Prentice-Hall, Englewood Cliffs, 1973.

[68] R. I. Tanner, *Plane creeping flows of incompressible second-order fluids*, Physics of Fluids, 9 (1966), pp. 1246–1247.

[69] C. Taylor and P. Hood, *A numerical solution of the Navier-Stokes equations using the finite element technique*, Internat. J. Comput. & Fluids, 1 (1973), pp. 73–100.

[70] Nhan Phan Thien and Roger I. Tanner, *A new constitutive equation derived from network theory*, Journal of Non-Newtonian Fluid Mechanics, 2 (1977), pp. 353–365.

[71] Field G. Van Zee, Paolo Bientinesi, Tze Meng Low, and Robert A. van de Geijn, *Scalable parallelization of FLAME code via the workqueuing model*, ACM Trans. Math. Softw., 34 (2008), pp. 1–29.

[72] Jan Verschelde, *Algorithm 795: Phcpack: A general-purpose solver for polynomial systems by homotopy continuation*, ACM Transactions on Mathematical Software, 25 (1999), pp. 251–276.

[73] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra, *Automated empirical optimization of software and the ATLAS project*, Parallel Computing, 27 (2001), pp. 3–35. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[74] Michael M. Wolf and Michael T. Heath, *Combinatorial optimization of matrix-vector multiplicaion in finite element assembly*, SIAM J. Sci. Comput., 31 (2009), p. 2960.

[75] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu, *The Finite Element Method — Its Basis and Fundamentals, 6th edition*, Elsevier, 2005, first published in 1967.