AN ABSTRACT OF THE DISSERTATION OF

Kenneth M. Kennedy for the degree of Doctor of Philosophy in Mathematics presented on May 30, 2017.

Title: Model Adaptivity and Numerical Solutions Using Sensitivity Analysis

Abstract approved: _____

Malgorzata Peszynska

In this work we consider the dependence of solutions to a partial differential equations system on its data. The problem of interest is a coupled model of nonlinear flow and transport in porous media, with applications, e.g. to environmental modeling. The model of flow we consider is known as the non-Darcy model, and its solutions: the velocity, and pressure unknowns, depend on the coefficients of permeability and inertia, and other data such as boundary conditions. In turn, the transport solutions depend on the velocity of the fluid, and on boundary and initial conditions. Furthermore, one can be interested in a particular quantity computable from the flow and transport solutions, and represented by a functional. In this work we evaluate rigorously the sensitivity, i.e., the derivative, of the solutions, or of the quantity of interest, upon the data.

Due to its delicate nature, the sensitivity is evaluated either in a direct way, called Forward Sensitivity, or via an adjoint method, which only uses a variational

form. Our first contribution is that we find a way to find the sensitivity for the coupled flow and transport model without having to solve multiple flow problems. Second, we prove the well-posedness of the flow problem, and set up the numerical approximation using the framework similar to that of expanded mixed finite element methods.

Next, the numerical approximation of the problem leads to a nonlinear system of discrete equations, which is difficult to solve. To aid in solving this system, we propose to take advantage of sensitivity analysis, which is used in a novel way within a homotopy framework. The theoretical results in this thesis are illustrated with numerical simulations. The code, in Python, for the examples is provided.

Model Adaptivity and Numerical Solutions Using Sensitivity Analysis

by

Kenneth M. Kennedy

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented May 30, 2017
Commencement June 2017

Doctor of Philosophy dissertation of Kenneth M. Kennedy presented on May 30, 2017.

APPROVED:

Major Professor, representing Mathematics

Chair of the Department of Mathematics

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Kenneth M. Kennedy, Author

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Malgorzata Peszynska. Without her help and guidance I have no doubt that I would never have completed this research. I would also like to thank my committee members Dr. John Lee, Dr. Mina Ossiander, Dr. Nathan Gibson, and Dr. Lan Xue. Dr. Sourabh Apte also served on my committee before my defense and I would like to thank him as well.

This research was funded in part by the Department of Energy, Office of Science "Modeling, analysis and simulation of preferential flow in porous media" (Principal Investigator: R.E. Showalter, Co-Principal Investigator: Malgorzata Peszynska). Additional funding was provided by the National Science Foundation (NSF). The work on sensitivity analysis was inspired by work on DMS-0511190 "Model adaptivity for porous media" (Principal Investigator: Malgorzata Peszynska). Additional funding was provided by NSF-DMS 1115827 "Hybrid modeling in porous media" (Principal Investigator: Malgorzata Peszynska). Finally, NSF DMS-1522734 "Phase transitions in porous media across multiple scales" (Principal Investigator: Malgorzata Peszynska) provided additional funding.

I would like to thank my family and friends for their love, encouragement, and support. Thank you Alicia and Nicole for your love and support as I finished this chapter of my life so that we can go into the next chapter of our lives. I am grateful to my parents, Mike and Beth Kennedy, for always showing great pride in my work and supporting me in so many ways.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

In this work we formulate several new ideas which are applied to a model problem of flow and coupled transport in porous media. Problems like this are important in groundwater modeling, and in optimization of, e.g., groundwater contamination scenarios. Since most such models do not have solutions in closed form, one resorts to computational algorithms which approximate their solutions.

As computational abilities have increased through advances in computer hardware, the desire to solve ever more complicated models has increased as well. More accurate models of physical phenomena allow scientists and engineers to make better predictions. Since computational power is limited, it is desirable to find ways to decrease the computational cost of solving a system, but without sacrificing the accuracy.

One way of decreasing computational cost is to use the simplest model that captures the necessary details of the system. In particular, one traditional way has been to use linearizations of nonlinear models. Another direction is to test how much accuracy is needed in order to determine a particular quantity of interest. Finally, one other way of decreasing cost is finding more efficient algorithms to find the solution to the model. In this work we explore several methods in these directions. In particular, we consider a nonlinear flow model known as the non-Darcy model which includes additional inertia terms added to the linear Darcy

flow model. The nonlinear model is needed when the flow rates are high, and it models additional resistance of the fluid to the medium.

Before choosing a model, and implementing its numerical approximation, it is important to know that the models of interest are well-posed. A problem is well-posed if it has a unique solution that is continuously dependent on its parameters. Well-posedness is more easily proved for linear problems, and is generally not immediate for nonlinear problems. Typically, increasing the complexity of the problem increases the difficulty of proving that the problem is well-posed. One of our contributions in this work is a proof that the nonlinear flow model called the non-Darcy model is well-posed; this is demonstrated using recently published analysis results. To our knowledge, this is the first proof of the well-posedness of the non-Darcy model in this framework.

Next, we consider the dependence of the flow solutions upon the parameters representing the physical model. For instance, in fluid flow in porous media a resistance term combing the inverse permeabilities as well as inertia terms appears. The resistance term requires some data which are obtained empirically, and one is interested in knowing how the solution to a model varies with changes in the parameters. In this work, sensitivity analysis is explored with that variation in mind.

Sensitivity analysis has many different uses and contexts. In this work, we discuss sensitivity of a model as finding the derivative of the solution to a numerical model with respect to the model parameters. Those derivatives show how the solution varies as the parameters change. That information can be used, e.g, for

model reduction, and for statistical analyses, as well as for informing how accurate measurements need to be for the model to give accurate results.

The mathematical problem of finding the sensitivities, i.e., the derivatives of a solution with respect to the model parameters, can be quite delicate, since it requires more regularity from the solutions than the theory usually guarantees. In addition, the problem of finding sensitivities can be quite complex if the number, or the dimension, of parameters is large. Therefore, different avenues for finding sensitivities have been explored, including forward and adjoint approaches. These have been primarily defined for scalar equations; see [44, 41, 15, 46, 42, 61, 47, 16, 45, 43]. In this work we extend these approaches to a coupled system; see Chapter 3.

Next, we apply our sensitivity methods to find the solution to a nonlinear model. In general, these may be difficult to find, especially for stationary problems. Once the existence and uniqueness of a solution are verified, a method for finding or approximating that solution must be found. Typically, one finds a series of linear problems that may be solved instead of the nonlinear model. A classical example is Newton's method, which uses local linear approximations to the problem to find a root. Another class of methods for finding the solution to a nonlinear model are continuation methods. In those methods, a path between an easy to solve problem and a difficult to solve problem is followed. In this work, sensitivity analysis and numerical continuation are combined into a novel set of methods; see Chapter 5.

This work is organized as follows. In Chapter 2, mathematical preliminaries and some nonlinear model problems are developed. Since sensitivity analysis is

concerned with the derivatives, some generalized notions of derivatives will be presented including distributional derivatives and Frechet derivatives. Sobolev spaces are introduced and some basic theorems are presented. A simple algebraic model is developed which is used in later chapters to show, in a simple setting, how some techniques are used. A nonlinear flow model is presented and the velocity from the flow model is used as the velocity in a coupled transport model. The well-posedness of all of the models is presented. The nonlinear flow model is shown to be well-posed using recently published results. Numerical methods are developed and presented for the nonlinear flow model and the coupled transport model, solved with traditional stable approaches.

In Chapter 3, sensitivity analysis is introduced and developed for the coupled fluid flow and transport model. The existence of sensitivities is shown and the general sensitivity equation is developed. The forward sensitivity method and the adjoint sensitivity method are described and applied to the couple flow and transport model. To the author's knowledge, the sensitivity of a coupled system has not been discussed elsewhere. Chapter 4 consists of some examples applying the results from Chapter 3.

In Chapter 5, a novel application of the sensitivity equation to numerical continuation is shown. Numerical continuation is a solution method based on tracing a homotopy between an easy to solve problem and a more difficult problem of interest. In the present work, a related linear problem is used as the easy to solve problem and the nonlinear problem is is the more difficult problem. Several algorithms are presented which demonstrate how the sensitivity equation may arise in

numerical continuation. A superior method is shown in the novel Quasi-Newton continuation method.

In Chapter 7, model adaptivity is explored. Model adaptivity is concerned with approximating the modeling error, that is, the error incurred from the choice of model. It is proposed that the modeling error may be approximated using a homotopy such as the one developed in Chapter 5. The method is applied to the nonlinear flow problem.

In the Appendix, example source code, written in Python, is provided. The examples include the numerical implementations used for the experiments in this work.

Chapter 2 Preliminaries and Model Problems

In this chapter some mathematical background, the models of interest, and numerical methods are introduced to provide the reader with the necessary background. Section 2.1 introduces the mathematical background that will be used including abstract notions of derivatives including distributional derivatives in Section 2.1.1.1 following [69] and Frechet derivatives in Section 2.1.1.2 following [22], and Sobolev spaces in Section 2.1.2, following [10], which are Banach spaces of functions with derivatives.

Throughout this work, we will define various functional spaces defined over an open bounded region Ω of flow and transport, with sufficiently smooth boundary $\partial\Omega$. We will omit writing "in Ω.", if the region Ω is clear from the context.

We will introduce several model problems. Many models can be expressed in the form

$$Lu + N(u) = f, \tag{2.0.1}$$

where u is the unknown that is sought, L is the linear part of the model, and N is the non-linear part of the model. Before other analyses on the problem can begin, the problem must be shown to be well posed.

Recall that a problem is well posed if (1) a solution exists, (2) the solution is unique, and (3) the solution depends continuously on the data. Also of interest in

this work is differentiability with respect to the data, but this latter property is frequently not easy to prove with standard techniques.

The model problems we consider are as follows. First, for illustration, we develop a simple algebraic model; see Section 2.2. Second, we describe a nonlinear fluid flow model in porous media known as the non-Darcy model; see Section 2.3. A transport model coupled to the flow is discussed in Section 2.4.

The model problems are shown to be well-posed in Section 2.2.1, Section 2.3.1, and Sections 2.4.1-2.4.2, respectively. For the algebraic model, the differentiability with respect to the parameters is also shown in Section 2.2.2.

For the nonlinear flow and transport models, the numerical algorithm for finding the solution is described in Section 2.3.2 and Sections 2.4.3-2.4.4, respectively. The nonlinear flow solver consists of two parts: a linear flow solver and a Newton iteration. The linear flow model is solved using cell-centered finite difference methods which are described in Section 2.3.2.1. Hereby we mention that the cell-centered finite difference method is equivalent, up to quadrature error, to a mixed finite element method; this is described in Sections 2.3.2.2-2.3.2.6. The Newton iteration for the flow model is described in Section 2.3.2.7.

The transport model is discretized in time and space, as shown in Section 2.4.3. The space discretization is based on the finite volume method which is described in Section 2.4.4.

2.1 Mathematical Preliminaries

In this section, two forms of generalized derivatives are explored: distributional derivatives and Frechet derivatives. Also, spaces of functions with derivatives, known as Sobolev spaces, are introduced.

2.1.1 Generalized Derivatives

It is often necessary to consider functions which are not smooth enough for differential equations to be applied to them directly. In those cases, it is necessary to develop a more general definition of the derivative of a function. First, the notion of a distributional derivative in one dimension is developed. Next, the definition is extended to multiple dimensions to define partial derivatives. Finally, a more general class of derivatives in Banach spaces is introduced.

Recall the following definition.

Definition 2.1. A function f is said to have *compact support* in $\Omega \subset \mathbf{R}^d$ if $\{x\colon f(x) \neq 0\}$ is a compact subset of Ω.

Functions with compact support allow integration on compact sets which often simplifies calculations. In particular, the following sets are often used.

Definition 2.2. The set of functions with compact support and derivatives of order at least k, is denoted $C_0^k(\Omega)$. The set of infinitely differentiable functions with compact support is denoted $C_0^\infty(\Omega)$.

2.1.1.1 Distributional Derivatives

It is simple to construct examples of functions in one dimension which are not differentiable. A classic example is the function $f(x) = |x|$. This function is well behaved except at $x = 0$ where the derivative changes abruptly. Let $\phi \in C_0^\infty(\mathbf{R})$ and, formally, perform an integration by parts. Let A be large enough so that $\phi(x) = 0$ for $|x| \geq A$, then

$$\begin{aligned}\int_{-\infty}^{\infty} f'(x)\,\phi(x)\,dx &= \int_{-A}^{A} f'(x)\,\phi(x)\,dx \\ &= f(A)\,\phi(A) - f(-A)\,\phi(-A) - \int_{-A}^{A} f(x)\,\phi'(x)\,dx \\ &= -\int_{-\infty}^{\infty} f(x)\,\phi'(x)\,dx. \end{aligned} \tag{2.1.1}$$

This equality will guide the development of distributional derivatives.

It will help to have a definition of distributions. The following follows the formulation in [69].

Definition 2.3. A *distribution* f is a continuous and linear functional $f\colon C_0^\infty(\Omega) \to \mathbf{R}$. For $\phi \in C_0^\infty(\Omega)$ denote $f(\phi) = (f, \phi)$. By linear it is meant that

$$(f, a\phi + b\psi) = a\,(f, \phi) + b\,(f, \psi)\,, \tag{2.1.2}$$

and by continuous, it is meant that if $\{\phi_n\} \subset C_0^\infty(\Omega)$ is a sequence of functions

that converge uniformly to $\phi \in C_0^\infty(\Omega)$, as do their derivatives, then

$$(f, \phi_n) \to (f, \phi) \text{ as } n \to \infty. \tag{2.1.3}$$

Next, the convergence for distributions is defined.

Definition 2.4. A sequence $\{f_n\}$ of distributions is said to *converge weakly* to f if

$$(f_n, \phi) \to (f, \phi) \text{ as } n \to \infty, \tag{2.1.4}$$

for all $\phi \in C_0^\infty(\Omega)$.

Keeping in mind the motivation in (2.1.1), the following definition for the derivative of a distribution is given.

Definition 2.5. For any distribution f, define its *derivative* ∂f by the formula

$$\left(\partial_{x_j} f, \phi\right) = -\left(f, \partial_{x_j}\phi\right). \qquad \forall \phi \in C_0^\infty(\Omega) \tag{2.1.5}$$

Higher-order derivatives are defined using multi-index notation. If $\Omega \subset \mathbf{R}$, then the x_j may be omitted.

Definition 2.6. Let $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_d)$ be a *multi-index* (that is, a d-tuple of non-negative integers). Denote $|\alpha| = \sum_{i=1}^d \alpha_i$. Then denote the *higher-order partial derivative of* f by

$$\partial^\alpha f \;=\; \partial_{x_1}^{\alpha_1} \partial_{x_2}^{\alpha_2} \cdots \partial_{x_d}^{\alpha_d} f, \tag{2.1.6}$$

where $\partial_{x_j}^k f$ is defined by the formula

$$\left(\partial_{x_j}^k f, \phi\right) = -\left(\partial_{x_j}^{k-1} f, \partial_{x_j}\phi\right), \qquad \forall \phi \in C_0^\infty(\Omega). \tag{2.1.7}$$

2.1.1.2 Frechet Derivatives on Banach Spaces

Following Chapter 8 of [22], another derivative is introduced.

Definition 2.7. Let E, F be Banach spaces and $A \subset E$ an open set. Let $f, g\colon A \to F$ be continuous mappings. Then f and g are *tangent* at $x_0 \in A$ if

$$\lim_{\substack{x \to x_0 \\ x \neq x_0}} \frac{\|f(x) - g(x)\|_F}{\|x - x_0\|_E} = 0, \tag{2.1.8}$$

which requires $f(x_0) = g(x_0)$ since f, g are continuous. Then f is *differentiable* at $x_0 \in A$ if there is a linear mapping $u\colon E \to F$ such that

$$f(x_0) + u(x - x_0), \tag{2.1.9}$$

is tangent to f at x_0. Call u the *Frechet derivative* of f at x_0 and denote it $Df(x_0)$.

This notion may be extended to partial derivatives.

In what follows we will deal with derivatives with respect to a parameter of solutions to some equations. In this context, the solutions are defined implicitly, thus we recall the Implicit Function Theorem.

Definition 2.8. Assume Z, Y, Q are Banach spaces, and let $F\colon Z \times Q \to Y$ be

continuous on a neighborhood of U of the point (z_0, q). F is said to have a *strong partial Frechet derivative with respect to* z if there exists a linear mapping $\{\partial_z F\}$ that is the Frechet derivative of F in the component z. That is

$$\lim_{\substack{z \to z_0 \\ z \neq z_0}} \frac{\|F(z,q) - F(z_0,q) - \{\partial_z F\}(z - z_0, q)\|_Y}{\|z - z_0\|_Z} = 0. \tag{2.1.10}$$

The above definitions are equivalent to the usual derivative when both are applicable. As is true for derivatives on the real line, the Implicit Function Theorem applies.

Theorem 2.9 (Implicit Function Theorem [68])**.** *Assume Z, Y, Q are Hilbert spaces, and let $F\colon \mathcal{D}(F) \subset Z \times Q \to Y$ be continuous on a neighborhood U of the point $(z_0, q_o) \in int[\mathcal{D}(F)]$. If*

- $F(z_0, q_0) = 0$
- *F has a strong partial Frechet derivative $\partial_z F(z_0, q_0)$*
- $[\partial_z F(z_0, q_0)]^{-1}$ *exists and is bounded*

then there exist open neighborhoods U, W with $z_0 \in U \subset Z$, $q_0 \in W \subset Q$ such that for any $q \in W$, the equation $F(z, q) = 0$ has a unique solution $z = u(q) \in U$ and the mapping $u\colon W \to U$ is continuous. Thus $u(q)$ satisfies the equation $F(u(q), q) = 0$ for $q \in W$. Moreover, if $\partial_q F(z_0, q_0)$ exists, then $u(q)$ is Frechet differentiable at q_0 and the derivative is

$$\partial_q u(q_0) = -[\partial_z G(z_0, q_0)]^{-1} \circ [\partial_q G(z_0, q_0)]. \tag{2.1.11}$$

2.1.2 Sobolev Spaces

Sobolev spaces are an important class of function spaces which arise naturally in the analysis of partial differential equations. In this section the exposition in [10] is followed closely. Let $\Omega \subset \mathbf{R}^d$, $d = 1$, 2, or 3 with a Lipschitz continuous boundary $\Gamma = \partial\Omega$.

Definition 2.10. $L^p(\Omega)$ is the subspace of measurable functions,

$$L^p(\Omega) = \left\{ v : \left(\int_\Omega |v|^p \, dx \right)^{1/p} = \|v\|_{L^p(\Omega)} < +\infty \right\}, \tag{2.1.12}$$

for any integer $p \geq 1$. In particular $L^2(\Omega)$, the *square integrable functions* on Ω, are often of interest.

We have the following theorem.

Theorem 2.11. *For $p \geq 1$, the normed space $\left(L^p(\Omega), \|\cdot\|_{L^p(\Omega)} \right)$ is a Banach space. [48]*

The Riesz Representation Theorem and Corollary for $L^2(\Omega)$ are often applied.

Theorem 2.12 (Riesz Representation Theorem [48])**.** *Let $p > 1$ and $\frac{1}{p} + \frac{1}{q} = 1$. Then $l \in (L^p(\Omega))^*$ iff there exists a unique $g \in L^q(\Omega)$ such that*

$$l(f) = \int_\Omega f g dx, \qquad f \in L^p(\Omega). \tag{2.1.13}$$

Furthermore, g satisfies $\|l\|_{(L^p(\Omega))^*} = \|g\|_{L^q(\Omega)}$.

Corollary 2.13 (Riesz Representation Theorem for L^2 [48]). *$l \in (L^2(\Omega))^*$ iff there exists a unique $g \in L^2(\Omega)$ such that*

$$l(f) = \int_\Omega f g dx, \qquad f \in L^2(\Omega). \tag{2.1.14}$$

Furthermore, g satisfies $\|l\|_{(L^2(\Omega))^*} = \|g\|_{L^2(\Omega)}$.

Another set of useful spaces are the Sobolev spaces.

Sobolev spaces are defined by

$$W^{m,p}(\Omega) = \{v \in L^p(\Omega) : D^\alpha v \in L^p(\Omega) \, \forall \, |\alpha| \leq k\}, \tag{2.1.15}$$

where

$$D^\alpha v = \frac{\partial^{|\alpha|} v}{\partial_{x_1}^{\alpha_1} \cdots \partial_{x_n}^{\alpha_n}}, |\alpha| = \alpha_1 + \cdots + \alpha_n, \tag{2.1.16}$$

in the sense of distributions. In particular, the spaces

$$H^k(\Omega) = W^{k,2}(\Omega), \tag{2.1.17}$$

will be used. On this space, we will use the *semi-norm*

$$|v|_{m,\Omega} = \left(\sum_{|\alpha|=m} |D^\alpha v|^2_{L^2(\Omega)} \right)^{1/2}, \tag{2.1.18}$$

and the *norm*

$$\|v\|_{m,\Omega} = \left(\sum_{k \leq m} |v|^2_{k,\Omega} \right)^{1/2}. \tag{2.1.19}$$

Note that $L^2(\Omega) = H^0(\Omega)$, which leads to the notation $|v|_{0,\Omega}$ to denote the norm $\|v\|_{L^2(\Omega)}$. It is often written $|v|_0$ and $\|v\|_{L^2}$ when there is only one space of interest. Of particular interest are $L^2(\Omega)$, $H^1(\Omega)$, $H_0^1(\Omega)$, $H^2(\Omega)$, and $H_0^2(\Omega)$.

Definition 2.14. Denote by $\mathcal{D}(\Omega)$ the *space of indefinitely differentiable functions* having a compact support in Ω, and by $H_0^k(\Omega)$ the completion of $\mathcal{D}(\Omega)$ for the topology defined by the norms 2.1.19.

There are other definitions which are equivalent, but this definition will suffice for Ω with Lipschitz continuous boundary.

Definition 2.15. With a Lipschitz continuous boundary, there exists an operator $\gamma_0 \colon H^1(\Omega) \to L^2(\Gamma)$ which is linear and continuous, such that $\gamma_0 v$ is the *trace* of v on Γ for every smooth v. Call $\gamma_0 v$ the *trace* of v on Γ and denote it by $v|_\Gamma$. Identify

$$H^{1/2}(\Gamma) = \gamma_0\left(H^1(\Omega)\right), \tag{2.1.20}$$

with

$$\|g\|_{H^{1/2}(\Gamma)} = \inf_{\substack{v \in H^1(\Omega) \\ \gamma_0 v = g}} \|v\|_{H^1(\Omega)}. \tag{2.1.21}$$

Then

$$H^1(\Gamma) \subset \gamma_0\left(H^1(\Omega)\right) \subset L^2(\Omega) \equiv H^0(\Gamma), \tag{2.1.22}$$

where every inclusion is strict. Similarly, the traces of functions in $H^2(\Omega)$ are

$$H^{3/2}(\Gamma) = \gamma_0\left(H^2(\Omega)\right), \tag{2.1.23}$$

with norm

$$\|g\|_{H^{3/2}(\Gamma)} = \inf_{\substack{v\in H^2(\Omega)\\ \gamma_0 v = g}} \|v\|_{H^2(\Omega)}. \tag{2.1.24}$$

This can be generalized to traces of higher-order derivatives, as well. If the boundary is smooth enough, then it is possible to define $\frac{\partial v}{\partial n}|_\Gamma \in H^{1/2}(\Gamma)$ for $v \in H^2(\Omega)$. Intuitively, the Sobolev spaces with fractional order can be considered as having regularity properties that are between the properties of the integer orders nearby.

Definition 2.16. Let

$$H_0^1(\Omega) = \left\{v : v \in H^1(\Omega), v|_\Gamma = 0\right\}, \tag{2.1.25}$$

and

$$H_0^2(\Omega) = \left\{v : v \in H^1(\Omega), v|_\Gamma = 0, \frac{\partial v}{\partial n}|_\Gamma = 0\right\}. \tag{2.1.26}$$

Theorem 2.17 (Poincaré Inequality)**.** *For $v \in H_0^1(\Omega)$, the Poincaré inequality is*

$$|v|_{0,\Omega} \le C(\Omega)\,|v|_{1,\Omega}, \tag{2.1.27}$$

and the seminorm $|\cdot|_{1,\Omega}$ is therefore a norm on $H_0^1(\Omega)$, equivalent to $\|\cdot\|_{1,\Omega}$.

Next, consider functions that vanish on part of the boundary.

Definition 2.18. Suppose $\Gamma = \Gamma^D \cup \Gamma^N$ with $\Gamma^D \cap \Gamma^N = \emptyset$. Define

$$H_{0,D}^1(\Omega) = \left\{v : v \in H^1(\Omega), v|_D = 0\right\}, \tag{2.1.28}$$

then

$$H_0^1(\Omega) \subset H_{0,D}^1(\Omega) \subset H^1(\Omega). \tag{2.1.29}$$

The dual space of $H^{1/2}(\Gamma)$ is often used.

Definition 2.19. Let $H^{-1/2}(\Gamma) = \left(H^{1/2}(\Gamma)\right)^*$ be the *dual space* of $H^{1/2}(\Gamma)$ with dual norm

$$\|v^*\|_{-1/2,\Gamma} = \sup_{v \in H^{1/2}(\Gamma)} \frac{\langle v, v^* \rangle}{\|v\|_{1/2,\Gamma}}, \tag{2.1.30}$$

where $\langle \cdot, \cdot \rangle$ denotes the duality between $H^{-1/2}(\Gamma)$ and $H^{1/2}(\Gamma)$.

It is sometimes convenient to write, formally, $\int_\Gamma v^* v dx$ instead of $\langle v^*, v \rangle$.

A few important issues arise in treating these spaces. If Γ_0 is a part of Γ, then $\phi \in H^{1/2}(\Gamma_0)$ cannot be extended by the zero function outside of Γ_0 to a function in $H^{1/2}(\Gamma)$ even in the case that $\mathcal{D}(\Gamma_0)$ is dense in $H^{1/2}(\Gamma_0)$. In the dual sense, if $\Gamma = \Gamma_1 \cup \Gamma_2$, one does not get all of $H^{-1/2}(\Gamma)$ by patching functions of $H^{-1/2}(\Gamma_1)$ and $H^{-1/2}(\Gamma_2)$. This has consequences in the analysis of finite element methods since one often encounters $H^{1/2}(\partial T)$ and $H^{-1/2}(\partial T)$ where T is an element in the partition of Ω.

Another important space is $H(\mathrm{div}; \Omega)$.

Definition 2.20. Let

$$H(\mathrm{div}; \Omega) = \left\{ u : u \in \left(L^2(\Omega)\right)^d, \nabla \cdot u \in L^2(\Omega) \right\}, \tag{2.1.31}$$

where $\nabla \cdot u$ is the divergence of u, with the norm

$$\|u\|_{\mathrm{div},\Omega} = \left(|u|^2_{0,\Omega} + |\nabla \cdot u|^2_{0,\Omega} \right)^{1/2}. \tag{2.1.32}$$

For $u \in H(\mathrm{div};\Omega)$, it is possible to define

$$u \cdot n|_{\Gamma} \in H^{-1/2}(\Gamma). \tag{2.1.33}$$

This space will occur in the analysis of mixed finite element methods constantly. Functions belonging to $H(\mathrm{div};\Omega)$ satisfy an integration by parts formula.

Theorem 2.21 (Integration By Parts)**.** *For $u \in H(\mathrm{div};\Omega)$, the following formula for integration by parts holds*

$$\int_{\Omega} p \nabla \cdot u dx + \int_{\Omega} u \cdot \nabla p dx = \langle u \cdot n, v \rangle, \qquad \forall p \in H^1(\Omega). \tag{2.1.34}$$

2.2 An Algebraic Model

As a particular case of (2.0.1), consider the simple non-linear algebraic equation

$$\beta u |u| + u = -kp, \tag{2.2.1}$$

with $\beta \geq 0$. Relating parts of this equation to (2.0.1), identify $Lu = u$, $N(u) = \beta u|u|$, and $f = -kp$. Below, it will be shown that this problem is well-posed and the differentiability with respect to the parameters β and kp will be explored. To

show that the problem is well posed, a unique solution must be found and it must be shown that it depends continuously on the parameters β and kp.

2.2.1 A Unique Solution

If $\beta = 0$, then the unique solution is easy to obtain as $u = -kp$. If $\beta \neq 0$, then there are two possibilities. First, if $u > 0$, the quadratic formula yields $u = \frac{-1 \pm \sqrt{1-4\beta kp}}{2\beta}$. Since the square root gives a positive value and subtracting a positive value from -1 will give a negative number, it must be that $u = \frac{-1+\sqrt{1-4\beta kp}}{2\beta}$. Also, since $u > 0$, $\sqrt{1-4\beta kp} > 1$ so $kp < 0$. Second, if $u < 0$, the quadratic formula yields $u = \frac{-1 \pm \sqrt{1+4\beta kp}}{-2\beta}$. Since $u < 0$, it must be that $-1 \pm \sqrt{1+4\beta kp} > 0$. Since the square root gives a positive value and subtracting a positive value from -1 will give a negative number, it must be that $u = \frac{-1+\sqrt{1+4\beta kp}}{-2\beta}$. Also, since $-1 + \sqrt{1+4\beta kp} > 0$, it must be $kp > 0$. All of that leads to the unique solution

$$u = \begin{cases} -kp, & \text{if } \beta = 0, \\ \frac{-1+\sqrt{1-4\beta kp}}{2\beta}, & \text{if } \beta \neq 0 \text{ and } kp \leq 0, \\ \frac{1-\sqrt{1+4\beta kp}}{2\beta}, & \text{if } \beta \neq 0 \text{ and } kp \geq 0. \end{cases} \tag{2.2.2}$$

As long as $\beta \neq 0$ it is easy to see that this is continuous in β and kp. We would like to know that $\lim_{\beta \to 0} u(\beta) = -kp$, that is that u is continuous at $\beta = 0$. If

$kp \leq 0$, using L'Hopital's rule

$$\begin{aligned} \lim_{\beta \to 0} \frac{-1 + \sqrt{1 - 4\beta kp}}{2\beta} &= \lim_{\beta \to 0} \frac{\frac{-4kp}{2\sqrt{1-4\beta kp}}}{2} \\ &= \lim_{\beta \to 0} \frac{-kp}{\sqrt{1 - 4\beta kp}} \\ &= -kp. \end{aligned} \tag{2.2.3}$$

Similarly, if $kp \geq 0$,

$$\begin{aligned} \lim_{\beta \to 0} \frac{1 - \sqrt{1 + 4\beta kp}}{2\beta} &= \lim_{\beta \to 0} \frac{\frac{-4kp}{2\sqrt{1+4\beta kp}}}{2} \\ &= \lim_{\beta \to 0} \frac{-kp}{\sqrt{1 + 4\beta kp}} \\ &= -kp. \end{aligned} \tag{2.2.4}$$

So the unique solution depends continuously on the data; that is, the problem is well posed.

2.2.2 Differentiability With Respect to the Parameters

Next, it is useful to show that the solution is differentiable with respect to the parameters β and kp. Using implicit differentiation with respect to β, formally,

$$\{\partial_\beta u\} = -\frac{u\,|u|}{2\beta\,|u| + 1}, \tag{2.2.5}$$

which is continuous in β since $\beta |u| \geq 0$ and u is continuous in β, and so the differentiation was justified. Also

$$\{\partial_{kp} u\} \quad = \quad -\frac{1}{2\beta |u| + 1}, \tag{2.2.6}$$

which is continuous in kp for the same reasons as above.

2.3 Fluid Flow in Porous Media

For a thorough treatment of fluid flow in porous media see [6]. In this work, a model is derived based on [26]; we also follow [27, 25, 23].

Three equations are required in order to develop a model for fluid flow in porous media: conservation of momentum, conservation of mass, and an equation of state. The equation for *conservation of momentum* describes the relationship between the velocity u and the gradient of pressure p:

$$\kappa (x, \rho, u)\, u + \nabla p \quad = \quad g (x)\,. \tag{2.3.1}$$

Here $\kappa (x, \rho, u)$ is the (non-linear) resistance tensor, ρ is the density, and $g \in \left(L^2 (\Omega)\right)^2$ represents gravity-like effects. In general, κ is a full tensor which can be determined from measurements or homogenized from microscale data [24, 31]. See [59, 58] specifically for finding κ from pore-scale simulations.

The form of κ is taken to be

$$\kappa(x,\rho,u) = K^{-1} + \beta(x,\rho u), \tag{2.3.2}$$

as in [27], where K^{-1} and $\beta(x,\rho u)$ are symmetric, uniformly positive definite tensors with components in $L^{\infty}(\Omega)$. A tensor $\beta(x)$ is *symmetric, uniformly positive definite* if there exist constants $0 < \beta_m \leq \beta_M$, independent of x, such that $\beta_m \|u\| \leq u \cdot \beta(x)\, u \leq \beta_M \|u\|$. Taking $\beta(x,\rho u) \equiv 0$ results in the Darcy system. Let $\beta(x,\rho u)$ be diagonal. If κ has off-diagonal terms, then it is said to be *anisotropic*.

The equation for *conservation of mass* is

$$\phi(x)\,\partial_t \rho + \nabla \cdot u = f(x,t). \tag{2.3.3}$$

In this equation $\phi(x)$ is the porosity. The function $f(x,t)$ represents sources and sinks of mass such as wells. Since κ is uniformly, symmetric positive definite, it is invertible. It is possible to use the inverse of κ to write this as one equation, but that will not be done.

The *equation of state* is an equation which relates ρ and p so that one of the variables may be eliminated. To simplify the problem assume that ρ and ϕ are constants equal to one:

$$\begin{aligned} \kappa(x,u)\,u + \nabla p &= g(x), \\ \nabla \cdot u &= f(x). \end{aligned}$$

In this case, take $f \in L^2(\Omega)$. For simplicity of notation, x is sometimes omitted so that

$$\begin{aligned} \kappa(u)\, u + \nabla p &= g, \\ \nabla \cdot u &= f. \end{aligned}$$

In this work,

$$\kappa(x,u) = K^{-1}(x) + \beta(x) \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix}, \tag{2.3.4}$$

where $K^{-1}(x)$ and $\beta(x)$ are symmetric uniformly positive definite tensors and $\beta(x)$ is diagonal.

Since Ω is a bounded domain, it will also be important to introduce boundary conditions. Let $\Gamma^D \subset \partial\Omega$ be the part of the boundary where Dirichlet conditions are to be satisfied and $\Gamma^N \subset \partial\Omega$ be the part of the boundary where Neumann conditions are to be satisfied. It is assumed that $\Gamma = \Gamma^N \cup \Gamma^D$ and $\Gamma^N \cap \Gamma^D = \emptyset$. With that notation, the following system of equations are complete

$$\nabla \cdot u = f(x), \qquad \text{in } \Omega, \tag{2.3.5}$$

$$\kappa(u)\, u = -\nabla p + g(x), \qquad \text{in } \Omega, \tag{2.3.6}$$

$$p = p_D(x), \qquad \text{on } \Gamma^D, \tag{2.3.7}$$

$$u \cdot n = u_N(x), \qquad \text{on } \Gamma^N. \tag{2.3.8}$$

Call that system of equations the *strong form of the flow model.*

2.3.1 Well-Posedness of the Nonlinear non-Darcy Flow Model

Proving the well-posedness for a non-linear flow model presents some difficulties. First we overview approaches known from the literature for related models, and next we prove one of our main contributions.

2.3.1.1 Related Models

Here we provide literature overview on non-Darcy flow models and their analyses. The models considered are similar to, but not exactly the same as our model of flow (2.3.5)-(2.3.8). An important distinction is that our model is not an evolution model while all of the following models are.

In [23], models of the form

$$G(\rho, v) + \nabla p = 0, \qquad x \in \Omega, t \geq 0, \tag{2.3.9}$$

$$\phi \partial_t \rho + \nabla \cdot (\rho v) = f, \qquad x \in \Omega, t \geq 0, \tag{2.3.10}$$

$$p = p(\rho), \tag{2.3.11}$$

or an equivalent system of the form

$$\overline{G}(\rho, m) + \nabla p = 0, \qquad x \in \Omega, t \geq 0, \tag{2.3.12}$$

$$\phi \partial_t \rho + \nabla \cdot m = f, \qquad x \in \Omega, t \geq 0, \tag{2.3.13}$$

$$p = p(\rho), \tag{2.3.14}$$

where $m = \rho v$ is the momentum, are considered. The model considered in this work does not have the time derivative, but has a similar form to (2.3.9)-(2.3.11) or (2.3.12)-(2.3.14).

The existence and uniqueness of a solution are proved by invoking a theorem from [38]. Finally, a finite element method (see Section 2.3.2.2) for the approximation of the solution is developed. In particular, the lowest order mixed finite element spaces are advocated and analyzed for the spatial derivatives. The time derivative is approximated using a difference quotient as in Section 2.4.3.

Another similar problem in addressed in [25]. Their model is of the form

$$\begin{aligned} \epsilon \partial_t \rho + \nabla \cdot \rho v &= 0, \\ (\rho c)^* \partial_t T + (\rho c)_f v \cdot \nabla T - \nabla \cdot [(\lambda^* (T) + D(v)) \cdot \nabla T] &= 0, \end{aligned}$$

and one of the following momentum equations

$$\begin{aligned} \mu(T) v + K \cdot (\nabla p + \rho g) &= 0, \\ \frac{\rho}{\epsilon} \partial_t v + \mu(T) K^{-1} v + c_j \rho \sigma(v) + \nabla p + \rho g &= 0. \end{aligned}$$

They assume the fluid is "incompressible but dilatable" which they define with the constitutive equation

$$\rho = \rho_0 (1 - \beta (T - T_0)).$$

It is also assumed that

$$\begin{aligned} \mu(T) &= \lambda_0 e^{\gamma(T-T_0)}, \\ D_{ij}(v) &= \alpha_L |v|^{-1} v_i v_j + \alpha_T |v|^{-1} \left(|v|^2 \delta_{ij} - v_i v_j \right). \end{aligned}$$

The terms $(\rho c)_f$ and $(\rho c)_s$ are the fluid and solid head capacity and $(\rho c)_* = \epsilon (\rho c)_f + (1-\epsilon)(\rho c)_s$. Next, $\lambda^*(T)$ is the thermal conductivity of the medium, D the dispersion tensor, $\mu(T)$ the viscosity, and K the permeability. The term $\sigma(v)$ is allowed to take several forms which are related to the permeability of the Darcy form of the equation. The thermal terms and the time dependence are the primary differences with the problem we are concerned with. The main result of the paper is to give some assumptions that guarantee the existence and uniqueness of a solution.

In [4], a class of problems with the form

$$g(x, |u|_B) u = -\Pi \nabla p,$$

is analyzed. The term $|u|_B$ is defined by

$$|u|_B = \sqrt{(B(x) u, u)},$$

where $B(x)$ is a positive definite tensor with bounded entries. It is required that $g(x, 0) > 0$. In the analysis, the spatial variable is eliminated by assuming that Π and B are the identity matrix. It is further assumed that $g(0) > 0$ and $g'(s) \geq 0$

for all $s \geq 0$. Those assumptions allow them to write

$$G\left(|u|\right) = g\left(|u|\right) u = |\nabla p|$$

and

$$u = -K\left(|\nabla p|\right) \nabla p,$$

where

$$K\left(|\nabla p|\right) = \frac{1}{g\left(G^{-1}\left(|\nabla p|\right)\right)}.$$

It is shown that $F\left(y\right) = K\left(|y|\right) y$ is monotone. Two initial boundary value problems are considered. Both use the following equations:

$$\begin{aligned} \partial_t p = \nabla \cdot K\left(|\nabla p|\right) \nabla p, \text{ in } \Omega \times \left(0, \infty\right), \\ p\left(x, 0\right) = p_0\left(x\right), \text{ in } \Omega, \\ p \cdot n = 0, \text{ on } \Gamma_e \times \left(0, \infty\right). \end{aligned}$$

Two types of boundary conditions are considered: Dirichlet conditions are considered with the form

$$p\left(x, t\right) = \phi\left(x, t\right), \text{ on } \Gamma_i \times \left(0, \infty\right),$$

and a total flux condition with the form

$$-\int_{\Gamma_i} K\left(|\nabla p|\right) \nabla p \cdot n = Q\left(t\right), \text{ on } \Gamma_i \times \left(0, \infty\right).$$

Uniqueness is proved for both problems.

2.3.1.2 Well-posedness proof

Our model is a special case of a model studied in [66], and so we would like to apply the following theorem from that paper.

Theorem 2.22 (Theorem 3.2 of [66])**.** *Let V and Q be real Hilbert spaces, and let there be given the operators E_1, $A : V \to V'$, $B : V \to Q'$, E_2, $C : Q \to Q'$. Assume E_1 and E_2 are continuous, linear, symmetric, and nonnegative, B is continuous and linear, and A and C are maximal monotone with (for simplicity) $A(0) \ni 0$, $C(0) \ni 0$. Denote by V_1 and Q_2 the spaces V and Q with respective semiscalar products arising from E_1 and E_2, and let their (Hilbert space) duals be designated by V_1' and Q_2'. Assume the following:*

- *The operators E_1, A, and B satisfy*

$$\lim_{\substack{\|u\|_V + \|\xi\|_{V'} \to +\infty \\ \text{with } \xi \in A(u)}} \left(|u|_{V_1}^2 + \xi(u) + \|Bu\|_{Q'}^2 \right) = +\infty. \tag{2.3.15}$$

- *The operator $B : V \to Q'$ has closed range. (Then the same holds for $B' : Q \to V'$.)*

- *The operators E_2 and C satisfy*

$$\lim_{\substack{\|\eta\|_{Q'} \to +\infty \\ \text{with } \eta \in C(p)}} \left(|p|_{Q_2}^2 + \eta(p) \right) = +\infty. \tag{2.3.16}$$

- *There is a constant K such that for every $h \in Q'$ and $\varepsilon > 0$, the condition*

$$p \in KerB' : (\varepsilon \mathcal{R}_2 + E_2)\, p\,(q) = h\,(q) \ \ \textit{for all } q \in KerB' \tag{2.3.17}$$

implies that $\|p\|_Q \leq K \|h\|_{Q'}$.

Then the resolvent system $u \in V$, $p \in Q$:

$$E_1 u + \xi + B'p = g, \qquad \xi \in A\,(u) \ \ \textit{in } V', \tag{2.3.18}$$

$$E_2 p - Bu + \eta = f, \qquad \eta \in C\,(p) \ \ \textit{in } Q' \tag{2.3.19}$$

has a solution for each pair $g \in V_1'$ and $f \in Q_2'$.

We will take $E_1 = 0$, $E_2 = 0$, $A\,(u) = \kappa\,(u)\, u$, B' the negative of the gradient operator, B the divergence operator, and $C = 0$. In that case, (2.3.18)-(2.3.19) becomes

$$\kappa\,(u)\, u - \nabla p = g, \qquad \text{in } V',$$

$$\nabla \cdot u = -f, \qquad \text{in } Q',$$

which is the system in (2.3.5)-(2.3.8).

Remark 3.1 of [66] tells us that (2.3.15) holds if A maps bounded sets into bounded sets, and satisfies the growth condition

$$\lim_{\substack{\|u\|_V \to +\infty \\ \xi \in A(u)}} \left(\xi\,(u) + \|Bu\|_{Q'}^2 \right) = +\infty.$$

Another simplification comes from Remark 3.2, which tells us that (2.3.16) holds if $C = 0$. Finally, Remark 3.3 tells us that (2.3.17) is satisfied if $C = 0$ and $E_2 = 0$. The following corollary summarizes the findings so far.

Corollary 2.23. *Let $V = H(\mathrm{div}; \Omega)$ and $W = L^2(\Omega)$. The system*

$$\kappa(u)\, u - \nabla p = g, \qquad \text{in } V', \tag{2.3.20}$$

$$\nabla \cdot u = -f, \qquad \text{in } W', \tag{2.3.21}$$

with κ defined by (2.3.4) is well posed for $f \in W'$ and $g \in V'$ since

- *The divergence operator is continuous and linear with closed range.*
- *The operator $\kappa(u)\, u$ satisfies $\kappa(0)\, 0 = 0$.*
- *The operator $\kappa(u)\, u$ maps bounded sets into bounded sets.*
- *The growth condition*

$$\lim_{\substack{\|u\|_V \to +\infty \\ \xi \in A(u)}} \left(u \cdot \kappa(u)\, u + \|\nabla \cdot u\|_{W'}^2 \right) = +\infty \tag{2.3.22}$$

  is satisfied.

- *The operator $\kappa(u)\, u$ is a monotone operator.*

Proof. It is well known that the divergence operator is continuous and linear with closed range [65]. Since $\kappa(0) = K^{-1}$ is a positive definite tensor

$$\kappa(0)\, 0 = 0.$$

Let $S \subset V$ be a bounded set, then there exists $C > 0$ such that $\|u\|_V \leq C$ for all $u \in S$. Let $u \in S$, then

$$\begin{aligned}
\|\kappa(|u|)\, u\| &= \left\| K^{-1}u + \sum_{i=1}^{n} |u|^{\alpha_i} \beta_i^{-1} u \right\| \\
&\leq \left\| K^{-1}u \right\| + \left\| \sum_{i=1}^{n} |u|^{\alpha_i} \beta_i^{-1} u \right\| \\
&\leq \left\| K^{-1}u \right\| + \sum_{i=1}^{n} |u|^{\alpha_i} \left\| \beta_i^{-1} u \right\| \\
&\leq \left\| K^{-1} \right\| \|u\| + \sum_{i=1}^{n} |u|^{\alpha_i} \left\| \beta_i^{-1} \right\| \|u\| \\
&\leq \left(\left\| K^{-1} \right\| + \max_{1 \leq i \leq n} |u|^{\alpha_i} \sum_{i=1}^{n} \left\| \beta_i^{-1} \right\| \right) C \\
&\leq \left(C_K + n \max_{1 \leq i \leq n} |u|^{\alpha_i} \max_{1 \leq i \leq n} \left\| \beta_i^{-1} \right\| \right) C \\
&\leq \left(C_K + C_\beta \max_{1 \leq i \leq n} |u|^{\alpha_i} \right) C
\end{aligned}$$

Replacing the norms of the tensors is justified since K^{-1} and β_i^{-1} have bounded entries. Since $|u| \leq c\|u\|_V$ for some $c > 0$ we have

$$\begin{aligned}
\|\kappa(|u|)\, u\| &\leq \left(C_K + C_\beta \max_{1 \leq i \leq n} |u|^{\alpha_i} \right) C \\
&\leq \left(C_K + \max_{1 \leq i \leq n} (c\|u\|_V)^{\alpha_i} \right) C
\end{aligned}$$

which is a constant independent of u. So $\kappa(|u|)\, u$ maps bounded sets into bounded sets.

Next,

$$\begin{aligned}\lim_{\substack{\|u\|_V \to +\infty \\ \xi \in A(u)}} \left(u \cdot \kappa\left(|u|\right) u + \|\nabla \cdot u\|_{W'}^2\right) &\geq \lim_{\substack{\|u\|_V \to +\infty \\ \xi \in A(u)}} u \cdot \kappa\left(|u|\right) u \\ &\geq \lim_{\substack{\|u\|_V \to +\infty \\ \xi \in A(u)}} u \cdot \kappa\left(0\right) u \\ &= \lim_{\substack{\|u\|_V \to +\infty \\ \xi \in A(u)}} u \cdot K^{-1} u \\ &= +\infty,\end{aligned}$$

so the growth condition is satisfied.

Finally, it remains to show that the operator $A\left(u\right) = \kappa\left(|u|\right) u$ is monotone. Let $u, v \in V$, then

$$\begin{aligned}\left(A\left(u\right) - A\left(v\right), u - v\right) &= \left(K^{-1}u + \beta \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix} u, u - v\right) \\ &\quad - \left(K^{-1}v + \beta \begin{bmatrix} |v_1| & \\ & |v_2| \end{bmatrix} v, u - v\right) \\ &= \left(K^{-1}\left(u - v\right), u - v\right) \\ &\quad + \left(\beta \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix} u - \beta \begin{bmatrix} |v_1| & \\ & |v_2| \end{bmatrix} v, u - v\right)\end{aligned}$$

Since K^{-1} is positive definite,

$$\left(K^{-1}\left(u - v\right), u - v\right) \geq 0.$$

Let $M = \max(|u_1|, |v_1|)$ and $m = \min(|u_1|, |v_1|)$, then

$$u_1^2 |u_1| + v_1^2 |v_1| = M^3 + m^3$$

and

$$|u_1 v_1 (|u_1| + |v_1|)| \leq Mm(M + m) = M^2 m + Mm^2$$

so

$$\begin{aligned}(u_1 |u_1| - v_1 |v_1|)(u_1 - v_1) &\geq M^3 + m^3 - M^2 m - Mm^2 \\ &= \left[M^2 - m^2\right](M - m) \\ &\geq 0.\end{aligned}$$

In the same way,

$$(u_2 |u_2| - v_2 |v_2|)(u_2 - v_2) \geq 0.$$

Since the entries of $\beta(x)$ are nonnegative,

$$\left(\beta \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix} u - \beta \begin{bmatrix} |v_1| & \\ & |v_2| \end{bmatrix} v, u - v\right) \geq 0,$$

so $A(u) = \kappa(|u|)u$ is monotone. □

Remark 2.24. The present theory makes it difficult to use a more general form for

β. For instance, if

$$\beta = \begin{bmatrix} \beta_{11} & \beta_{12} \\ \beta_{12} & \beta_{22} \end{bmatrix}$$

and

$$|u| = \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix},$$

then

$$\beta\,|u| = \begin{bmatrix} \beta_{11}\,|u_1| & \beta_{12}\,|u_2| \\ \beta_{12}\,|u_1| & \beta_{22}\,|u_2| \end{bmatrix}.$$

A theorem in [49] states that the product $\beta\,|u|$ is positive definite iff $\beta\,|u|$ is normal, that is

$$\beta\,|u|\,(\beta\,|u|)^T = (\beta\,|u|)^T\,\beta\,|u|\,. \tag{2.3.23}$$

The entry in the first row and first column on the left side of (2.3.23) is

$$\beta_{11}^2\,|u_1|^2 + \beta_{12}^2\,|u_2|^2$$

and on the right side of (2.3.23) is

$$\beta_{11}^2\,|u_1|^2 + \beta_{12}^2\,|u_1|^2\,,$$

which are not equal unless $|u_1| = |u_2|$.

Remark 2.25. Another form for $\kappa(u)$ was proposed in [27] with

$$\kappa(u) = K^{-1} + \beta |u|$$

where

$$|u| = \sqrt{\sum_{i=1}^{d} u_i^2}.$$

That choice of $\kappa(u)$ will not suffer from the problems in Remark 2.24, but will have another problem. An example will show some of the difficulty in that choice. Let

$$\beta = \begin{bmatrix} 10000 & \\ & \frac{1}{10000} \end{bmatrix}.$$

Choose $u = (1,0)^T$ and $v = \left(\frac{1}{5}, 100\right)^T$ then $|u| = 1$ and

$$|v| = \sqrt{\frac{250001}{25}} > 100.$$

Then

$$\begin{aligned} (\beta(|u|u - |v|v), u - v) &= 10000(|u|u_1 - |v|v_1)(u_1 - v_1) \\ &\quad + \frac{1}{10000}(|u|u_2 - |v|v_2) \\ &= 8000 - 1600|v| - \frac{1}{100}|v| \\ &< 0. \end{aligned}$$

That is, the operator $\beta(\cdot)$ is not monotone.

2.3.2 Numerical Algorithm for the Nonlinear Flow Model

There are two pieces to the solver that will be developed: solving the linear flow problem with anisotropy, then solving the non-linear flow problem. The linear flow problem may be solved by applying the results in [2]. In [2], a method is proposed which is based on the mixed finite element method. It is shown that the mixed finite element method is equivalent, up to the error in the numerical quadrature, to a cell-centered finite difference method. It is proposed that the non-linear flow problem may be solved by using an approximate Newton's method. A similar method was proposed in [27], however, in that paper only diagonal tensors were allowed. The proof of convergence of the method is not provided, however numerical experiments have demonstrated the effectiveness of the method. See Listing 1 for the implementation.

2.3.2.1 Cell-Centered Finite Difference Methods for Elliptic Equations

Consider the equation

$$-\nabla \cdot (\kappa \nabla p) = f. \tag{2.3.24}$$

Let κ be a symmetric uniformly positive definite tensor. Then the equation (2.3.24) is elliptic. For well-posedness, we need to impose boundary conditions. There are many options for boundary conditions that will close the system, but in this work Dirichlet conditions which fix p and Neumann conditions which fix $\kappa \nabla p$ are used.

It is not sufficient to use only Neumann conditions, but only Dirichlet conditions will suffice.

For simplicity of presentation, two spatial dimensions will be used. Equations of this type arise naturally in many settings including the study of fluid flow in porous media, where κ can be a full tensor. A full tensor arises in the study of fluid flow in porous media when there is anisotropy in the medium. For an exposition on a simpler model ($\kappa \equiv I$), see [12] or [40]. In [40], a higher order method is explored. The following is related to the method presented in [2] and [62].

For an illustration of the computational grid used for the cell-centered finite difference method, see Figure 2.3.1. Let $x_0 < x_1 < \cdots < x_{N_x}$ and $y_0 < y_1 < \cdots < y_{N_y}$ be given sequences defining the grid. Let $x_{i+1/2} = \frac{x_i + x_{i+1}}{2}$ and $y_{j+1/2} = \frac{y_j + y_{j+1}}{2}$. Define $\Delta x_i = x_{i+1} - x_i$ and $\Delta y_j = y_{j+1} - y_j$. The pressure will be approximated at the centers of the cells, that is

$$p_{i+1/2,j+1/2} \approx p\left(\frac{x_i + x_{i+1}}{2}, \frac{y_j + y_{j+1}}{2}\right). \tag{2.3.25}$$

The difficulty is, then, to approximate the differential equation above. First, discretize the divergence operator and approximate $\kappa\nabla p$ at the cell edges by

$$\begin{aligned}[\nabla \cdot (\kappa \nabla p)]_{i+1/2,j+1/2} &= \frac{[\kappa\nabla p]\,|_{x_{i+1},y_{j+1/2},1} - [\kappa\nabla p]\,|_{x_i,y_{j+1/2},1}}{\Delta x_i} \\ &\quad + \frac{[\kappa\nabla p]\,|_{x_{i+1/2},y_{j+1},2} - [\kappa\nabla p]\,|_{i+1/2,y_j,2}}{\Delta y_j}, \end{aligned} \tag{2.3.26}$$

where the last subscript indicates which component of $\kappa\nabla p$ is used. Next, consider

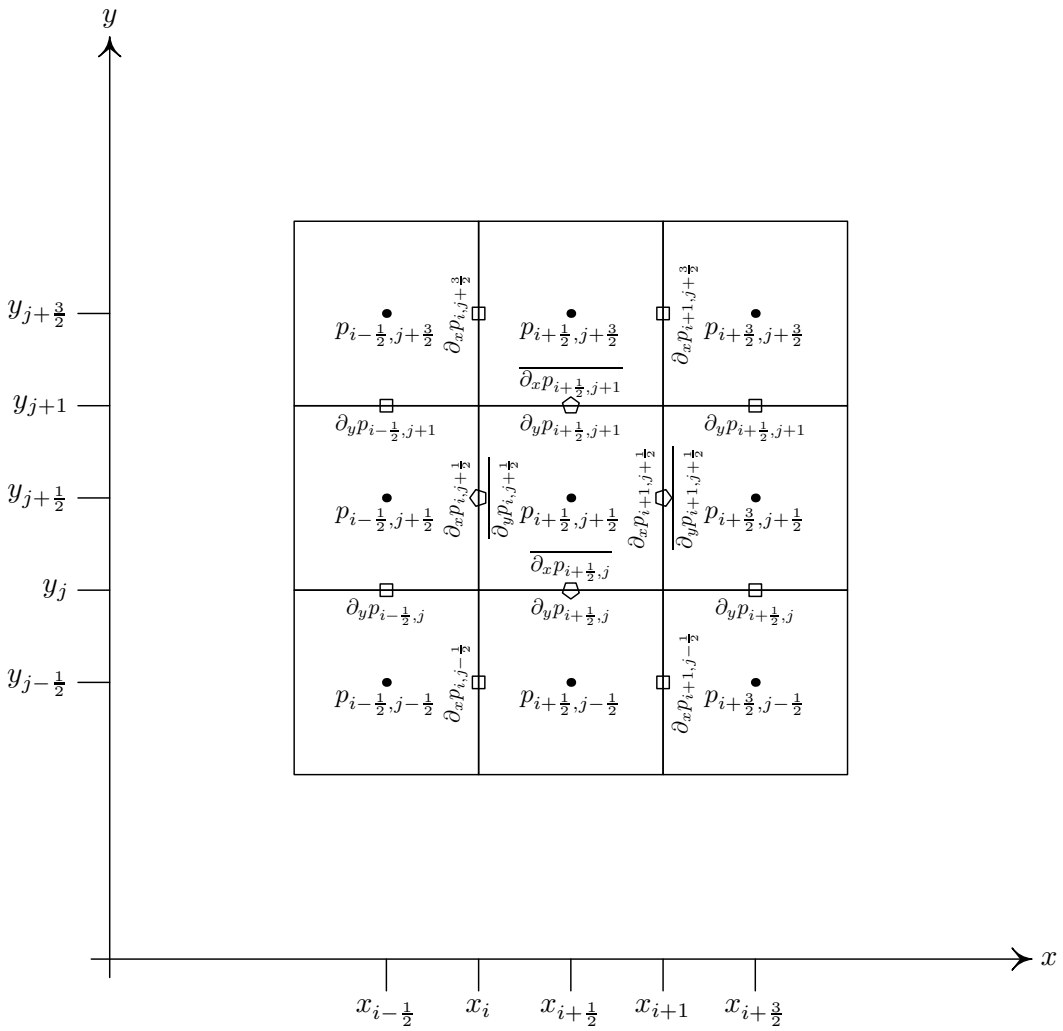

Figure 2.3.1: The computational grid for the Cell-Centered Finite Difference Method. The pressure p is approximated at cell-centers $\left(x_{i+\frac{1}{2}}, y_{j+\frac{1}{2}}\right)$ marked by $\bullet$. The stencil for $p_{i+\frac{1}{2},j+\frac{1}{2}}$ includes $\partial_d p$ at the points indicated by $\square$ and $\pentagon$. At the points indicated by $\square$, only the derivative in the normal direction is approximated. At the points indicated by $\pentagon$, both the derivatives in the normal and the parallel directions are approximated. The derivative in the parallel directions $\overline{\partial_d p}$ are approximated by interpolation.

the form of $\kappa \nabla p$. Suppose that at a given point,

$$\kappa = \begin{bmatrix} \kappa_{11} & \kappa_{12} \\ \kappa_{12} & \kappa_{22} \end{bmatrix}, \tag{2.3.27}$$

then

$$\begin{bmatrix} \kappa_{11} & \kappa_{12} \\ \kappa_{12} & \kappa_{22} \end{bmatrix} \begin{bmatrix} \partial_x p \\ \partial_y p \end{bmatrix} = \begin{bmatrix} \kappa_{11}\partial_x p + \kappa_{12}\partial_y p \\ \kappa_{12}\partial_x p + \kappa_{22}\partial_y p \end{bmatrix}. \tag{2.3.28}$$

These calculations suggest that $\partial_x p$ and $\partial_y p$ must be approximated at each cell edge. In particular, for the direction perpendicular to the cell edge, simply approximate using the usual difference quotient. That is

$$\begin{aligned} \partial_x p|_{x_i, y_{j+1/2}} &\approx \partial_x p_{i,j+1/2} = \frac{p_{i+1/2,j+1/2} - p_{i-1/2,j+1/2}}{\Delta x_i}, \\ \partial_y p|_{x_{i+1/2}, y_j} &\approx \partial_y p_{i+1/2,j} = \frac{p_{i+1/2,j+1/2} - p_{i+1/2,j-1/2}}{\Delta y_j}. \end{aligned} \tag{2.3.29}$$

For the direction parallel to the cell edge, it is necessary to interpolate the derivative. This leads to

$$\begin{aligned} \partial_x p_{i+1/2,j} = & \frac{(\Delta x_{i+1} - \Delta x_{i-1})\,\Delta y_j}{(\Delta x_i + \Delta x_{i-1})\,(\Delta x_i + \Delta x_{i+1})\,(\Delta y_j + \Delta y_{j-1})} p_{i+1/2,j-1/2} \\ & + \frac{(\Delta x_{i+1} - \Delta x_{i-1})\,\Delta y_{j-1}}{(\Delta x_i + \Delta x_{i-1})\,(\Delta x_i + \Delta x_{i+1})\,(\Delta y_j + \Delta y_{j-1})} p_{i+1/2,j+1/2} \\ & + \frac{\Delta y_j}{(\Delta y_j + \Delta y_{j-1})\,(\Delta x_i + \Delta x_{i+1})} p_{i+3/2,j-1/2} \end{aligned}$$

$$
\begin{aligned}
&-\frac{\Delta y_j}{(\Delta y_j+\Delta y_{j-1})(\Delta x_i+\Delta x_{i-1})}p_{i-1/2,j-1/2}\\
&+\frac{\Delta y_{j-1}}{(\Delta y_j+\Delta y_{j-1})(\Delta x_i+\Delta x_{i+1})}p_{i+3/2,j+1/2}\\
&-\frac{\Delta y_{j-1}}{(\Delta y_j+\Delta y_{j-1})(\Delta x_i+\Delta x_{i-1})}p_{i-1/2,j+1/2}
\end{aligned}
\tag{2.3.30}
$$

and

$$
\begin{aligned}
\partial_y p_{i,j+1/2} \;=\;& \frac{\Delta x_i(\Delta y_{j+1}-\Delta y_{j-1})}{(\Delta y_j+\Delta y_{j-1})(\Delta y_j+\Delta y_{j+1})(\Delta x_i+\Delta x_{i-1})}p_{i-1/2,j+1/2}\\
&+\frac{\Delta x_{i-1}(\Delta y_{j+1}-\Delta y_{j-1})}{(\Delta y_j+\Delta y_{j-1})(\Delta y_j+\Delta y_{j+1})(\Delta x_i+\Delta x_{i-1})}p_{i+1/2,j+1/2}\\
&+\frac{\Delta x_i}{(\Delta x_i+\Delta x_{i-1})(\Delta y_j+\Delta y_{j+1})}p_{i-1/2,j+3/2}\\
&-\frac{\Delta x_i}{(\Delta x_i+\Delta x_{i-1})(\Delta y_j+\Delta y_{j-1})}p_{i-1/2,j-1/2}\\
&+\frac{\Delta x_{i-1}}{(\Delta x_i+\Delta x_{i-1})(\Delta y_j+\Delta y_{j+1})}p_{i+1/2,j+3/2}\\
&-\frac{\Delta x_{i-1}}{(\Delta x_i+\Delta x_{i-1})(\Delta y_j+\Delta y_{j-1})}p_{i+1/2,j-1/2}.
\end{aligned}
\tag{2.3.31}
$$

Finally, a single expression is necessary. Let $f_{i+1/2,j+1/2}$ be the value of f at the cell center, then

$$
\begin{aligned}
-\Delta x_i\Delta y_j f_{i+1/2,j+1/2} \;=\;& \Delta y_j\left[\kappa\nabla p\right]|_{x_{i+1},y_{j+1/2},1}-\Delta y_j\left[\kappa\nabla p\right]|_{x_i,y_{j+1/2},1}\\
&+\Delta x_i\left[\kappa\nabla p\right]|_{x_{i+1/2},y_{j+1},2}-\Delta x_i\left[\kappa\nabla p\right]|_{i+1/2,y_j,2}\\
=\;& \Delta y_j\left[\kappa_{11}\partial_x p+\kappa_{12}\partial_y p\right]|_{x_{i+1},y_{j+1/2}}\\
&-\Delta y_j\left[\kappa_{11}\partial_x p+\kappa_{12}\partial_y p\right]|_{x_i,y_{j+1/2}}\\
&+\Delta x_i\left[\kappa_{12}\partial_x p+\kappa_{22}\partial_y p\right]|_{x_{i+1/2},y_{j+1},2}
\end{aligned}
$$

$$-\Delta x_i \left[\kappa_{12}\partial_x p + \kappa_{22}\partial_y p\right]|_{i+1/2,y_j,2}. \tag{2.3.32}$$

In the case of a homogeneous κ and uniform grid, this results in

$$\begin{aligned} -\Delta x_i \Delta y_j f_{i+1/2,j+1/2} &= -4\kappa_{11} p_{i+1/2,j+1/2} \\ &\quad +\kappa_{11}\left(p_{i+3/2,j+1/2} + p_{i-1/2,j+1/2}\right) \\ &\quad +\kappa_{22}\left(p_{i+1/2,j+3/2} + p_{i+1/2,j-1/2}\right) \\ &\quad +\frac{\kappa_{12}}{2}\left(p_{i-1/2,j-1/2} + p_{i+3/2,j+3/2}\right) \\ &\quad -\frac{\kappa_{12}}{2}\left(p_{i+3/2,j-1/2} + p_{i-1/2,j+3/2}\right). \end{aligned} \tag{2.3.33}$$

It is shown in [2, 62] that the method constructed using cell-centered finite differences is equivalent, up the the error in numerical quadrature, to a mixed finite element method. The equivalence is demonstrated below.

As mentioned above, the regularity requirements on the data are reduced using the finite element method, with respect to the finite difference approaches.

2.3.2.2 Mixed Finite Element Methods

Finite element methods are numerical methods for solving partial differential equations, which are particularly useful for complicated domains with irregular boundaries, and for solutions with low regularity.

Similarly to the finite volume method, they decompose the domain into a collection of elements. In contrast to the finite volume method, finite element methods

treat the differential equation in weak form. On each element, a basis for a finite dimensional subspace of the test functions and the solutions is found and a system of equations is derived. This process is described in more detail below.

The most common type of finite element methods are Galerkin finite element methods, but in this work mixed finite element methods are of primary interest. Mixed finite element methods have the advantage of producing a conservative velocity [2, 10].

2.3.2.3 Elements and Function Spaces

Since the connection between finite element methods and finite difference methods is of interest, only rectangular elements will be considered. The following definition from [9] will be necessary.

Definition 2.26. A partition $\mathcal{T} = \{T_1, T_2, \ldots, T_M\}$ of Ω into triangular or quadrilateral elements is called *admissible* provided the following properties hold:

- $\overline{\Omega} = \bigcup_{i=1}^{M} T_i$.
- If $T_i \cap T_j$ consists of exactly one point, then it is a common vertex of T_i and T_j.
- If for $i \neq j$, $T_i \cap T_j$ consists of more than one point, then $T_i \cap T_j$ is a common edge of T_i and T_j.

When every element has diameter at most $2h$ it is convenient to write $\mathcal{T}_h$.

It is important in the present systems to approximate the space $H(\mathrm{div}; T)$ where T is an element. Since the elements are rectangles, the spaces $RT_{[k]}$ will be used. The spaces $RT_{[k]}$ are rectangular elements which approximate the solution with functions formed as tensor products of polynomials of degree $k+1$ in one direction combined with polynomials of lower degree in other directions. The space $RT_{[k]}$ is formally defined below.

The implementation in Listing 1 use $RT_{[0]}$, but it is useful to develop the theory in general. Let $\widehat{K} = (-1, 1)^2$ be the *reference element*. Then the following definitions are needed.

Definition 2.27. On the element K, define $P_k(K)$ to be the space of polynomials of degree less than or equal to k. Then define

$$P_{k_1,k_2}(K) = \left\{ p(x_1, x_2) : p(x_1, x_2) = \sum_{\substack{i \le k_1 \\ j \le k_2}} a_{ij} x_1^i x_2^j \right\}, \tag{2.3.34}$$

the space of polynomials of degree less than or equal to k_1 in x_1 and less than or equal to k_2 in x_2. Then define

$$Q_k(K) = P_{k,k}(K). \tag{2.3.35}$$

With those spaces in mind, define $RT_{[k]}$.

Definition 2.28. Let $RT_{[k]}$ be defined by

$$\begin{aligned} RT_{[k]} &= (Q_k)^2 + xQ_k, \\ &= P_{k+1,k} \times P_{k,k+1}, \end{aligned} \tag{2.3.36}$$

and

$$\dim RT_{[k]} = 2\,(k+1)\,(k+2)\,. \tag{2.3.37}$$

Moreover, for $q_k \in RT_{[k]}$,

$$\nabla \cdot q_k \in Q_k, \tag{2.3.38}$$

and

$$q \cdot n|_{e_i} \in P_k\,(e_i)\,, \tag{2.3.39}$$

where e_i is an edge. Also, let

$$RT^0_{[k]} = \left\{ q \in RT_{[0]} : \nabla \cdot q = 0 \right\}. \tag{2.3.40}$$

Note that $RT_{[0]}$ will allow the approximate solution (p_h, u_h) to satisfy

$$\begin{aligned} \|u - u_h\|_{H_{\text{div}}} &= O\,(h)\,, \\ \|p - p_h\|_{L^2} &= O\,(h)\,, \end{aligned}$$

where (p, u) is the exact solution.

In this work we do not study the approximation error between u and u_h, but rather focus on the modeling error and solution techniques.

2.3.2.4 Saddle Point Problems

Many problems of interest can be expressed in the form of a saddle point problem.

Definition 2.29. Let V be a Hilbert space, $A : V \to V'$ a continuous linear operator, $B : V \to Q'$ a linear operator and $B^T : Q \to V'$ it transpose. A *saddle point problem* has the form

$$\begin{aligned} Au + B^T p &= f, \quad \text{in } V', \\ Bu &= g, \quad \text{in } Q'. \end{aligned} \tag{2.3.41}$$

Problems of this form arise naturally in some applications. It is sometimes possible to eliminate the variable u and solve for p using the relation

$$g = BA^{-1}\left(f - B^T p\right).$$

The theory for saddle point problems with linear operators is developed in [10].

The classical method for solving saddle point problems is Uzawa's Algorithm. From [9], the basic algorithm and two methods of implementation are given. Suppose the system has been discretized so that $A \in \mathbf{R}^{n\times n}$, $B \in \mathbf{R}^{m\times n}$.

Algorithm 2.30 (Uzawa's Algorithm)**.** *[9] Let $p_0 \in \mathbf{R}^m$. Find u_k and p_k so that*

$$\begin{aligned} Au_k &= f - B^T p_{k-1}. \\ p_k &= p_{k-1} + \alpha\left(Bu_k - g\right). \end{aligned}$$

It is sufficient that $\alpha < 2\left\|BA^{-1}B^T\right\|^{-1}$.

To implement the method directly requires an extra multiplication by A^{-1} in every iteration than is necessary if the algorithm is implemented carefully.

Algorithm 2.31 (Gradient Based Uzawa's Algorithm)**.** *[9] Let* $p_0 \in \mathbf{R}^m$ *and* $Au_1 = f - B^T p_0$. *For* $k = 1, 2, \ldots,$ *compute*

$$\begin{aligned} q_k &= g - Bu_k, \\ \lambda_k &= B_k^T, \\ h_k &= A^{-1}\lambda_k, \\ \alpha_k &= \frac{q_k' q_k}{\lambda_k h_k}, \\ p_k &= p_{k-1} - \alpha_k q_k, \\ u_{k+1} &= u_k + \alpha_k h_k. \end{aligned}$$

In order to speed up convergence, it is often necessary to use conjugate directions.

Algorithm 2.32 (Uzawa's Algorithm with Conjugate Directions)**.** *[9] Let* $p_0 \in \mathbf{R}^m$ *and* $Au_1 = f - B^T p_0$. *Set* $d_1 = -q_1 = Bu_1 - g$. *For* $k = 1, 2, \ldots,$ *find*

$$\begin{aligned} \lambda_k &= B^T d_k, \\ h_k &= A^{-1} p_k, \\ \alpha_k &= \frac{q_k' q_k}{p_k' h_k}, \\ p_k &= p_{k-1} + \alpha_k d_k, \end{aligned}$$

$$\begin{aligned} u_{k+1} &= u_k - \alpha_k h_k, \\ q_{k+1} &= g - B u_{k+1}, \\ \beta_k &= \frac{q'_{k+1} q_{k+1}}{q'_k q_k}, \\ d_{k+1} &= -q_{k+1} + \beta_k d_k. \end{aligned}$$

Uzawa's algorithm often has slow convergence, so it is beneficial to try to express a model in a way that avoids the use of Uzawa's Algorithm.

Example. The following example illustrates the use of Uzawa's Algorithm.

Consider solving the problem

$$\begin{aligned} \kappa u + \nabla p &= G, \\ \nabla \cdot u &= F, \\ p|_{\partial\Omega} &= 0, \end{aligned}$$

on a cell-centered finite difference grid in $(0,1) \times (0,1)$. Let the grid be defined by $p_{i+jN} = p\left(\frac{i+1}{N+1}, \frac{j+1}{N+1}\right)$ for $i \in \{0, \ldots, N-1\}$ and $j \in \{0, \ldots, N-1\}$, $u_{i+j(N+1)} = u\left(\frac{i}{N}, \frac{j+1}{N+1}\right)$ for $i \in \{0, \ldots N\}$ and $j \in \{0, \ldots, N-1\}$, $u_{N(N+1)+i+jN} = u\left(\frac{i+1}{N+1}, \frac{j}{N}\right)$ for $i \in \{0, \ldots, N-1\}$ and $j \in \{0, \ldots, N-1\}$. Let $h = \frac{1}{N+1}$ be the grid spacing.

The gradient operator may be approximated by $\nabla p\left(\frac{i}{N}, \frac{j+1}{N+1}\right) \approx \frac{p_{i+jN} - p_{i-1+jN}}{h}$ and $\nabla p\left(\frac{i+1}{N+1}, \frac{j}{N}\right) \approx \frac{p_{i+jN} - p_{i+(j-1)N}}{h}$ in the interior of the domain. The operator ∇_h which represents the finite difference approximation of the gradient has dimension $2(N+1)N \times N^2$. Row $i + j(N+1)$ for $i \in \{1, \ldots N-1\}$ and $j \in \{0, \ldots, N-1\}$

corresponds to the gradient on the cell edge at $\left(\frac{i}{N}, \frac{j+1}{N+1}\right)$ and has entries $\frac{1}{h}$ and $-\frac{1}{h}$ in columns $i+jN$ and $i-1+jN$, respectively. Row $j\,(N+1)$ for $j \in \{0,\ldots,N-1\}$ corresponds to the gradient on the boundary and has entry $\frac{1}{h}$ in column jN. Row $N+j\,(N+1)$ for $j \in \{0,\ldots,N-1\}$ also corresponds to the gradient on the boundary and has entry $-\frac{1}{h}$ in column $N-1+jN$. Row $N\,(N+1)+i+jN$ for $i \in \{0,\ldots,N-1\}$ and $j \in \{1,\ldots,N-1\}$ corresponds to the gradient on the cell edge at $\left(\frac{i+1}{N+1}, \frac{j}{N}\right)$ and has entries $\frac{1}{h}$ and $-\frac{1}{h}$ in columns $i+jN$ and $i+(j-1)\,N$. Row $N\,(N+1)+i$ for $i \in \{0,\ldots,N-1\}$ corresponds to the gradient on the boundary and has entry $\frac{1}{h}$ in column i. Row $N\,(N+1)+i+N^2$ for $i \in \{0,\ldots,N-1\}$ also corresponds to the gradient on the boundary and has entry $-\frac{1}{h}$ in column $i+(N-1)\,N$.

The divergence operator may be approximated by

$$\begin{aligned}\mathrm{div}_h u\left(\frac{i}{N+1}, \frac{j}{N+1}\right) \approx & \frac{u_{i+j(N+1)} - u_{i-1+j(N+1)}}{h} \\ & + \frac{u_{N(N+1)+i+jN} - u_{N(N+1)+i+(j-1)N}}{h}.\end{aligned}$$

The operator div_h which represents the finite difference approximation of the divergence operator has dimension $N^2 \times 2\,(N+1)\,N$. Row $i+jN$ for $i \in \{0,\ldots,N-1\}$ and $j \in \{0,\ldots,N-1\}$ corresponds to the divergence at $\left(\frac{i+1}{N+1}, \frac{j+1}{N+1}\right)$ and has entries $\frac{1}{h}$ in columns $i+j\,(N+1)$ and $N\,(N+1)+i+jN$ as well as entries $-\frac{1}{h}$ in columns $i-1+j\,(N+1)$ and $N\,(N+1)+i+(j-1)\,N$.

Note that ∇_h and div_h are negative transposes of one another. This fact is

easily checked for the case $N = 2$. The operators are given by

$$\begin{aligned}
\nabla_h &= \frac{1}{h}\begin{bmatrix}
1 & & & & \\
-1 & 1 & & & \\
 & -1 & & & \\
 & & 1 & & \\
 & & -1 & 1 & \\
 & & & -1 & \\
1 & & & & \\
 & 1 & & & \\
-1 & & 1 & & \\
 & -1 & & 1 & \\
 & & -1 & & \\
 & & & -1 &
\end{bmatrix}, \\
\mathrm{div}_h &= -\nabla_h^T.
\end{aligned}$$

By a simple rearrangement, this discretization can be put into a form that may be solved using Uzawa's algorithm. Given N, let ∇_h and div_h be given as above. Find u_h, p_h that satisfy

$$\begin{aligned}
\kappa u_h + \nabla_h p_h &= G, \\
-\mathrm{div}_h u_h &= -F, \\
p_h|_{\partial\Omega} &= 0.
\end{aligned}$$

Algorithms 2.31 and 2.32 are implemented for this experiment in Listing 10. With $N = 10$ (i.e., with $10 \times 10 = 100$ pressure unknowns), the gradient based implementation takes 440 iterations and the conjugate gradient based implementation takes only 15 iterations. In what follows, the system is decoupled so that the slow convergence of Uzawa's algorithm may be avoided.

2.3.2.5 Approximation of Integrals and Relation to Cell Centered Finite Differences

Now we discuss how the mixed finite element method is related to the cell-centered finite difference implementation. Through this relationship we know that, even for not very regular solutions, the cell-centered approximations converge. In turn, the implementation is easier since the velocity unknowns can be easily eliminated. We follow known work cited below.

Since the integrals encountered in the formulation of a finite element method cannot be computed exactly, methods for approximating the integrals are necessary. Two one dimensional integration methods: the midpoint rule and the trapezoidal rule are useful [62, 2].

Definition 2.33. Let $f \in C^2[a,b]$ and let $h = b - a$, then there exists $\xi_T \in (a,b)$ such that

$$\int_a^b f(x)\,dx = \frac{h}{2}\left[f(a) + f(b)\right] - \frac{h^3}{12} f''(\xi_T). \tag{2.3.42}$$

The quantity $\frac{h}{2}\left[f(a) + f(b)\right]$ defines the *trapezoidal rule*. There exists $\xi_M \in (a,b)$

such that

$$\int_a^b f(x)\,dx = hf\left(\frac{a+b}{2}\right) + \frac{h^3}{3}f''(\xi_M)\,. \tag{2.3.43}$$

The quantity $hf\left(\frac{a+b}{2}\right)$ defines the *midpoint rule.*

For proofs see [12].

Consider solving

$$\begin{aligned} \int_\Omega \kappa u\cdot v - \int_\Omega p\nabla\cdot v &= \int_\Omega G\cdot v, & (2.3.44)\\ \int_\Omega w\nabla\cdot u &= \int_\Omega Fw, & (2.3.45) \end{aligned}$$

in $\Omega = (0,1)\times(0,1)$. This is a weak formulation of (2.3.5)-(2.3.6). Let $p_h = \sum p_{i,j}\xi_{i,j}$ where $\xi_{i,j}$ is a characteristic function for cell i,j and let

$$u_h|_{\Omega_{i,j}}(x,y) = \begin{pmatrix} \psi_{i-1/2}(x)\,u_{i-1/2} + \psi_{i+1/2}(x)\,u_{i+1/2} \\ \phi_{j-1/2}(y)\,u_{j-1/2} + \phi_{j+1/2}(y)\,u_{j+1/2} \end{pmatrix},$$

where $\psi_{i-1/2}$ is a basis polynomial of degree one supported only on $\Omega_{i-1,j}\cup\Omega_{i,j}$ and $\phi_{j-1/2}$ is a basis polynomial of degree one supported only on $\Omega_{i,j-1}\cup\Omega_{i,j}$. Using $w_h = \xi_{i,j}$, the discrete analog of (2.3.45) is

$$\begin{aligned} \Delta x_i\Delta y_j F_{i,j} &= \int_{\Omega_{i,j}} \nabla\cdot u \\ &= u_{i+1/2} - u_{i-1/2} + u_{j+1/2} - u_{j-1/2}. \end{aligned} \tag{2.3.46}$$

Next, the momentum equation will be treated using a combination of the midpoint

and trapezoidal rules.

First, the x-direction will be considered. Let $v_h = \left(\psi_{i+1/2,j}, \xi_{i+1/2,j}\right)$ be a test function that is linear in the x-direction and constant in the y-direction. Note that v_h is supported on $\Omega_{i,j} \cup \Omega_{i+1,j}$. The first term of (2.3.44) is approximated by

$$\begin{aligned}
\int_\Omega \kappa u \cdot v_h &= \int_{\Omega_{i,j}} \kappa u \cdot v_h + \int_{\Omega_{i+1,j}} \kappa u \cdot v_h \\
&= \int_{\Omega_{i,j}} \kappa_1 \left(\psi_{i-1/2}(x)\, u_{i-1/2} + \psi_{i+1/2}(x)\, u_{i+1/2}\right) \psi_{i+1/2,j} \\
&\quad + \int_{\Omega_{i,j}} \kappa_2 \left(\phi_{j-1/2}(y)\, u_{j-1/2} + \phi_{j+1/2}(y)\, u_{j+1/2}\right) \xi_{i+1/2,j} \\
&\quad + \int_{\Omega_{i+1,j}} \kappa_1 \left(\psi_{i-1/2}(x)\, u_{i-1/2} + \psi_{i+1/2}(x)\, u_{i+1/2}\right) \psi_{i+1/2,j} \\
&\quad + \int_{\Omega_{i+1,j}} \kappa_2 \left(\phi_{j-1/2}(y)\, u_{j-1/2} + \phi_{j+1/2}(y)\, u_{j+1/2}\right) \xi_{i+1/2,j} \\
&= \frac{\Delta x_i + \Delta x_{i+1}}{2} \Delta y_j \kappa_1 u_{i+1/2} + c\,(\Delta x)^3 (\Delta y)^3 . \qquad (2.3.47)
\end{aligned}$$

The second term of (2.3.44) is, exactly,

$$\begin{aligned}
\int_\Omega p \nabla \cdot v_h &= \int_{\Omega_{i,j}} p_{i,j} \psi_{i+1/2,j} + \int_{\Omega_{i+1,j}} p_{i+1,j} \psi_{i+1/2,j} \\
&= \Delta y_j \left(p_{i,j} - p_{i+1,j}\right).
\end{aligned}$$

The source term in (2.3.44) is approximated, in the same way as in (2.3.47), by

$$\int_\Omega G \cdot v_h = \frac{\Delta x_i + \Delta x_{i+1}}{2} \Delta y_j G_{i+1/2} + c\,(\Delta x)^3 (\Delta y)^3 .$$

This leads to a method where

$$u_{i+1/2} = 2\frac{\Delta y_j}{\Delta x_i + \Delta x_{i+1}}\kappa_1^{-1}\left(p_{i,j} - p_{i+1,j}\right) + \kappa_1^{-1}G_{i+1/2}. \tag{2.3.48}$$

In an analogous way, the y-direction may be approximated using

$$v_h = \left(\xi_{i,j} + \xi_{i,j+1}, \phi_{i,j+1/2}\right).$$

This decouples the saddle point problem since, if p_h were known, it would be possible to find u_h using those formulas. Substituting (2.3.48) into (2.3.46) yields

$$\begin{aligned}
\Delta x_i \Delta y_j F_{i,j} = & \; 2\frac{\Delta y_j}{\Delta x_i + \Delta x_{i+1}}\kappa_1^{-1}\left(p_{i,j} - p_{i+1,j}\right) \\
& -2\frac{\Delta y_j}{\Delta x_{i-1} + \Delta x_i}\kappa_1^{-1}\left(p_{i-1,j} - p_{i,j}\right) \\
& +2\frac{\Delta x_i}{\Delta y_j + \Delta y_{j+1}}\kappa_2^{-1}\left(p_{i,j} - p_{i,j+1}\right) \\
& -2\frac{\Delta x_i}{\Delta y_{j-1} + \Delta y_j}\kappa_2^{-1}\left(p_{i,j-1} - p_{i,j}\right) \\
& +\kappa_1^{-1}\left(G_{i+1/2} - G_{i-1/2}\right) + \kappa_2^{-1}\left(G_{j+1/2} - G_{j-1/2}\right),
\end{aligned}$$

which can be rearranged so that it has the form of a cell-centered finite difference method similar to Section 2.3.2.1.

In Section 2.3.2.1, $\left(x_{i+\frac{1}{2}}, y_{j+\frac{1}{2}}\right)$ was the cell-center where p was approximated while in this section (x_i, y_j) is the cell-center where p is approximated. This difference is only a difference in notation and does not reflect a difference in the approximation.

2.3.2.6 Mixed Finite Element Method for the Linear Flow Model

The interpretation of $RT[0]$ solutions provided in Section 2.3.2.3 above is very useful for linear problem with diagonal κ which does not depend on u. For our nonlinear flow problem, we encounter a full tensor, and thus follow different work [2]. Another approach is considered in [27].

In [2], anisotropic linear flow problems are considered. A method is proposed which has an auxiliary velocity variable. In particular, $u_h \in V_h$, $\overline{u}_h \in \widehat{V}_h$, and $p_h \in W_h$ are sought which solve

$$(\nabla \cdot u_h, w) = (f, w), \qquad \forall w \in W_h, \tag{2.3.49}$$

$$(\overline{u}_h, v) = (p_h, \nabla \cdot v) - (p_D, v \cdot n)_{\Gamma^D}, \qquad \forall v \in V_h^0, \tag{2.3.50}$$

$$(\kappa u_h, \overline{v}) = (\overline{u}_h, \overline{v}) + (g, \overline{v}), \qquad \forall \overline{v} \in \widehat{V}_h, \tag{2.3.51}$$

$$(u_h \cdot n, \mu)_{\Gamma^N} = (u_N, \mu)_{\Gamma^N}, \qquad \forall \mu \in \Lambda_h. \tag{2.3.52}$$

The following conditions are required for the theorems from [2]:

(C1) Given $f \in L^2$, p_D, and u_N, there exists a unique solution $p \in H^2(\Omega)$ such that

$$\|p\|_2 \;\leq\; C\left[\|f\|_0 + \|p_D\|_{3/2,\Gamma^D} + \|u_N\|_{1/2,\Gamma^N}\right]$$

where C depends on Ω, κ, and u_N,

(C2) $\nabla \cdot V_h = W_h$,

(C3) $V_h \cdot n|_{\partial\Omega} = \Lambda_h$,

(C4) $V_h^N \subset \widehat{V_h}$,

(C5) κ is uniformly positive definite in Ω.

Those conditions are used in several theorems.

Theorem 2.34 (Theorem 3.1 from [2]). *Assume (C1)-(C5). Let*

$$\begin{aligned} A_h &= \|u_h\|_0 + \|\overline{u}_h\|_0 + \|u_h \cdot n\|_{0,\Gamma^N} + \|p_h\|_0 \\ &\quad + \|\sqrt{u_N}\lambda_h\|_{0,\Gamma^N} + \|\lambda_h\|_{-1/2,\Gamma^N} \\ C_D &= \|f\|_0 + \|p_D\|_{1/2,\Gamma^D} + \|u_N\|_{1/2,\Gamma^N}. \end{aligned}$$

If $(u_h, \overline{u}_h, p_h, \lambda_h)$ *is a solution of* (2.3.49)-(2.3.52)*, then*

$$\begin{aligned} \|\nabla \cdot u_h\|_0 &\leq \|f\|_0, \\ A_h &\leq CC_D \end{aligned}$$

where C *depends on* Ω, $\|\kappa\|_{1,\infty}$, *and* $\|u_N\|_{0,\infty,\Gamma^N}$.

Corollary 2.35 (Corollary 3.2 from [2]). *Assume (C1)-(C5). There exists a unique solution to* (2.3.49)-(2.3.52).

Theorem 2.36. *[Theorem 3.3 from [2]] Assume that (C1)-(C5) are satisfied. Let* l *be the degree of polynomials in* W_h, k *be the degree of polynomials in* V_h, *and* m *be the degree of polynomials in* Λ_h. *There exists a constant* C *independent of* h

and dependent on Ω, p, u, $\|K\|_{0,\infty}$, *and* $\|u_N\|_{0,\infty,\Gamma^N}$ *such that*

$$\begin{aligned} \|u-u_h\|_0 + \|\overline{u}-\overline{u}_h\|_0 + \|\sqrt{u_N}\,(\lambda-\lambda_h)\|_{0,\Gamma^N} &\leq Ch^j, \\ \|\nabla\cdot(u-u_h)\|_{-s} &\leq Ch^{j+s}, \\ \|(u-u_h)\cdot n\|_{0,\Gamma^N} &\leq Ch^j, \end{aligned}$$

where $1 \leq j \leq \min(k,m)$, $0 \leq s \leq l$, *and* $0 \leq j \leq l$. *If* $0 \leq s \leq \min(k,l,m)-1$, Ω *is* $(s+2)$ *regular, and* C *depends also on* $\|\kappa\|_{s+1,\infty}$ *and* $\|u_N\|_{s+2,\infty,\Gamma^N}$, *then for any* $0 \leq j \leq \min(k,l,m)$,

$$\begin{aligned} \|p-p_h\|_{-s} &\leq Ch^{j+s}, \\ \|\lambda-\lambda_h\|_{-s-1/2,\Gamma^N} &\leq Ch^{j+s+1/2}. \end{aligned}$$

With those theorems established, a cell-centered finite difference stencil is derived for the pressure using the lowest-order RTN spaces on rectangles. Note that in this case, $k=l=m=1$ in Theorem 2.36.

Corollary 2.37. *If the conditions of Theorem 2.36 are met, then on the lowest-order RTN spaces on rectangles the following estimates hold*

$$\begin{aligned} \|u-u_h\|_0 + \|\overline{u}-\overline{u}_h\|_0 + \|\sqrt{u_N}\,(\lambda-\lambda_h)\|_{0,\Gamma^N} &\leq Ch, \\ \|\nabla\cdot(u-u_h)\|_0 &\leq Ch, \\ \|(u-u_h)\cdot n\|_{0,\Gamma^N} &\leq Ch, \end{aligned}$$

and

$$\begin{aligned} \|p - p_h\|_0 &\leq Ch, \\ \|\lambda - \lambda_h\|_{-1/2,\Gamma^N} &\leq Ch^{3/2}. \end{aligned}$$

On an element $E \in \mathcal{T}_h$, let

$$\begin{aligned} V_h(E) &= \left\{(\alpha_1 x_1 + \beta_1, \alpha_2 x_2 + \beta_2)^T : \alpha_i, \beta_i \in \mathbf{R}\right\} \\ W_h(E) &= \{\alpha : \alpha \in \mathbf{R}\} \end{aligned}$$

and on an edge or face e,

$$\Lambda_h(e) = \{\alpha : \alpha \in \mathbf{R}\}$$

for $d = 2$. The standard nodal basis is used, where for V_h, Λ_h the nodes are at the midpoints of the edges of the elements and for W_h the nodes are at the midpoints of the elements (cell centers). Also choose $\hat{V}_h = V_h$.

Appropriate quadrature rules are used to set up a system of equations which can be solved. The solutions $u, \overline{u} \in V_h$ and $p \in W_h$ are sought such that

$$\begin{aligned} (\nabla \cdot u, w) &= (f, w) && w \in W_h \\ (\overline{u}, v)_{TM} &= (p, \nabla \cdot v) - (p_D, v \cdot n)_{\Gamma^D} && v \in V_h \\ (\kappa u, \overline{v})_{TM} &= (\overline{u}, \overline{v})_T + (g, \overline{v})_T && \overline{v} \in V_h \\ (u \cdot n, \mu)_{\Gamma^N} &= (u_N, \mu)_{\Gamma^N} && \mu \in \Lambda_h^N \end{aligned}$$

where $(\cdot,\cdot)_M$ and $(\cdot,\cdot)_T$ mean application of the midpoint and trapezoidal rules, respectively, and for $v, q \in \mathbf{R}^d$,

$$(v,q)_{TM} \quad = \quad (v_1,q_1)_{T\times M} + (v_2,q_2)_{M\times T}$$

That is, one computes the integral by applying the trapezoidal rule in the ith direction and the midpoint rule in the other direction.

In summary, the method used here is similar to the one applied in [62]. The main advantage is that off-diagonal terms are allowed in [2], while [62] is only applicable if the tensor is diagonal. It has been shown that the stencil for the method developed in [2] reduces to the stencil used in [62] when both are applicable.

In this work we implemented the approach in this section keeping track of κ dependent on the unknown velocity u_h. The implementation in Listing 1 allows for the use of anisotropy, but the examples in this work will only use diagonal terms in the tensor κ.

2.3.2.7 Newton Iteration for the Nonlinear Flow Model

With the above method for solving the linear problem, the non-linear problem will be solved using a general Newton-Krylov solver from Scipy [34, 5, 37] which expects to have a function to compute the residual. A method for computing the residual is presented in Algorithm 2.38. See [35] for a description of the convergence theory for such solvers.

Algorithm 2.38 (Newton-Krylov Residual)**.** *Given p, do the following:*

- *Find u so that* $\kappa(|u|)\,u = -\nabla_h p$.
- *Find the residual* $\nabla_h \cdot u$.

The Newton-Krylov iteration is then computed with an inexact line search.

2.4 Coupled Flow and Transport

Once the velocity is known from the flow model, it may be of interest to consider the transport of a solute in the fluid. Let $c(x,t)$ denote the concentration of the solute at position x and time t. It is assumed that the presence of the solute does not affect the flow. In general, the *conservation of mass* for the solute is written

$$\partial_t(\phi c) + \nabla\cdot(cu) + \nabla\cdot F \;=\; s(x,t), \tag{2.4.1}$$

where u is taken from the flow model (2.3.5)-(2.3.8). The term cu is the advective flux and represents transport due to movement of the fluid. The term F is the diffusive flux and represents transport due to diffusion.

Define F using Fick's First Law of diffusion[19]:

$$F \;=\; -D\nabla c, \tag{2.4.2}$$

where D is the diffusion/dispersion tensor. As usual, it is assumed that $D = D(x)$ is a symmetric non-negative definite constant tensor.

If $\phi(x,t) = 1$, and inserting Fick's First Law of diffusion into 2.4.1, then

$$\partial_t c + \nabla \cdot (cu) = \nabla \cdot (D \nabla c) + s(x,t). \tag{2.4.3}$$

To close the system of equations, initial and boundary conditions are needed:

$$c(x,0) = c_0(x), \qquad x \in \Omega, \tag{2.4.4}$$

$$c(x,t) = c_I(x,t), \qquad x \in \Gamma_I, \tag{2.4.5}$$

$$D \nabla c \cdot n = 0, \qquad x \in \Gamma_O, \tag{2.4.6}$$

where

$$\Gamma_I = \{x \in \Gamma : u \cdot n < 0\}, \tag{2.4.7}$$

$$\Gamma_O = \{x \in \Gamma : u \cdot n \geq 0\}, \tag{2.4.8}$$

that is, the inflow and outflow parts of the boundary, respectively.

It remains to show that the model is well-posed. There are two situations to consider when verifying the well-posedness of the transport model. First, if $D \equiv 0$ then there is no diffusion and the problem is hyperbolic. In that case the method of characteristics may be applied. If $D \neq 0$, then there is diffusion and the problem is parabolic. In that case, the results in [38] will be applied.

2.4.1 Well-Posedness If $D \equiv 0$

For simplicity of exposition, the source term $s(x,t)$ is omitted and it is assumed that $\nabla \cdot u = 0$. The two dimensional case is considered. In the case that $D \equiv 0$, (2.4.3) becomes

$$\partial_t c + \nabla \cdot (cu) = 0.$$

Using the product rule, this is equivalent, since $\nabla \cdot u = 0$, to

$$\partial_t c + u \cdot \nabla c = 0.$$

This may be written less compactly as

$$\partial_t c + u_1 \partial_x c + u_2 \partial_y c = 0$$

where $u = (u_1, u_2)^T$. It may be verified that if $x(t)$ and $y(t)$ satisfy

$$\begin{aligned} \frac{dx}{dt} &= u_1(x(t), y(t)), \\ \frac{dy}{dt} &= u_2(x(t), y(t)), \end{aligned}$$

then $\frac{d}{dt} c(x(t), y(t)) = 0$. So if u_1 and u_2 are Lipschitz continuous in x and y, then there are unique solutions $x(t)$ and $y(t)$ for any initial data $x(0) = x_0$ and $y(0) = y_0$. This may be shown using Picard's existence theorem. We call the curve

$(x(t), y(t))$ a *characteristic* of c.

In order to find the value $c(x_0, y_0, t_0)$ for $t_0 > 0$ it is sufficient to solve the differential equations

$$\begin{aligned} \frac{dx}{dt} &= u_1(x(t), y(t)), \\ \frac{dy}{dt} &= u_2(x(t), y(t)), \end{aligned}$$

backwards in time with $x(t_0) = x_0$ and $y(t_0) = y_0$ so that $c(x(0), y(0), 0)$ may be evaluated. If the characteristic passes through the boundary at some $t_1 > 0$, then the value of c at that boundary point is used.

2.4.2 Well Posedness If $D \neq 0$

In Chapter III of [38], models of the form

$$\mathcal{L}u \equiv \partial_t c - \mathcal{M}c = \partial_{x_i} f_i - f, \tag{2.4.9}$$

where

$$\mathcal{M}c = \sum_{i=1}^{d} \left[\partial_{x_i} \left[a_{ij}(x,t)\, \partial_{x_i} c + a_i(x,t)\, c\right] - b_i(x,t)\, \partial_{x_i} c - a(x,t)\, c\right], \tag{2.4.10}$$

are studied. In the present case, $D = (a_{ij})$ is the diffusive term, $a_i = 0$, $b_i = u_i$ and $a = 0$. The model is said to be *parabolic* if there exist constants $\nu, \mu > 0$ such

that

$$\nu \sum_{i=1}^{d} \xi_i^2 \leq \sum_{i=1}^{d} \sum_{j=1}^{d} a_{ij}(x,t)\,\xi_i \xi_j \leq \mu \sum_{i=1}^{d} \xi_i^2 \tag{2.4.11}$$

for any $\xi \in \mathbf{R}^d$. Since D is diagonal with positive entries, the problem is parabolic. The following conditions will be needed.

(P1) There exists μ_1 such that $\|u\|_{L^2(\Omega)} \leq \mu_1$.

(P2) The source terms have finite norms, that is there exists a constant μ_2 such that

$$\begin{aligned} \|f\|^2_{2,\Omega\times(0,T)} &= \int_{\Omega\times(0,T)} \sum_{i=1}^{d} f_i^2 dx dt \leq \mu_2, \\ \|f\|_{q_1,r_1,\Omega\times(0,T)} &= \left(\int_0^T \|f\|^r_{L^q(\Omega)}\, dt \right)^{1/r} \leq \mu_2. \end{aligned}$$

where q_1 and r_1 satisfy

$$\begin{aligned} \frac{1}{r_1} + \frac{1}{q_1} &= \frac{3}{2}, \\ q_1 \in (1,2], \quad \text{and} \quad & r_1 \in [1,2), \end{aligned}$$

The following uniqueness theorem may be applied.

Theorem 2.39 (Uniqueness for Parabolic Models [38, III.3.1])**.** *If the coefficients of* (2.4.9) *satisfy (P1) and (P2), then the first boundary value problem for* (2.4.9) *cannot have two distinct solutions.*

Also the following existence theorem.

Theorem 2.40 (Existence for Parabolic Models [38, III.5.1])**.** *If (P1) and (P2) are satisfied and* $\psi_0(x) \in L^2(\Omega)$*, then the problem*

$$\begin{aligned} \mathcal{L}c &= -f, \\ \partial_n u + \sigma(x,t)\, c|_{\partial\Omega\times(0,T)} &= \psi(x,t), \\ u|_{t=0} &= \psi_0(x), \end{aligned}$$

has a solution if there exists $\mu_2 > 0$ *such that*

$$\begin{aligned} \|\sigma\|_{q_2,r_2,\partial\Omega\times(0,T)} &= \left(\int_0^T \|\sigma\|^{r_2}_{L^{q_2}(\partial\Omega)}\, dt\right)^{1/r_2} \leq \mu_2, \\ \|\psi\|_{q_2,r_2,\partial\Omega\times(0,T)} &= \left(\int_0^T \|\psi\|^{r_2}_{L^{q_2}(\partial\Omega)}\, dt\right)^{1/r_2} \leq \mu_2, \end{aligned}$$

where q_2 *and* r_2 *are subject to*

$$\begin{aligned} \frac{1}{r_2} + \frac{1}{2q_2} &= \frac{1}{2}, \\ q_2 \in (1,\infty] \quad &\textit{and} \quad r_1 \in [2,\infty). \end{aligned}$$

2.4.3 Numerical Methods for Initial Value Problems

Initial value problems arise naturally for time dependent systems. Below, the finite volume method for a diffusive transport system will be introduced. The finite volume method leads to a discretization in space which will require the solution of

an initial value problem in time.

Definition 2.41. An *initial value problem (IVP)* takes the form

$$u'(t) = f(u(t), t), \qquad \text{for } t > t_0, \tag{2.4.12}$$

$$u(t_0) = u_0. \tag{2.4.13}$$

In general, u may be a vector with n components. In that case, $f(u,t)$ will also have n components.

Next, the theory needed for the existence and uniqueness of a solution is developed.

Theorem 2.42. *[40] If f is Lipschitz continuous over some region $\mathcal{D}$ then there is a unique solution to the IVP at least up to time $T^* = \min\left(t_1, t_0 + \frac{a}{S}\right)$, where*

$$S = \max_{(u,t)\in\mathcal{D}} \|f(u,t)\|.$$

See [40] for several numerical methods for the IVP. The two simplest methods are are called the forward and backward Euler methods, respectively.

Definition 2.43. [40] The *forward Euler* and *backward Euler* methods are given by

$$U_{n+1} = U_n + \Delta t f(U_n), \tag{2.4.14}$$

$$U_{n+1} = U_n + \Delta t f(U_{n+1}), \tag{2.4.15}$$

respectively, where Δt is the time step. Both of these methods are first order accurate.

There are several important methods, which will not be treated in this work, that have higher accuracy. Some of these methods have been carefully implemented in standard numerical libraries such as Scipy [34] and will be used for numerical experiments. Runge-Kutta methods [29] and backward difference formulas [11, 13, 32, 33] are algorithms that may be used.

2.4.4 The Finite Volume Method

The finite volume method is used to approximate the solution of a conservation law. As the name suggests, the domain Ω is decomposed into a collection of volumes. With that decomposition complete, the conservation law is discretized. One of the primary advantages of the finite volume method is that it is conservative. This section closely follows [39] in the treatment of first order methods.

2.4.4.1 Conservation Laws

A conservation law has the form

$$\partial_t c + \nabla \cdot F = s\left(x, t\right), \tag{2.4.16}$$

where F, called the flux, may take many forms. Only F that are linear in c and ∇c are considered. In particular, let

$$F(c, \nabla c) = cu - D\nabla c, \tag{2.4.17}$$

where $u = u(x)$ is a vector valued function and $D = D(x)$ is a symmetric positive definite tensor. For simplicity, take D to be diagonal. In this case, call cu and $D\nabla c$ the advective part and diffusive part of the flux function, respectively.

Let V be an arbitrary region where the conservation law is satisfied and that c and F are sufficiently smooth to apply the divergence theorem. Then,

$$\begin{aligned} \frac{\partial}{\partial t}\int_V c dx &= -\int_V \nabla \cdot F dx \\ &= -\int_{\partial V} F \cdot n dx. \end{aligned} \tag{2.4.18}$$

This result indicates that the integral on the left side changes only by considering the flux on the boundary. This is the basis for the finite volume method which consists of finding a suitable discretization of this equation.

2.4.4.2 Discretization

In space, the region Ω is discretized into rectangles. That is, let $x_0 < x_1 < \cdots < x_{N_x}$ and $y_0 < y_1 < \cdots < y_{N_y}$, then let $V_{i,j} = (x_i, x_{i+1}) \times (y_j, y_{j+1})$. The discretization is based on taking c to be a constant on each volume at each discrete time. Denote by $C_{i,j}^n$ the value of c in $V_{i,j}$ at time t_n. Time is discretized into

$t_0 < t_1 < \cdots < t_{N_t}$.

There are several difficulties in discretizing the flux function. It is simple to find discretizations which are consistent with the equations, but it is more difficult to find stable discretizations. For the advective part of the flux function, an upwind flux will be used (and described below). The advective part may be treated explicitly in time, but may also be treated implicitly. For the diffusive part of the flux function, approximate ∇c at the cell boundaries, apply D, then take the discrete divergence. The diffusive part will be treated implicitly in time in order to avoid a restrictive time step.

When the advective part is treated explicitly in time, the discretization will be

$$\Delta x \Delta y C_{i,j}^{n+1} - \Delta t D_{i,j}^{n+1} = \Delta x \Delta y C_{i,j}^{n} - \Delta t A_{i,j}^{n} + \Delta t s_{i,j}^{n^*}, \qquad (2.4.19)$$

where $A_{i,j}^n = \left[\int_{V_{i,j}} \nabla \cdot (cu)\right]_{i,j}^{n}$ and $D_{i,j}^n = \left[\int_{V_{i,j}} \nabla \cdot (D \nabla c)\right]_{i,j}^{n+1}$ are discretizations of the advective and diffusive parts, respectively. When the advective part is treated implicitly in time, the discretization will be

$$\Delta x \Delta y C_{i,j}^{n+1} + \Delta t D_{i,j}^{n+1} - \Delta t A_{i,j}^{n+1} = \Delta x \Delta y C_{i,j}^{n} + \Delta t s_{i,j}^{n^*}. \qquad (2.4.20)$$

In either case, the source term s, may be evaluated at t_n or t_{n+1}. In the absence of diffusion, it is very advantageous to treat the advective term explicitly since it is unnecessary to solve a linear system.

If there is diffusion, then there is a need to solve a linear system regardless of how the advective term is handled.

2.4.4.3 Discretization of the Advective Part

A possible discretization of the advection part of the flux is discussed below. First,

$$\begin{aligned}\int_{V_{i,j}} \nabla \cdot (cu) &= \int_{\partial V_{i,j}} cu \cdot n \\ &= \Delta y_j \left((cu)_{i-1/2,j} - (cu)_{i+1/2,j} \right) \\ &\quad + \Delta x_i \left((cu)_{i,j-1/2} - (cu)_{i,j+1/2} \right). \end{aligned} \tag{2.4.21}$$

The difficulty is that C is not continuous at $x_{i-1/2,j}$, $x_{i+1/2,j}$, $y_{i,j-1/2}$, and $y_{i,j+1/2}$ so the formula cannot be applied directly. Call the quantity cu the flux which needs to be approximated. Let $F_{i-1/2,j}$ and $F_{i,j-1/2}$ denote the approximation to $(cu)_{i-1/2,j}$ and $(cu)_{i,j-1/2}$, respectively. It is convenient to introduce the notation

$$v^+ = \max(v, 0), \tag{2.4.22}$$

$$v^- = \min(v, 0), \tag{2.4.23}$$

for a quantity v.

Consider the donor-cell upwind method [39]. This method is simple to implement and is stable, but there is a restriction on the time step.

First, we provide a description of the method. The flux at the left edge of the volume is an approximation to the amount flowing from cell $V_{i-1,j}$ to $V_{i,j}$ (or vice versa). If $u_{i-1/2,j} > 0$, then the flow is from $V_{i-1,j}$ into $V_{i,j}$ and so the amount flowing in during $(t, t + \Delta t)$ is $\Delta t u_{i-1/2,j} C_{i-1,j}$. The description is similar if $u_{i-1/2,j} < 0$, except that the flow goes from $V_{i,j}$ into $V_{i-1,j}$.

Formally, the flux function is given by

$$F_{i-1/2,j} = u^{+}_{1,i-1/2,j} C_{i-1,j} + u^{-}_{1,i-1/2,j} C_{i,j}, \tag{2.4.24}$$

$$F_{i,j-1/2} = u^{+}_{2,i,j-1/2} C_{i,j-1} + u^{-}_{2,i,j-1/2} C_{i,j}. \tag{2.4.25}$$

The method is stable as long as

$$\left| \frac{u_1 \Delta t}{\Delta x} \right| + \left| \frac{u_2 \Delta t}{\Delta y} \right| \leq 1. \tag{2.4.26}$$

Let $h = \min(\Delta x, \Delta y)$, $U_1 = \max_{i,j} |u_{1,i,j}|$, and $U_2 = \max_{i,j} |u_{2,i,j}|$. It is sufficient to choose Δt so that

$$\Delta t \leq \frac{h}{U_1 + U_2}.$$

There are other methods which have less restrictive stability requirements than the one above, but they are more difficult to implement.

2.4.4.4 Discretization of the Diffusive Part

Proceeding as in the discretization of the advective part,

$$\begin{aligned} \int_{V_{i,j}} \nabla \cdot (D \nabla c) &= \int_{\partial V_{i,j}} D \nabla c \cdot n \\ &= \Delta y_j \left[D \nabla c \right]_{i-1/2,j} - \Delta y_j \left[D \nabla c \right]_{i+1/2,j} \\ &\quad + \Delta x_i \left[D \nabla c \right]_{i,j-1/2} - \Delta x_i \left[D \nabla c \right]_{i,j+1/2}. \end{aligned} \tag{2.4.27}$$

Then approximate

$$\partial_x c_{i-1/2,j} \approx \frac{C_{i,j} - C_{i-1,j}}{\frac{1}{2}(\Delta x_{i-1} + \Delta x_i)}, \tag{2.4.28}$$

$$\partial_y c_{i,j-1/2} \approx \frac{C_{i,j} - C_{i,j-1}}{\frac{1}{2}(\Delta y_{j-1} + \Delta y_j)}. \tag{2.4.29}$$

Then apply D at each cell edge to the computed components of the gradient.

The method shown here is very similar to the cell-centered method we defined for the flow.

One significant difficulty in discretizing the diffusive part is the temptation to use an explicit scheme. An explicit scheme can be used, but it results in the very restrictive time step $\Delta t \leq \alpha \Delta x^2$ [12]. If the diffusive term is treated implicitly, then there is no restriction in the time step [12].

2.5 Summary

In this chapter, the background needed to continue with sensitivity analysis and its applications have been developed. More general notions of derivatives and spaces of functions with derivatives were developed in Section 2.1. Theorem 2.9, the Implicit Function Theorem, will be applied in the next chapter to show the existence of sensitivities.

In Section 2.3 a nonlinear flow model was developed and analyzed. First, the system was set up, then in Section 2.3.1, the system was shown to be well posed. The well-posedness relies on a recently published theorem. Next, some related

models are discussed in Section 2.3.1.1. The present model is different since it does not include time dependent velocities. Finally, in Section 2.3.2, a numerical algorithm for the solution to the nonlinear flow model was developed. The algorithm may be analyzed as a mixed finite element method, but is implemented using cell-centered finite difference methods.

Finally, in Section 2.4 a coupled transport model was developed. The well-posedness without and with diffusion were treated in Section 2.4.1 and Section 2.4.2, respectively. Without diffusion, the standard method of characteristics may be applied, while with diffusion a theorem from [38] is applied. The time discretization is described in Section 2.4.3 and the spatial discretization, based on the finite volume method, is described in Section 2.4.4.

Chapter 3 Sensitivity Analysis

In this work sensitivity analysis is understood as finding derivatives of the solution to a model, or a quantity of interest derived from the solution, with respect to model parameters. Other notions of sensitivity analysis, and other techniques have been explored in literature.

One class of methods is based on changing the model input parameters one at a time [63, 51, 20]. Another class of methods is based on automatic differentiation, where software tracks not only the solution, but the derivative of the solution as well [28]. Finally, a probabilistic notion of sensitivity considers the variance of the solution in relation to the model parameters [18, 64].

The derivatives with respect to model parameters, which are called sensitivities, give information about how the model responds to changes in the parameters. As we mentioned before, there are delicate issues related to this differentiation, and we explore some below. In the remainder of this work we will assume that the sensitivities we compute are well-defined as solutions to the PDEs, or their weak formulations. We will, however, not prove that they are indeed well defined as the derivatives of the (possibly weak) solutions to the underlying PDEs.

The information on the derivatives may be useful in areas such as model reduction where some parameters may be considered constant if the model is not sensitive to them [30]. Sensitivities may also be of interest since they show the need

to have precise values for a parameter. Examples and applications for sensitivity analysis will be developed below.

In this chapter the general theory for sensitivity analysis is developed. In Section 3.1, some results about the existence of sensitivities are presented. In Section 3.2 the sensitivity equation is developed, then two formulations for sensitivity analysis are given: the forward sensitivity analysis (FS) in Section 3.3 and the adjoint sensitivity analysis (AS) in Section 3.4.

FS will compute the sensitivity of the solution explicitly while AS will set up an adjoint problem in order to avoid that computation. FS is best applied when there are many quantities of interest or few parameters.

In turn, AS is best applied when there are few quantities of interest or many parameters. When the AS methods are applied to the coupled transport model in Section 3.4.2, the sensitivity of the velocity appears in the adjoint system.

The main contribution from this Chapter is that a method is developed, based on the AS method for the flow model, which avoids the explicit calculation of the sensitivity of the velocity.

A further difficulty with the AS transport model is that it is posed backwards-in-time. One other contributions of this work is that we define a method so that a transport solver designed for forward-in-time problems can be used for backwards-in-time problems, thus eliminating the need for special solvers.

3.1 Existence of Sensitivities

The theory for the well-posedness of a model usually guarantees that the solution will depend continuously on the model parameters. This is not, of course, sufficient to guarantee the existence of sensitivities. In Section 2.1.1.2, the Implicit Function Theorem, Theorem 2.9, will guarantee us the weak existence of sensitivities. This version of the Implicit Function Theorem is set in abstract function spaces and may be difficult to apply in some situations.

In some situations it is possible to apply the following theorem.

Theorem 3.1 (Strong Existence of Sensitivities [70]). *If $\bar{f}_{\alpha_1,\ldots,\alpha_j}(x,t)$ possesses kth order continuous derivatives in the $n+j+1$ variables $\alpha_1,\ldots,\alpha_j,t,x_1,\ldots x_n$ then the solution of the initial value problem*

$$\begin{aligned}\frac{dx}{dt} =& \bar{f}_{\alpha_1,\ldots,\alpha_j}(x,t)\\ x(0) =& c = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}\end{aligned}$$

possesses kth order continuous derivatives in the $n+j$ variables $\alpha_1,\ldots,\alpha_j,c_1,\ldots,c_n$. Furthermore, the derivative $\frac{dx}{dt}$ possesses kth order continuous derivatives in the $n+j+1$ variables $\alpha_1,\ldots,\alpha_j,t,c_1,\ldots,c_n$.

With those theorems in mind, methods for computing the sensitivity of a quan-

tity of interest can be developed.

3.2 Sensitivity Equation

Consider a differential equation of the form

$$F\left(\pi; x, t, u, \partial_t u, \partial_x u, \partial_{xx} u, \dots\right) = 0, \tag{3.2.1}$$

for $x \in \mathbf{R}^d$, $t > 0$, and $\pi \in \mathbf{R}^{N_\pi}$ a vector of parameters. Let $G(\pi; u)$ be a quantity of interest. In order to find $\left\{\frac{dG}{d\pi}\right\}$, the vector containing the derivatives of G with respect to each parameter in π, it is natural to use the chain rule:

$$\left\{\frac{dG}{d\pi}\right\} = \{\partial_\pi G\} + \partial_u G \{\partial_\pi u\}. \tag{3.2.2}$$

Since G is a given function, it is possible to compute $\{\partial_\pi G\}$ and $\partial_u G$. If F satisfies one of the existence theorems above, then $\{\partial_\pi u\}$ exists.

The goal is to derive methods to compute $\{\partial_\pi u\}$ (in FS) or to find an alternative formulation to avoid the computation of the sensitivity (in AS). In particular, in FS, equation (3.2.2) is used just as it is written while in AS, the explicit computation of $\{\partial_\pi u\}$ is avoided by solving an adjoint problem.

3.3 Forward Sensitivity Analysis

Forward sensitivity analysis (FS) explicitly computes the sensitivity of the solution to the parameter. The sensitivity may be computed using either a continuous form or a discretized form. In the continuous form, the sensitivity equation is derived and then solved. In the discretized form, the solver is augmented to produce the sensitivity automatically using techniques such as automatic differentiation. Only the continuous form will be considered presently.

Continuing the example from Section 3.2, it is possible to formally differentiate (3.2.1) to obtain

$$\begin{aligned} 0 = \frac{dF}{d\pi} &= \partial_u F \left\{\partial_\pi u\right\} + \left[\partial_{\partial_t u} F\right] \left[\partial_t \left\{\partial_\pi u\right\}\right] \\ &\quad + \left[\partial_{\partial_x u} F\right] \left[\partial_x \left\{\partial_\pi u\right\}\right] + \left[\partial_{\{\partial_{xx} u\}} F\right] \left[\partial_{xx} \left\{\partial_\pi u\right\}\right] + \cdots \end{aligned}$$

which is a differential equation for $\{\partial_\pi u\}$. The differential equation, regardless of the form of F, is linear since all of the derivatives of F are evaluated at the known solution u. This indicates that finding the sensitivity $\{\partial_\pi u\}$ may be easier than solving for u. With that example in mind, the sensitivity equations for the examples from Chapter 2 may be developed. In Section 2.2, the equations for the sensitivities were already found in equations (2.2.5) and (2.2.6) by applying implicit differentiation.

3.3.1 Sensitivity for Fluid Flow in Porous Media

Recall that the strong form of the nonlinear flow model is shown in (2.3.5)-(2.3.8). The possibility of a vector of parameters π in κ is denoted by $\kappa = \kappa(\pi; |u|)$. The system becomes

$$\nabla \cdot u = f(x), \qquad \text{in } \Omega, \tag{3.3.1}$$

$$\kappa(\pi; |u|)\, u = -\nabla p + g(x), \qquad \text{in } \Omega, \tag{3.3.2}$$

$$p = p_D(x), \qquad \text{on } \Gamma^D, \tag{3.3.3}$$

$$u \cdot n = u_N(x), \qquad \text{on } \Gamma^N. \tag{3.3.4}$$

We define the sensitivity equation for (3.3.1)-(3.3.4) by, formally, differentiating each equation to arrive at

$$\nabla \cdot \{\partial_\pi u\} = 0, \qquad \text{in } \Omega, \tag{3.3.5}$$

$$\left[\kappa + \partial_{|u|}\kappa\left(\partial_u |u|\,(u)\right)\right]\{\partial_\pi u\} = -\nabla\{\partial_\pi p\} - \{\partial_\pi \kappa\}\, u, \qquad \text{in } \Omega, \tag{3.3.6}$$

$$\{\partial_\pi p\} = 0, \qquad \text{on } \Gamma^D, \tag{3.3.7}$$

$$\{\partial_\pi u\} \cdot n = 0, \qquad \text{on } \Gamma^N. \tag{3.3.8}$$

It is important to note that the resistance term in equation (3.3.6) depends on u, but not on $\{\partial_\pi u\}$.

Thus, the sensitivity system (3.3.5)-(3.3.8), is a linear flow problem even if the original problem is nonlinear.

It is also worth noting that the quantity $\{\partial_\pi u\}$ contains components for the derivatives of each component of the velocity with respect to each parameter. Each parameter may be considered separately; this results in solving a linear flow problem for each parameter.

To clarify what is meant by equation (3.3.6), the following specific example is given. Let

$$\kappa\left(r_1, r_2, \beta; u\right) = \begin{bmatrix} r_1 & \\ & r_2 \end{bmatrix} + \beta \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix}$$

where $r_1 > 0$, $r_2 > 0$, and $\beta \geq 0$ are parameters. It is of interest to find the sensitivity with respect to the parameter β, namely $\{\partial_\beta u\}$ and $\{\partial_\beta p\}$. The following derivatives are needed:

$$\begin{aligned} \{\partial_\beta \kappa\} &= \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix}, \\ \partial_{|u|}\kappa &= \beta, \\ \partial_u |u| &= \left[\begin{array}{c|c} \mathrm{sgn} u_1 & \\ \hline & \mathrm{sgn} u_2 \end{array}\right]. \end{aligned}$$

Next, the derivative $\partial_u |u|$ is a tensor which, for our purposes, maps vectors to matrices. Then (3.3.6) becomes

$$\left[\begin{bmatrix} r_1 & \\ & r_2 \end{bmatrix} + 2\beta \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix}\right] \{\partial_\beta u\} = -\nabla \{\partial_\beta p\} - \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix} u. \quad (3.3.9)$$

The gravity-like term

$$-\begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix} u,$$

on the right hand side of (3.3.9), is a known quantity. The equation is linear since the resistance term does not depend on $\{\partial_\beta u\}$ or $\{\partial_\beta p\}$.

In [57], the sensitivity of a related nonlinear flow model was explored in one space dimension.

3.3.2 Coupled Flow and Transport

Recall that the strong form of the transport equation is given in (2.4.3)-(2.4.8). It will be of particular interest to consider the sensitivity of the transport equation to parameters in the flow model. Suppose that u depends on the parameter π. Differentiating equation (2.4.3) with respect to π results in

$$\partial_t \{\partial_\pi c\} + \nabla \cdot (\{\partial_\pi c\} u + c \{\partial_\pi u\}) = \nabla \cdot (D \nabla \{\partial_\pi c\}). \tag{3.3.10}$$

The term $\nabla \cdot (c \{\partial_\pi u\})$ is a source term which may be computed once $\{\partial_\pi u\}$ and c are known. The form of the sensitivity equation is essentially identical to the original diffusive transport equation. Upon differentiation with respect to π, the boundary conditions (2.4.4)-(2.4.6) become

$$\{\partial_\pi c\}(x, 0) = 0, \qquad x \in \Omega, \tag{3.3.11}$$

$$\{\partial_\pi c\}(x,t) = 0, \qquad x \in \Gamma_I, \tag{3.3.12}$$

$$D\nabla\{\partial_\pi c\} \cdot n = 0, \qquad x \in \Gamma_O. \tag{3.3.13}$$

In [67], the existence of sensitivities of this type are discussed as well.

3.4 Adjoint Sensitivity Analysis

Adjoint sensitivity analysis (AS) is a method to compute the sensitivity of some quantities of interest by developing an adjoint problem. The quantity of interest is some function of the solutions to the underlying problem, e.g., flow velocities found from the nonlinear flow problem. In [61] AS is used for many quantities of interest. In [8] it is shown how it is possible to reuse quantities which are used in error estimates to compute the sensitivity of quantities of interest.

The idea in AS is to introduce an adjoint variable, differentiate the system with the Lagrange multiplier, rearrange the terms, integrate by parts, and finally solve the adjoint system. The AS method is dependent on the particular choice of the quantity of interest. In this work we consider linear functionals defined as an integral of the solution with some kernel. We note that, in general, for time-dependent problems, AS often results in backward-in-time problems.

Recall that the vector containing derivatives with respect to the parameters in π is given in equation (3.2.2) and that the differential equation satisfies (3.2.1), that is $F = 0$. If a Lagrange multiplier λ is introduced, it will be possible to avoid

the intermediate computation of $\{\partial_\pi u\}$. Let

$$I = G - \int \lambda^* F,$$

and note that $I = G$ since $F = 0$.

The method is to differentiate I, rearrange the terms, and integrate by parts. In this work we develop AS for the coupled flow and transport model.

3.4.1 AS for Fluid Flow in Porous Media

In this section we develop AS for the nonlinear flow problem.

Assume ξ is given, and define an example of a quantity of interest which may be of interest for a flow problem

$$H(u) = \int_\Omega \xi(x)\, u(x)\, dx + \int_{\partial\Omega} \xi_\partial(x)\, u(x) \cdot n dx,$$

where $\xi(x)$ is a given vector-valued weight function defined in Ω and $\xi_\partial(x)$ is a given scalar-valued weight function defined on $\partial\Omega$.

The goal is to compute $\{\partial_\pi H\}$, without having to calculate $\{\partial_\pi u\}$ and $\{\partial_\pi p\}$.

Recall the sensitivity equation is given by (3.3.5)-(3.3.8). In particular, note that

$$\int_\Omega \mu \nabla \cdot \{\partial_\pi u\}\, dx = 0,$$

$$\int_{\Omega} v\left[\left\{\partial_{\pi}\kappa\right\}u+\partial_{|u|}\kappa\left(\partial_{u}\left|u\right|\left(u\right)\right)\left\{\partial_{\pi}u\right\}+\kappa\left\{\partial_{\pi}u\right\}+\nabla\left\{\partial_{\pi}p\right\}\right]dx \;=\; 0,$$

for any functions $\mu\in H^{1}\left(\Omega\right)$ and $v\in H\left(\mathrm{div};\Omega\right)$.

Then we show how to calculate further

$$\begin{aligned}
\left\{\partial_{\pi}H\right\} &= \int_{\Omega}\xi\left\{\partial_{\pi}u\right\}dx+\int_{\partial\Omega}\xi_{\partial}\left\{\partial_{\pi}u\right\}\cdot ndx+\int_{\Omega}\mu\nabla\cdot\left\{\partial_{\pi}u\right\}dx\\
&\quad-\int_{\Omega}v\left[\left\{\partial_{\pi}\kappa\right\}u+\kappa\left(\pi;\left|u\right|\right)\left\{\partial_{\pi}u\right\}+\nabla\left\{\partial_{\pi}p\right\}\right]dx\\
&\quad-\int_{\Omega}v\left[\partial_{|u|}\kappa\left(\partial_{u}\left|u\right|\left(\left\{\partial_{\pi}u\right\}\right)\right)u\right]dx\\
&= \int_{\Omega}\left[\xi-\nabla\mu-\partial_{|u|}\kappa\left(\partial_{u}\left|u\right|\left(u\right)\right)v-\kappa\left(\pi;\left|u\right|\right)v\right]\left\{\partial_{\pi}u\right\}dx\\
&\quad+\int_{\Omega}\left\{\partial_{\pi}p\right\}\nabla\cdot vdx-\int_{\Omega}\left\{\partial_{\pi}\kappa\right\}uvdx\\
&\quad+\int_{\partial\Omega}\left[\mu+\xi_{\partial}\right]\left\{\partial_{\pi}u\right\}\cdot ndx-\int_{\partial\Omega}\left\{\partial_{\pi}p\right\}v\cdot ndx,
\end{aligned}$$

Recall that the goal is to avoid the computation of $\left\{\partial_{\pi}u\right\}$ and $\left\{\partial_{\pi}p\right\}$. If

$$\left[\partial_{|u|}\kappa\left(\partial_{u}\left|u\right|\left(u\right)\right)+\kappa\left(\pi;\left|u\right|\right)\right]v \;=\; -\nabla\mu+\xi, \tag{3.4.1}$$

$$\nabla\cdot v \;=\; 0, \tag{3.4.2}$$

with the boundary conditions

$$\mu|_{\Gamma_{D}} \;=\; -\xi_{\partial}, \tag{3.4.3}$$

$$v\cdot n|_{\Gamma_{N}} \;=\; 0, \tag{3.4.4}$$

then the integrals involving $\{\partial_\pi u\}$ and $\{\partial_\pi p\}$ will be zero. In that case,

$$\{\partial_\pi H\} \;=\; -\int_\Omega \{\partial_\pi \kappa\}\, uv dx. \tag{3.4.5}$$

The adjoint flow system (3.3.5)-(3.3.8) can be viewed as a linear flow model and can be solved using the same flow solver as the system (2.3.5)-(2.3.8).

Also, note that the resistance function for the AS system in equation (3.4.1) is the same as for the FS system in equation (3.3.6). Based on the preceding, the following theorem holds.

Theorem 3.2. *Suppose that u, p satisfy* (2.3.5)-(2.3.8) *where κ is dependent upon some parameter π. Let*

$$H(u) = \int_\Omega \xi(x)\, u(x)\, dx + \int_{\partial\Omega} \xi_\partial(x)\, u(x) \cdot n dx$$

be a quantity of interest. The sensitivity of H to the parameter π is given by (3.4.5)*, where v satisfies* (3.4.1)-(3.4.4).

3.4.2 Coupled Flow and Transport

Now consider the coupled problem of flow and transport; the latter solved for c. In a direct application of the AS method, one would find that, to find the sensitivity of some quantity of interest $G(c)$, we would have to solve the transport problem backward in time, and solve the flow problem repeatedly. The technique developed below allows us to avoid that.

Assume ψ is given, and the following quantity of interest for the coupled flow and transport model is defined as follows

$$G(c) = \int_{\Omega}\int_{0}^{T} \psi(x,t)\, c(x,t)\, dtdx, \tag{3.4.6}$$

where $\psi(x,t)$ is a weight function smooth enough that (3.4.6) is well defined.

Recall that the sensitivity equation for the coupled transport system is given in (3.3.10)-(3.3.13). In particular, note that

$$\int_{\Omega}\int_{0}^{T} \left[\partial_t \{\partial_\pi c\} + \nabla\cdot(\{\partial_\pi c\}\, u + c\,\{\partial_\pi u\}) - \nabla\cdot(D\nabla\{\partial_\pi c\})\right] q dtdx = 0 \tag{3.4.7}$$

for any q. Upon differentiation by π and adding the left hand side of equation (3.4.7), it can be seen that the sensitivity $\{\partial_\pi G\}$ is given by

$$\begin{aligned}
\{\partial_\pi G\} &= \int_{\Omega}\int_{0}^{T} \psi\,\{\partial_\pi c\}\, dtdx \\
&\quad - \int_{\Omega}\int_{0}^{T} \left[\partial_t\{\partial_\pi c\} - \nabla\cdot(D\nabla\{\partial_\pi c\})\right] q dtdx \\
&\quad - \int_{\Omega}\int_{0}^{T} \left[\nabla\cdot(\{\partial_\pi c\}\, u + c\,\{\partial_\pi u\})\right] q dtdx.
\end{aligned}$$

Integration by parts yields

$$\begin{aligned}
\{\partial_\pi G\} &= \int_{\Omega}\int_{0}^{T} \left[\psi + \partial_t q - u\cdot\nabla q - \nabla\cdot(D\nabla q)\right]\{\partial_\pi c\}\, dtdx \\
&\quad - \int_{0}^{T}\int_{\Omega} c\,\{\partial_\pi u\}\cdot\nabla q dxdt - \int_{\Omega}\left[q\,\{\partial_\pi c\}|_{t=0}^{T}\right] dx
\end{aligned}$$

$$+\int_0^T\int_{\partial\Omega}\left[qc\left\{\partial_\pi u\right\}+q\left\{\partial_\pi c\right\}u\right]\cdot n dxdt$$
$$+\int_0^T\int_{\partial\Omega}\left[\left\{\partial_\pi c\right\}D\nabla q-qD\nabla\left\{\partial_\pi c\right\}\right]\cdot n dxdt.$$

Further integration by parts yields

$$\begin{aligned}\left\{\partial_\pi G\right\} &= \int_\Omega\int_0^T\left[\psi+\partial_t q+\nabla\cdot(D\nabla q)+u\cdot\nabla q\right]\left\{\partial_\pi c\right\}dtdx\\ &+\int_0^T\int_\Omega c\left\{\partial_\pi u\right\}\cdot\nabla q dxdt-\int_0^T\int_{\partial\Omega}qc\left\{\partial_\pi u\right\}\cdot n dxdt\\ &+\int_0^T\int_{\Gamma_I}q\left(D\nabla\left\{\partial_\pi c\right\}\right)\cdot n dxdt\\ &-\int_0^T\int_{\Gamma_O}\left\{\partial_\pi c\right\}\left[D\nabla q-qu\right]\cdot n dxdt\\ &-\int_\Omega q\left(T\right)\left\{\partial_\pi c\right\}\left(T\right)dx.\end{aligned}$$

Recall that the goal is to eliminate any term which includes the sensitivity to the parameter π. If the adjoint variable q satisfies

$$\partial_t q+u\cdot\nabla q=-\nabla\cdot(D\nabla q)-\psi,\qquad x\in\Omega,\tag{3.4.8}$$
$$(D\nabla q-qu)\cdot n=0,\qquad x\in\Gamma_O,\tag{3.4.9}$$
$$q=0,\qquad x\in\Gamma_I,\tag{3.4.10}$$
$$q\left(T\right)=0,\tag{3.4.11}$$

then the sensitivity to the parameter π is eliminated and the simple expression

$$\{\partial_\pi G\} = \int_0^T \int_\Omega c\{\partial_\pi u\} \cdot \nabla q dx dt - \int_0^T \int_{\Gamma_O} qc\{\partial_\pi u\} \cdot n dx dt$$

is found.

Remark 3.3. The adjoint system (3.4.8)-(3.4.11) is a transport equation which is posed backward-in-time. The boundary conditions, in this case, are different than those posed for the transport problem (2.4.3)-(2.4.8). Note that, since (3.4.8)-(3.4.11) is solved backward-in-time, Γ_I acts as the outflow boundary and Γ_O acts as the inflow boundary. Since the outflow boundary is prescribed Dirichlet conditions, we expect that a boundary layer may form. On the outflow boundary, a Robin type boundary condition must be satisfied.

Often, the diffusive effects are ignored. In that case, the following remark will be helpful.

Remark 3.4. If $D \equiv 0$, then some simplifications occur. In that case,

$$\begin{aligned}\{\partial_\pi G\} = & \int_\Omega \int_0^T [\psi + \partial_t q + u \cdot \nabla q] \{\partial_\pi c\} dt dx \\ & + \int_0^T \int_\Omega c\{\partial_\pi u\} \cdot \nabla q dx dt - \int_0^T \int_{\Gamma_O} qc\{\partial_\pi u\} \cdot n dx dt \\ & + \int_0^T \int_{\Gamma_O} \{\partial_\pi c\} qu \cdot n dx dt - \int_\Omega q(T) \{\partial_\pi c\}(T) dx\end{aligned}$$

so it is only necessary to impose

$$\partial_t q + u \cdot \nabla q = -\psi, \tag{3.4.12}$$

$$q|_{\Gamma_O} = 0, \tag{3.4.13}$$

$$q(T) = 0, \tag{3.4.14}$$

which is still a backward-in-time problem.

At first, it seems that the computation of $\{\partial_\pi u\}$ cannot be avoided. It is, however, possible to use the method developed in Section 3.4.1 in order to avoid that computation. First, note that

$$\{\partial_\pi G\} = -\int_\Omega \{\partial_\pi u\} \cdot \left[\int_0^T c\nabla q dt\right] dx - \int_{\Gamma_O} \left(\int_0^T cq dt\right) \{\partial_\pi u\} \cdot n dx.$$

Given ψ, it is possible to solve the adjoint transport problem (3.4.8)-(3.4.11) or (3.4.12)-(3.4.14) for q. Since c has been computed, it is possible to form

$$\xi(x) = -\int_0^T c\nabla q dt$$

for $x \in \Omega$ and

$$\xi_\partial(x) = -\int_0^T cq dx$$

for $x \in \partial\Omega$. With ξ and ξ_∂ computed, it is possible to solve the adjoint flow problem (3.4.1)-(3.4.4) for μ and v. Finally, using Theorem 3.2,

$$\{\partial_\pi G\} = -\int_\Omega \{\partial_\pi \kappa\} uv dx.$$

This avoids the explicit computation of $\{\partial_\pi u\}$ and $\{\partial_\pi c\}$.

In summary, the following theorem holds.

Theorem 3.5. *Suppose that u, p satisfy* (2.3.5)-(2.3.8) *where κ is dependent upon some parameter π. Further, suppose that c satisfies* (2.4.3)-(2.4.8). *Let*

$$G(c) = \int_{\Omega} \int_{0}^{T} \psi(x,t)\, c(x,t)\, dtdx,$$

be a quantity of interest. Let q satisfy (3.4.8)-(3.4.11) *or* (3.4.12)-(3.4.14). *Let*

$$\xi(x) = -\int_{0}^{T} c\nabla q dt \tag{3.4.15}$$

for $x \in \Omega$ and

$$\xi_{\partial}(x) = -\int_{0}^{T} cq dx \tag{3.4.16}$$

for $x \in \partial\Omega$. Then the sensitivity of G to the parameter π is given by

$$\{\partial_{\pi} G\} = -\int_{\Omega} \{\partial_{\pi}\kappa\}\, uv dx, \tag{3.4.17}$$

where v satisfies (3.4.1)-(3.4.4).

Remark 3.6. It is possible to use a solver designed to operate forward-in-time, without modification, to solve these backward-in-time systems. Let $\tau = T - t$, then $\partial_{\tau} q = -\partial_t q$. The adjoint transport problem (3.4.8)-(3.4.11) may be rewritten as

$$\partial_{\tau}\vartheta - u \cdot \nabla\vartheta = \nabla \cdot (D\nabla\vartheta) + \psi, \qquad x \in \Omega, \tau \in [0,T] \tag{3.4.18}$$

$$(D\nabla\vartheta - \vartheta u)\cdot n = 0, \qquad x \in \Gamma_O, \tag{3.4.19}$$

$$\vartheta = 0, \qquad x \in \Gamma_I, \tag{3.4.20}$$

$$\vartheta(0) = 0, \tag{3.4.21}$$

where $q(t) = \vartheta(T-t)$. The adjoint transport problem without diffusion (3.4.12)-(3.4.14) may be rewritten in the same way as

$$\partial_t\vartheta - u\cdot\nabla\vartheta = \psi, \tag{3.4.22}$$

$$\vartheta|_{\Gamma_O} = 0, \tag{3.4.23}$$

$$\vartheta(0) = 0, \tag{3.4.24}$$

where $q(t) = \vartheta(T-t)$. Note that in both cases, u has the opposite sign as in (3.3.10)-(3.3.13).

3.5 Summary

In this chapter the general theory for sensitivity analysis was developed. In Section 3.1 some theorems guaranteeing the existence of sensitivities were presented. In Section 3.2, the sensitivity equation was defined and discussed. Two methods for computing the sensitivity were developed: forward sensitivity (FS) and adjoint sensitivity (AS).

Forward sensitivity analysis explicitly computes the solution to the sensitivity equation. In Section 3.3.1, FS was applied to the nonlinear flow model from Section

2.3. Also in Section 3.3.1, the form of the derivative of the resistance operator was discussed. In Section 3.3.2, FS was applied to the coupled transport model.

Adjoint sensitivity analysis solves an adjoint problem to avoid the explicit solution to the sensitivity equation. This has some advantages that will become clear in the next chapter. One advantage is that regardless of the number of parameters for which the sensitivity is to be computed, only one linear system needs to be solved. In Section 3.4.1, AS was applied to the nonlinear flow model from Section 2.3. It was shown that the solution to an adjoint problem, that has the form of a linear flow problem, may be used to compute the sensitivity of a quantity of interest. In Section 3.4.2, AS was applied to the coupled transport model. It was shown that the sensitivity of a quantity of interest may be found by solving a backwards-in-time transport model and using the results for the AS flow problem.

Chapter 4 Examples of Sensitivity Analysis

In this chapter, examples of the numerical computation of the sensitivity of the model problems presented in Chapter 2 are presented using the techniques developed in Chapter 3. The examples are designed to show the strength of our approaches, and are overviewed first.

In each example, a flow and transport problem is posed and solved, and the sensitivity of given quantities of interest are found. The flow problem is governed by equations (2.3.5)-(2.3.8) and the transport problem is governed by (2.4.3)-(2.4.8). Computation of the sensitivity using FS will rely on equations (3.3.5)-(3.3.8) for the flow problem and (3.3.10)-(3.3.13) for the transport problem. Computation of the sensitivity using AS will rely on equations (3.4.1)-(3.4.5) for the flow problem and equations (3.4.8)-(3.4.11) or (3.4.12)-(3.4.14) for the transport problem.

In Section 4.2, a pressure driven flow with constant parameters is discussed. The general form for the solutions in one space dimension is developed. In Section 4.3, a pressure driven flow with smooth parameters is discussed. Since the numerical methods do not yield the exact solution to the flow system, this problem gives an opportunity to show the order of convergence. In Section 4.4, a pressure driven flow with discontinuous parameters is discussed. The discontinuity in the problem presents some difficulty since some derivatives only exist in the sense of distributions. In the FS method, the distribution must be approximated numeri-

cally. One advantage of the AS method is that the distribution does not need to be approximated.

4.1 Setup of examples

Flow problem: boundary conditions and numerics Flows from the west side ($x = 0$) to the east side ($x = 1$) of $(0,1) \times (0,1)$ will be considered. On the west and east sides, Dirichlet boundary conditions are prescribed and on the north and south sides, Neumann no-flow boundary conditions are prescribed. The pressure on the west side is prescribed to be greater than the pressure on the east side. The flow and the sensitivity will have analytic solutions which will make it possible to show the performance of the numerical method. The numerical solution to each flow problem is found using the solver presented in Listing 1 in the Appendix with the tolerance for the Newton iteration set to 10^{-11} and with the initial guess for the pressure and velocity the solution to the linear (Darcy) system with $\kappa(\pi; 0)$.

For the flow problems, two quantities of interest are considered. Let $|\Omega|$ represent the area of Ω. Let

$$H_1(u) = \int_\Omega \xi_1(x)\, u(x)\, dx$$

where $\xi_1(x) = \left[\frac{1}{|\Omega|}, 0\right]^T$ and

$$H_2(u) = \int_\Omega \xi_2(x)\, u(x)\, dx$$

where $\xi_2(x) = \left[0, \frac{1}{|\Omega|}\right]^T$. These quantities of interest correspond, respectively, to

the average value of u_1 and u_2 in Ω. For the domains we are interested in, $|\Omega| = 1$.

Recall from Chapter 3, that FS sensitivity system (3.3.5)-(3.3.8) and the AS sensitivity system (3.4.1)-(3.4.4) for the flow problem will have a resistance term of the form

$$\kappa_{\text{sens}} = \left[\kappa + \partial_{|u|}\kappa\left(\partial_u |u|\,(u)\right)\right].$$

One component of that resistance term is $\{\partial_u |u|\}\, u$. For all of the following experiments, let

$$|u| = \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix}.$$

The well-posedness of problems involving that form of $|u|$ was discussed in Chapter 2. It was computed, in Section 3.3.1, that

$$\partial_u |u| = \left[\begin{array}{c|c} \text{sgn}u_1 & \\ \hline & \text{sgn}u_2 \end{array}\right].$$

Then

$$\partial_u |u|\,(u) = \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix} = |u|\,.$$

That term will appear in each section below.

The FS flow system (3.3.5)-(3.3.8) has a gravity term that depends on the parameter π of the form

$$g_\pi = -\{\partial_\pi \kappa\}\, u,$$

while the AS flow systems (3.4.1)-(3.4.4), has a gravity-like term that depends on

the weight function in the quantity of interest

$$g = \xi_i.$$

Transport problem: data an numerics In the transport problems, the boundary and initial conditions are given by

$$\begin{aligned} c(x,0) &= 0, \\ c(0,x_2,t) &= 1, \\ D &\equiv 0. \end{aligned}$$

Note that $\{x = (x_1, x_2) : x_1 = 0\}$ is the inflow boundary. The velocity will be taken from the associated non-linear non-Darcy flow problem.

The numerical solution to the transport problem is found using the solver presented in Listing 2 in the Appendix.

For the transport problems, one quantity of interest is considered. Let $\psi(x,t) = \frac{1}{T|\Omega|}$, where $|\Omega|$ is the area of the domain Ω, as above, and T is the final time. The quantity of interest

$$G(c) = \int_\Omega \int_0^T \psi(x,t)\, c(x,t)\, dtdx$$

represents the average value of c in the domain Ω over the interval $[0,T]$.

Error in Numerical Sensitivities In this chapter, the error in the numerical approximation of several sensitivities will be computed. By error, we mean the

absolute value of the difference between the numerically approximated value and the analytic sensitivity. Since the quantities of interest are defined by integrals over the domain, their range is the real numbers. This means that the sensitivity of the quantity of interest is a real number as well.

4.2 West to East With Constant Parameters

In this example, a pressure driven flow with constant coefficients from the west side to the east side of $[0,1]\times[0,1]$ is considered. An analytic solution is provided. Only two parameters are considered and the analytic sensitivity is shown for both. The analytic solution to the associated transport system is also shown and its sensitivity to the parameters in the flow problem is computed. The experiment is implemented in Listing 6 in the Appendix.

Let

$$\kappa\left(k,\beta;|u|\right)=\begin{bmatrix}\frac{1}{k}+\beta\left|u_1\right| & \\ & \frac{1}{k}+\beta\left|u_2\right|\end{bmatrix},$$

$$f\left(x\right)\equiv 0,$$

$$g\left(x\right)\equiv 0,$$

$$p_D\left(x\right)=1-x_1,$$

$$\Gamma_D=\left\{x:x_1=0 \text{ or } x_1=1\right\},$$

$$u_N\left(x\right)=0,$$

$$\Gamma_N=\left\{x:x_2=0 \text{ or } x_2=1\right\},$$

where $k > 0$ and $\beta \geq 0$. Consider the one dimensional system

$$\begin{aligned} \frac{u}{k} + \beta |u| u &= -\partial_x p, && (4.2.1) \\ \partial_x u &= 0, && (4.2.2) \\ p(0) = 1, \qquad & p(1) = 0, && (4.2.3) \end{aligned}$$

where $k > 0$ and $\beta \geq 0$. The system (4.2.1)-(4.2.3) has solution

$$\begin{aligned} p(x) &= 1 - x, \\ u(x) &= \begin{cases} k, & \beta = 0, \\ \frac{-1+\sqrt{1+4\beta k^2}}{2\beta k}, & \beta \neq 0. \end{cases} \end{aligned}$$

A simple substitution will verify that

$$\begin{aligned} p(x_1, x_2) &= 1 - x_1 \\ u_1(x_1, x_2) &= \begin{cases} k, & \beta = 0, \\ \frac{-1+\sqrt{1+4\beta k^2}}{2\beta k}, & \beta \neq 0, \end{cases} \\ u_2(x_1, x_2) &= 0, \end{aligned}$$

also solves the two dimensional problem, that is, the one dimensional problem is the first component of the solution to the two dimensional problem. The other

flow examples below will have similar solutions. Note that

$$\begin{aligned} \lim_{\beta \to 0^+} u_1(x) &= \lim_{\beta \to 0^+} \frac{k}{\sqrt{1+4\beta k^2}} \\ &= k \end{aligned}$$

so the solution is continuous in β.

In the transport problem, let

$$\begin{aligned} c(x,0) &= 0, \\ c(0,x_2,t) &= 1, \\ D &\equiv 0. \end{aligned}$$

Note that $\{x = (x_1, x_2) : x_1 = 0\}$ is the inflow boundary. As in the flow problem, it is possible to treat this problem as a one dimensional problem. The first-order, linear, constant-coefficient equation has the solution

$$c(x,t) = \begin{cases} 1, & x_1 < u_1 t, \\ 0, & x_1 > u_1 t. \end{cases} \tag{4.2.4}$$

The quantity of interest $G(c)$ may be computed as

$$G(c) = \frac{1}{T} \int_0^T \int_\Omega c(x,t)\, dx dt$$

$$
\begin{aligned}
&= \frac{1}{T}\int_0^T \left[\begin{cases} u_1 t, & 0 \le t \le \frac{1}{u_1} \\ 1, & \frac{1}{u_1} < t \end{cases} \right] dt \\
&\quad \frac{1}{T}\left[\frac{1}{2u_1} + T - \frac{}{2u_1} \right] \\
&= \begin{cases} \frac{u_1 T}{2}, & 0 \le T \le \frac{1}{u_1}, \\ 1 - \frac{1}{2u_1 T}, & \frac{1}{u_1} < T. \end{cases}
\end{aligned} \tag{4.2.5}
$$

4.2.1 Flow Sensitivity

The sensitivity of the flow system is computed in three ways. First, the analytic solution is found which is used in computing the error in the other methods. Next, the FS flow system is solved. Finally, the AS flow system is solved.

The analytic sensitivity is computed by differentiating the solution given above:

$$
\begin{aligned}
\{\partial_\beta p\}(x) &= 0, \\
\{\partial_k p\}(x) &= 0, \\
\{\partial_\beta u_1\}(x) &= \begin{cases} -k^3, & \beta = 0, \\ \frac{\sqrt{1+4\beta k^2} - 2\beta k^2 - 1}{2\beta^2 k\sqrt{1+4\beta k^2}}, & \beta \neq 0, \end{cases} \\
\{\partial_k u_1\}(x) &= \begin{cases} 1, & \beta = 0, \\ \frac{\sqrt{1+4\beta k^2} - 1}{2\beta k^2\sqrt{1+4\beta k^2}}, & \beta \neq 0. \end{cases}
\end{aligned}
$$

It may easily be shown that the sensitivities of the velocity are continuous by

computing the limit as $\beta \to 0$:

$$\begin{aligned}
\lim_{\beta\to 0^+} \{\partial_\beta u_1\}(x) &= \lim_{\beta\to 0^+} \frac{k - k\sqrt{1+4\beta k^2}}{2\beta + 10\beta^2 k^2} \\
&= \lim_{\beta\to 0^+} \frac{-2k^3}{(2+20\beta k^2)\sqrt{1+4\beta k^2}} \\
&= -k^3, \\
\lim_{\beta\to 0^+} \{\partial_k u_1\}(x) &= \lim_{\beta\to 0^+} \frac{1}{1+6\beta k^2} \\
&= 1.
\end{aligned}$$

The analytic sensitivity is used to compute the error in the numerical results.

To compute the FS sensitivity, the resistance and source terms must be computed. Differentiation shows that

$$\begin{aligned}
\partial_{|u|}\kappa &= \begin{bmatrix} \beta & \\ & \beta \end{bmatrix}, \\
\{\partial_k \kappa\} &= \begin{bmatrix} -\frac{1}{k^2} & \\ & -\frac{1}{k^2} \end{bmatrix}, \\
\{\partial_\beta \kappa\} &= \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix}.
\end{aligned}$$

The resistance term for the sensitivity to k and β is

$$\kappa_{\text{sens}} = \begin{bmatrix} \frac{1}{k} + 2\beta\,|u_1| & \\ & \frac{1}{k} + 2\beta\,|u_2| \end{bmatrix}.$$

The source term for the sensitivity to k is

$$g_k = -\{\partial_k \kappa\} u = \begin{bmatrix} \frac{u_1}{k^2} \\ \\ \frac{u_2}{k^2} \end{bmatrix},$$

and the source term for the sensitivity to β is

$$g_\beta = -\{\partial_\beta \kappa\} u = -\begin{bmatrix} u_1 |u_1| \\ \\ u_2 |u_2| \end{bmatrix}.$$

The forward sensitivity is the solution to

$$\nabla \cdot \{\partial_\pi u\} = 0, \qquad \text{in } \Omega, \tag{4.2.6}$$

$$\kappa_{\text{sens}} \{\partial_\pi u\} = -\nabla \{\partial_\pi p\} + g_\pi, \qquad \text{in } \Omega, \tag{4.2.7}$$

$$\{\partial_\pi p\} = 0, \qquad \text{on } \Gamma^D, \tag{4.2.8}$$

$$\{\partial_\pi u\} \cdot n = 0, \qquad \text{on } \Gamma^N. \tag{4.2.9}$$

Numerical results for the FS flow are in Table 4.2.1. Recall that the error is the absolute value of the difference between the numerical approximation of the sensitivity and the analytic sensitivity computed above. In all of the experiments, the error is smaller than the tolerance for the flow solver with zero error for several entries. This is a product of the simplicity of the system and should not be expected for more complicated systems.

For each quantity of interest with its corresponding weight function ξ, the AS

Table 4.2.1: West to East Flow With Constant Parameters: Error in FS Flow

	Forward Sensitivity Error			
	$\{\partial_k H_1\}$	$\{\partial_k H_2\}$	$\{\partial_\beta H_1\}$	$\{\partial_\beta H_2\}$
$\left\{ \begin{array}{l} k = 1.0 \\ \beta = 1.0 \end{array} \right\}$	$0.00\times10^{+00}$	1.33×10^{-15}	5.55×10^{-17}	3.78×10^{-28}
$\left\{ \begin{array}{l} k = 1.0 \\ \beta = 0.0 \end{array} \right\}$	$0.00\times10^{+00}$	1.44×10^{-34}	$0.00\times10^{+00}$	9.63×10^{-35}
$\left\{ \begin{array}{l} k = 2.0 \\ \beta = 1.0 \end{array} \right\}$	1.39×10^{-17}	3.17×10^{-16}	5.55×10^{-17}	4.98×10^{-28}
$\left\{ \begin{array}{l} k = 1.0 \\ \beta = 2.0 \end{array} \right\}$	$0.00\times10^{+00}$	2.55×10^{-16}	$0.00\times10^{+00}$	3.01×10^{-28}

sensitivity system is

$$\kappa_{\text{sens}} v = -\nabla\mu + \xi, \tag{4.2.10}$$

$$\nabla \cdot v = 0, \tag{4.2.11}$$

$$\mu|_{\Gamma_D} = 0, \tag{4.2.12}$$

$$v \cdot n|_{\Gamma_N} = 0. \tag{4.2.13}$$

As noted in Section 3.4.1, the resistance term is the same as in the FS system. Once the solution to those systems are found, the sensitivity to each parameter is computed using the integral

$$\{\partial_\pi H\} = -\int_\Omega \{\partial_\pi \kappa\}\, uv dx$$

Table 4.2.2: West to East Flow With Constant Parameters: Error in AS Flow

	Adjoint Sensitivity Error			
	$\{\partial_k H_1\}$	$\{\partial_k H_2\}$	$\{\partial_\beta H_1\}$	$\{\partial_\beta H_2\}$
$\left\{\begin{array}{l} k=1.0 \\ \beta=1.0 \end{array}\right\}$	$0.00\times10^{+00}$	3.41×10^{-16}	5.55×10^{-17}	1.80×10^{-16}
$\left\{\begin{array}{l} k=1.0 \\ \beta=0.0 \end{array}\right\}$	$0.00\times10^{+00}$	1.04×10^{-17}	$0.00\times10^{+00}$	1.73×10^{-17}
$\left\{\begin{array}{l} k=2.0 \\ \beta=1.0 \end{array}\right\}$	1.39×10^{-17}	1.38×10^{-16}	5.55×10^{-17}	1.53×10^{-16}
$\left\{\begin{array}{l} k=1.0 \\ \beta=2.0 \end{array}\right\}$	$0.00\times10^{+00}$	1.20×10^{-16}	$0.00\times10^{+00}$	3.12×10^{-17}

$$= \begin{cases} -\int_\Omega \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix} uv dx, & \pi = \beta, \\ \int_\Omega \begin{bmatrix} \frac{1}{k^2} & \\ & \frac{1}{k^2} \end{bmatrix} uv dx, & \pi = k. \end{cases}$$

It is important to note that only one adjoint system needs to be solved regardless of the number of parameters for which the sensitivity is desired. Each parameter, however, requires the computation of an integral.

Numerical results for the AS flow sensitivity are presented in Table 4.2.2. Recall that the error is the absolute value of the difference between the numerical approximation of the sensitivity and the analytic sensitivity computed above.

In all of the experiments, the error is smaller than the tolerance for the flow solver with zero error for several entries. This is thanks to the simplicity of the system and should not be expected for more complicated systems.

4.2.2 Sensitivity of Transport

As in the flow system, the analytic sensitivity is found and compared to the FS and AS transport solutions. The sensitivity of G to each parameter can be computed as

$$\{\partial_\pi G\} = \begin{cases} \frac{T}{2}\{\partial_\pi u_1\}, & 0 \le T \le \frac{1}{u_1}, \\ \frac{1}{2u_1^2 T}\{\partial_\pi u_1\}, & \frac{1}{u_1} < T. \end{cases} \tag{4.2.14}$$

The analytic sensitivity of c to the parameters is useful for verifying the accuracy of the FS transport solver:

$$\{\partial_\pi c\}(x,t) = \delta_{x_1 - u_1 t} t \{\partial_\pi u_1\}.$$

This sensitivity only exists in the sense of distributions. The finite volume transport solver approximates the average value of the quantity which involves an integral. This means that the numerical solution may be accurate since

$$\int_{V_{ij}} \{\partial_\pi c\}(x,t) = \begin{cases} 0, & \text{if } x_1 - u_1 t \notin V_{ij}, \\ t\{\partial_\pi u_1\}, & \text{if } x_1 - u_1 t \in V_{ij}. \end{cases}$$

The FS transport system satisfies (3.3.10)-(3.3.13). To solve this numerically, the source term $-\nabla \cdot (c\{\partial_\pi u\})$ must be computed for each parameter. In the numerical computations, the source terms are calculated using the numerical values of c and $\{\partial_\pi u\}$. In order to verify the numerical computations, it is useful to have

Table 4.2.3: West to East Flow With Constant Parameters: Error in FS Transport

	Forward Sensitivity Error	
	$\{\partial_k G\}$	$\{\partial_\beta G\}$
$\left\{\begin{array}{l} k=1.0 \\ \beta=1.0 \end{array}\right\}$	1.21×10^{-02}	7.45×10^{-03}
$\left\{\begin{array}{l} k=1.0 \\ \beta=0.0 \end{array}\right\}$	1.67×10^{-02}	1.67×10^{-02}
$\left\{\begin{array}{l} k=2.0 \\ \beta=1.0 \end{array}\right\}$	3.95×10^{-03}	1.23×10^{-02}
$\left\{\begin{array}{l} k=1.0 \\ \beta=2.0 \end{array}\right\}$	1.11×10^{-03}	5.56×10^{-04}

the analytic values:

$$-\nabla\cdot(c\{\partial_k u\}) = \begin{cases} \delta_{x_1-u_1t}, & \beta=0, \\ \frac{\sqrt{1+4\beta k^2}-1}{2\beta k^2\sqrt{1+4\beta k^2}}\delta_{x_1-u_1t}, & \beta\neq 0, \end{cases}$$

$$-\nabla\cdot(c\{\partial_\beta u\}) = \begin{cases} -k^3\delta_{x_1-u_1t}, & \beta=0, \\ \frac{\sqrt{1+4\beta k^2}-2\beta k^2-1}{2\beta^2 k\sqrt{1+4\beta k^2}}\delta_{x_1-u_1t}, & \beta\neq 0. \end{cases}$$

The error in the computed solution to the FS transport system is shown in Table 4.2.3. Recall that the error is the absolute value of the difference between the numerical approximation of the sensitivity and the analytic sensitivity computed above. The error in the approximation is of the order of the spatial grid.

The adjoint transport system (3.4.12)-(3.4.14) is solved backwards in time using the technique suggested in Remark 3.6. With the solution to the adjoint transport system, the adjoint flow problem with weight functions from (3.4.15) and (3.4.16)

Table 4.2.4: West to East Flow With Constant Parameters: Error in AS Transport

	Adjoint Sensitivity Error	
	$\{\partial_k G\}$	$\{\partial_\beta G\}$
$\left\{\begin{array}{l} k=1.0 \\ \beta=1.0 \end{array}\right\}$	6.66×10^{-16}	4.30×10^{-16}
$\left\{\begin{array}{l} k=1.0 \\ \beta=0.0 \end{array}\right\}$	4.72×10^{-16}	4.44×10^{-16}
$\left\{\begin{array}{l} k=2.0 \\ \beta=1.0 \end{array}\right\}$	6.47×10^{-05}	2.02×10^{-04}
$\left\{\begin{array}{l} k=1.0 \\ \beta=2.0 \end{array}\right\}$	2.14×10^{-15}	1.03×10^{-15}

is solved. Finally, (3.4.17) is used to compute the sensitivity of the quantity of interest. The numerical results for the transport sensitivity are presented in Table 4.2.4. In every experiment, the AS transport method outperforms the FS transport method.

4.3 West to East With Smooth Parameters

In this example, a pressure driven flow with smooth coefficients from the west side to the east side of $[0,1]\times[0,1]$ is considered. Three parameters are considered and the analytic sensitivity is shown for each. The analytic solution to the associated transport system is also shown and its sensitivity to the parameters in the flow problem is computed. The experiment is implemented in Listing 7 in the Appendix.

Let

$$\begin{aligned}
\kappa\left(k,\beta;|u|\right) &= \begin{bmatrix} \frac{1}{k}+\left(\beta+\gamma x_1\right)|u_1| & \\ & \frac{1}{k}+\left(\beta+\gamma x_1\right)|u_2| \end{bmatrix}, \\
f\left(x\right) &\equiv 0, \\
g\left(x\right) &\equiv 0, \\
p_D\left(x\right) &= x_1, \\
\Gamma_D &= \left\{x : x_1 = 0 \text{ or } x_1 = 1\right\}, \\
u_N\left(x\right) &= 0, \\
\Gamma_N &= \left\{x : x_2 = 0 \text{ or } x_2 = 1\right\},
\end{aligned}$$

for given parameters $k > 0$, $\beta \geq 0$, and $\gamma > -\beta$. In the same way as in Section 4.2 above, the system is equivalent to the following one dimensional system.

$$\begin{aligned}
\frac{u_1}{k}+\beta\left(x\right)|u_1|\,u_1 &= -\partial_{x_1}p, \\
\partial_{x_1}u_1 &= 0, \\
p\left(0\right)=1, \qquad & p\left(1\right)=0,
\end{aligned}$$

where $\beta\left(x\right) = \beta + \gamma x_1$ for given parameters $k > 0$, $\beta \geq 0$, and $\gamma > -\beta$. The system has solution

$$\begin{aligned}
u_1 &= \frac{-1+\sqrt{1+2k^2\left(2\beta+\gamma\right)}}{k\left(2\beta+\gamma\right)}, \\
u_2 &= 0,
\end{aligned}$$

Table 4.3.1: West to East Flow With Smooth Parameters: Flow Solver Error

		Error		
	N	p	u_1	u_2
$\left\{ \begin{array}{c} k=1.0 \\ \beta=2.0 \\ \gamma=-1.0 \end{array} \right\}$	8	5.88×10^{-04}	5.39×10^{-13}	$0.00\times10^{+00}$
	16	1.47×10^{-04}	5.83×10^{-13}	$0.00\times10^{+00}$
	32	3.67×10^{-05}	1.95×10^{-12}	$0.00\times10^{+00}$
	Order	2.00	-	-
$\left\{ \begin{array}{c} k=2.0 \\ \beta=2.0 \\ \gamma=-1.0 \end{array} \right\}$	8	8.68×10^{-04}	1.72×10^{-11}	$0.00\times10^{+00}$
	16	2.17×10^{-04}	1.81×10^{-11}	$0.00\times10^{+00}$
	32	5.43×10^{-05}	3.31×10^{-11}	$0.00\times10^{+00}$
	Order	2.00	-	-
$\left\{ \begin{array}{c} k=1.0 \\ \beta=1.0 \\ \gamma=1.0 \end{array} \right\}$	8	5.88×10^{-04}	5.17×10^{-13}	$0.00\times10^{+00}$
	16	1.47×10^{-04}	6.32×10^{-13}	$0.00\times10^{+00}$
	32	3.67×10^{-05}	1.44×10^{-12}	$0.00\times10^{+00}$
	Order	2.00	-	-
$\left\{ \begin{array}{c} k=1.0 \\ \beta=2.0 \\ \gamma=5.0 \end{array} \right\}$	8	1.36×10^{-03}	2.28×10^{-15}	$0.00\times10^{+00}$
	16	3.40×10^{-04}	1.55×10^{-15}	$0.00\times10^{+00}$
	32	8.50×10^{-05}	2.05×10^{-15}	$0.00\times10^{+00}$
	Order	2.00	-	-

$$p\left(x_1, x_2\right) \;=\; -\frac{\gamma}{2}u_1^2x_1^2 - \left(\beta u_1^2 + \frac{u_1}{k}\right)x_1 + 1,$$

This example also provides an opportunity to test the convergence of the flow solver to the analytic p. The order is not computed for u_1 or u_2 since the error is due to stopping a Newton iteration. The results are in Table 4.3.1. In this case, the error is defined to be the maximum pointwise error in p and u, respectively.

In the transport problem, let

$$c\left(x, 0\right) \;=\; 0,$$

$$\begin{aligned} c\left(0, x_2, t\right) &= 1, \\ D &\equiv 0. \end{aligned}$$

The solution satisfies equation (4.2.4) and the quantity of interest satisfies (4.2.5).

4.3.1 Flow Sensitivity

The analytic flow sensitivity, FS flow sensitivity, and AS flow sensitivity are computed. The analytic sensitivities for u are

$$\begin{aligned} \left\{\partial_k u_1\right\} &= \frac{\sqrt{4\beta k^2 + 2\gamma k^2 + 1} - 1}{k^2\left(2\beta + \gamma\right)\sqrt{4\beta k^2 + 2\gamma k^2 + 1}}, \\ \left\{\partial_k u_2\right\} &= 0, \\ \left\{\partial_\beta u_1\right\} &= \frac{2\sqrt{2k^2\left(2\beta + \gamma\right) + 1} - 2k^2\left(2\beta + \gamma\right) - 2}{k\left(2\beta + \gamma\right)^2\sqrt{2k^2\left(2\beta + \gamma\right) + 1}}, \\ \left\{\partial_\beta u_2\right\} &= 0, \\ \left\{\partial_\gamma u_1\right\} &= \frac{\sqrt{2k^2\left(2\beta + \gamma\right) + 1} - k^2\left(2\beta + \gamma\right) - 1}{k\left(2\beta + \gamma\right)^2\sqrt{2k^2\left(2\beta + \gamma\right) + 1}}, \\ \left\{\partial_\gamma u_2\right\} &= 0, \end{aligned}$$

and the analytic sensitivities for p are

$$\begin{aligned} \left\{\partial_k p\right\} &= -\gamma u_1\left\{\partial_k u_1\right\} x_1^2 - \left(2\beta u_1\left\{\partial_k u_1\right\} + \frac{k\left\{\partial_k u_1\right\} - u_1}{k^2}\right) x_1, \\ \left\{\partial_\beta p\right\} &= -\gamma u_1\left\{\partial_\beta u_1\right\} x_1^2 - \left(u_1^2 + 2\beta u_1\left\{\partial_\beta u_1\right\} + \frac{\left\{\partial_\beta u_1\right\}}{k}\right) x_1, \\ \left\{\partial_\gamma p\right\} &= -\left(\frac{u_1^2}{2} + \gamma u_1\left\{\partial_\gamma u_1\right\}\right) x_1^2 - \left(2\beta u_1 + \frac{1}{k}\right)\left\{\partial_\gamma u_1\right\} x_1. \end{aligned}$$

These formulas are useful in verifying the correctness of the algorithms.

The following are needed to compute the sensitivity

$$\begin{aligned}
\partial_{|u|}\kappa &= \begin{bmatrix} \beta+\gamma x_1 & \\ & \beta+\gamma x_1 \end{bmatrix}, \\
\{\partial_k \kappa\} &= \begin{bmatrix} -\frac{1}{k^2} & \\ & -\frac{1}{k^2} \end{bmatrix}, \\
\{\partial_\beta \kappa\} &= \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix}, \\
\{\partial_\gamma \kappa\} &= \begin{bmatrix} x_1 |u_1| & \\ & x_1 |u_2| \end{bmatrix}.
\end{aligned}$$

The resistance for each flow system is

$$\kappa_{\text{sens}} = \begin{bmatrix} \frac{1}{k} & \\ & \frac{1}{k} \end{bmatrix} + 2 \begin{bmatrix} \beta(x_1)|u_1| & \\ & \beta(x_1)|u_2| \end{bmatrix}.$$

The source term for the sensitivity to k is

$$g_k = -\{\partial_k \kappa\} u = \begin{bmatrix} \frac{u_1}{k^2} \\ \frac{u_2}{k^2} \end{bmatrix},$$

Table 4.3.2: West to East Flow With Smooth Parameters: Error in FS Flow

		Forward Sensitivity Error					
	N	$\{\partial_k H_1\}$	$\{\partial_k H_2\}$	$\{\partial_\beta H_1\}$	$\{\partial_\beta H_2\}$	$\{\partial_\gamma H_1\}$	$\{\partial_\gamma H_2\}$
$k = 1.0$ $\beta = 2.0$ $\gamma = -1.0$	8	2.66×10^{-15}	$0.00\times10^{+00}$	2.53×10^{-15}	$0.00\times10^{+00}$	6.38×10^{-15}	$0.00\times10^{+00}$
	16	6.94×10^{-16}	$0.00\times10^{+00}$	6.80×10^{-16}	$0.00\times10^{+00}$	1.69×10^{-15}	$0.00\times10^{+00}$
	32	2.58×10^{-15}	$0.00\times10^{+00}$	6.79×10^{-13}	$0.00\times10^{+00}$	6.03×10^{-15}	$0.00\times10^{+00}$
$k = 2.0$ $\beta = 2.0$ $\gamma = -1.0$	8	3.47×10^{-14}	$0.00\times10^{+00}$	1.70×10^{-13}	$0.00\times10^{+00}$	3.75×10^{-13}	$0.00\times10^{+00}$
	16	2.22×10^{-14}	$0.00\times10^{+00}$	1.09×10^{-13}	$0.00\times10^{+00}$	2.40×10^{-13}	$0.00\times10^{+00}$
	32	3.12×10^{-14}	$0.00\times10^{+00}$	1.52×10^{-13}	$0.00\times10^{+00}$	3.36×10^{-13}	$0.00\times10^{+00}$
$k = 1.0$ $\beta = 1.0$ $\gamma = 1.0$	8	3.05×10^{-15}	$0.00\times10^{+00}$	2.86×10^{-15}	$0.00\times10^{+00}$	4.38×10^{-15}	$0.00\times10^{+00}$
	16	1.19×10^{-15}	$0.00\times10^{+00}$	1.10×10^{-15}	$0.00\times10^{+00}$	1.75×10^{-15}	$0.00\times10^{+00}$
	32	6.27×10^{-15}	$0.00\times10^{+00}$	6.87×10^{-13}	$0.00\times10^{+00}$	9.18×10^{-15}	$0.00\times10^{+00}$
$k = 1.0$ $\beta = 2.0$ $\gamma = 5.0$	8	1.39×10^{-17}	$0.00\times10^{+00}$	1.39×10^{-17}	$0.00\times10^{+00}$	$0.00\times10^{+00}$	$0.00\times10^{+00}$
	16	6.94×10^{-17}	$0.00\times10^{+00}$	1.73×10^{-16}	$0.00\times10^{+00}$	1.39×10^{-17}	$0.00\times10^{+00}$
	32	1.39×10^{-17}	$0.00\times10^{+00}$	1.39×10^{-17}	$0.00\times10^{+00}$	6.94×10^{-18}	$0.00\times10^{+00}$

for the sensitivity to β is

$$g_\beta = -\{\partial_\beta \kappa\}\, u = \begin{bmatrix} |u_1|\, u_1 \\ |u_2|\, u_2 \end{bmatrix},$$

and for the sensitivity to γ is

$$g_\gamma = -\{\partial_\gamma \kappa\}\, u = \begin{bmatrix} x\,|u_1|\, u_1 \\ x\,|u_2|\, u_2 \end{bmatrix}.$$

The FS sensitivity is the solution to (4.2.6)-(4.2.9). Numerical results for the FS flow are in Table 4.3.2. Recall that the error is the absolute value of the difference between the numerical approximation of the sensitivity and the analytic sensitivity computed above.

For each quantity of interest with its corresponding weight function ξ, the AS

sensitivity system is (4.2.10)-(4.2.13). Once the solution to the adjoint sensitivity systems are found, the sensitivity to each parameter is computed using the integral

$$\begin{aligned}\{\partial_\pi H\} &= -\int_\Omega \{\partial_\pi \kappa\}\, uv dx \\ &= \begin{cases} -\int_\Omega \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix} uv dx, & \pi = \beta, \\ -\int_\Omega \begin{bmatrix} x_1 |u_1| & \\ & x_1 |u_2| \end{bmatrix} uv dx, & \pi = \gamma, \\ \int_\Omega \begin{bmatrix} \frac{1}{k^2} & \\ & \frac{1}{k^2} \end{bmatrix} uv dx, & \pi = k. \end{cases}\end{aligned}$$

Numerical results for the AS flow are presented in Table 4.3.2. Recall that the error is the absolute value of the difference between the numerical approximation of the sensitivity and the analytic sensitivity computed above.

Both methods perform well on the flow problem. In each case the error is less than the tolerance used to stop the computation of the solution.

4.3.2 Transport Sensitivity

The sensitivity for the quantity of interest in the transport system satisfies (4.2.14). The FS transport system satisfies (3.3.10)-(3.3.13). Numerical results for the FS transport are in Table 4.3.4.

Table 4.3.3: West to East Flow With Smooth Parameters: Error in AS Flow

		Adjoint Sensitivity Error					
	N	$\{\partial_k H_1\}$	$\{\partial_k H_2\}$	$\{\partial_\beta H_1\}$	$\{\partial_\beta H_2\}$	$\{\partial_\gamma H_1\}$	$\{\partial_\gamma H_2\}$
$k = 1.0$ $\beta = 2.0$ $\gamma = -1.0$	8	2.64×10^{-15}	$0.00\times10^{+00}$	2.53×10^{-15}	$0.00\times10^{+00}$	6.37×10^{-15}	$0.00\times10^{+00}$
	16	6.94×10^{-16}	$0.00\times10^{+00}$	6.66×10^{-16}	8.67×10^{-19}	1.76×10^{-15}	8.67×10^{-19}
	32	2.53×10^{-15}	3.93×10^{-19}	2.40×10^{-15}	7.05×10^{-19}	6.04×10^{-15}	4.13×10^{-19}
$k = 2.0$ $\beta = 2.0$ $\gamma = -1.0$	8	3.47×10^{-14}	$0.00\times10^{+00}$	1.70×10^{-13}	$0.00\times10^{+00}$	3.74×10^{-13}	$0.00\times10^{+00}$
	16	2.22×10^{-14}	$0.00\times10^{+00}$	1.09×10^{-13}	$0.00\times10^{+00}$	2.40×10^{-13}	$0.00\times10^{+00}$
	32	3.12×10^{-14}	8.98×10^{-20}	1.52×10^{-13}	3.52×10^{-19}	3.36×10^{-13}	2.54×10^{-20}
$k = 1.0$ $\beta = 1.0$ $\gamma = 1.0$	8	3.00×10^{-15}	$0.00\times10^{+00}$	2.84×10^{-15}	$0.00\times10^{+00}$	4.39×10^{-15}	$0.00\times10^{+00}$
	16	1.19×10^{-15}	1.73×10^{-18}	1.12×10^{-15}	$0.00\times10^{+00}$	1.69×10^{-15}	1.73×10^{-18}
	32	6.30×10^{-15}	4.88×10^{-19}	5.98×10^{-15}	6.51×10^{-19}	9.17×10^{-15}	5.42×10^{-19}
$k = 1.0$ $\beta = 2.0$ $\gamma = 5.0$	8	1.39×10^{-17}	$0.00\times10^{+00}$	6.94×10^{-18}	$0.00\times10^{+00}$	3.47×10^{-18}	$0.00\times10^{+00}$
	16	5.55×10^{-17}	8.67×10^{-19}	1.39×10^{-17}	$0.00\times10^{+00}$	1.04×10^{-17}	$0.00\times10^{+00}$
	32	1.39×10^{-17}	4.04×10^{-19}	6.94×10^{-18}	9.04×10^{-20}	$0.00\times10^{+00}$	5.07×10^{-20}

The adjoint transport system (3.4.12)-(3.4.14) is solved backwards in time using the technique suggested in Remark 3.6. With the solution to the adjoint transport system, the adjoint flow problem with weight functions from (3.4.15) and (3.4.16) is solved. Finally, (3.4.17) is used to compute the sensitivity of the quantity of interest. Numerical results for the AS transport are in Table 4.3.5. Recall that the error is the absolute value of the difference between the numerical approximation of the sensitivity and the analytic sensitivity computed above. The FS transport sensitivities perform worse than the AS transport sensitivities. The FS transport sensitivities converge to the true solution with order approximately 1 while the AS transport converge with order approximately 2.

Table 4.3.4: West to East Flow With Smooth Parameters: Error in FS Transport

		Forward Sensitivity Error		
	N	$\{\partial_k G\}$	$\{\partial_\beta G\}$	$\{\partial_\gamma G\}$
$\left\{ \begin{array}{c} k = 1.0 \\ \beta = 2.0 \\ \gamma = -1.0 \end{array} \right\}$	8	4.40×10^{-03}	2.41×10^{-03}	1.21×10^{-03}
	16	2.18×10^{-03}	1.19×10^{-03}	5.97×10^{-04}
	32	1.08×10^{-03}	5.94×10^{-04}	2.97×10^{-04}
	Order	1.01	1.01	1.01
$\left\{ \begin{array}{c} k = 2.0 \\ \beta = 2.0 \\ \gamma = -1.0 \end{array} \right\}$	8	9.57×10^{-04}	2.55×10^{-03}	1.28×10^{-03}
	16	3.13×10^{-04}	8.33×10^{-04}	4.17×10^{-04}
	32	2.36×10^{-04}	6.28×10^{-04}	3.14×10^{-04}
	Order	1.01	1.01	1.01
$\left\{ \begin{array}{c} k = 1.0 \\ \beta = 1.0 \\ \gamma = 1.0 \end{array} \right\}$	8	4.40×10^{-03}	2.41×10^{-03}	1.21×10^{-03}
	16	2.18×10^{-03}	1.19×10^{-03}	5.97×10^{-04}
	32	1.08×10^{-03}	5.94×10^{-04}	2.97×10^{-04}
	Order	1.01	1.01	1.01
$\left\{ \begin{array}{c} k = 1.0 \\ \beta = 2.0 \\ \gamma = 5.0 \end{array} \right\}$	8	3.92×10^{-03}	1.46×10^{-03}	7.32×10^{-04}
	16	1.94×10^{-03}	7.24×10^{-04}	3.62×10^{-04}
	32	9.65×10^{-04}	3.60×10^{-04}	1.80×10^{-04}
	Order	1.01	1.01	1.01

Table 4.3.5: West to East Flow With Smooth Parameters: Error in AS Transport

		Adjoint Sensitivity Error		
	N	$\{\partial_k G\}$	$\{\partial_\beta G\}$	$\{\partial_\gamma G\}$
$\left\{ \begin{array}{c} k = 1.0 \\ \beta = 2.0 \\ \gamma = -1.0 \end{array} \right\}$	8	8.97×10^{-05}	4.92×10^{-05}	2.46×10^{-05}
	16	2.24×10^{-05}	1.23×10^{-05}	6.15×10^{-06}
	32	5.61×10^{-06}	3.08×10^{-06}	1.54×10^{-06}
	Order	2.00	2.00	2.00
$\left\{ \begin{array}{c} k = 2.0 \\ \beta = 2.0 \\ \gamma = -1.0 \end{array} \right\}$	8	1.95×10^{-05}	5.21×10^{-05}	2.60×10^{-05}
	16	1.65×10^{-13}	4.46×10^{-13}	2.44×10^{-13}
	32	1.22×10^{-06}	3.26×10^{-06}	1.63×10^{-06}
	Order	2.00	2.00	2.00
$\left\{ \begin{array}{c} k = 1.0 \\ \beta = 1.0 \\ \gamma = 1.0 \end{array} \right\}$	8	8.97×10^{-05}	4.92×10^{-05}	2.46×10^{-05}
	16	2.24×10^{-05}	1.23×10^{-05}	6.15×10^{-06}
	32	5.61×10^{-06}	3.08×10^{-06}	1.54×10^{-06}
	Order	2.00	2.00	2.00
$\left\{ \begin{array}{c} k = 1.0 \\ \beta = 2.0 \\ \gamma = 5.0 \end{array} \right\}$	8	8.00×10^{-05}	2.99×10^{-05}	1.49×10^{-05}
	16	2.00×10^{-05}	7.47×10^{-06}	3.73×10^{-06}
	32	5.00×10^{-06}	1.87×10^{-06}	9.33×10^{-07}
	Order	2.00	2.00	2.00

4.4 West to East With Discontinuous Parameters

In this example, a pressure driven flow with discontinuous coefficients from the west side to the east side of $[0,1] \times [0,1]$ is considered. The flow is simple enough that an analytic solution is provided. Five parameters are considered and the analytic sensitivity is shown for each. The analytic solution to the associated transport system is also shown and its sensitivity to the parameters in the flow problem is computed. The experiment is implemented in Listing 8 in the Appendix.

Let

$$\begin{aligned}
\kappa\left(k,\beta;|u|\right) &= \begin{bmatrix} \frac{1}{k(x)}+\beta\left(x\right)|u_1| & \\ & \frac{1}{k(x)}+\beta\left(x\right)|u_2| \end{bmatrix}, \\
f\left(x\right) &\equiv 0, \\
g\left(x\right) &\equiv 0, \\
p_D\left(x\right) &= x_1, \\
\Gamma_D &= \left\{x : x_1 = 0 \text{ or } x_1 = 1\right\}, \\
u_N\left(x\right) &= 0, \\
\Gamma_N &= \left\{x : x_2 = 0 \text{ or } x_2 = 1\right\},
\end{aligned}$$

where

$$k\left(x\right) = \begin{cases} k_1, & x_1 < x_0, \\ k_2, & x_1 > x_0, \end{cases}$$

$$\beta(x) = \begin{cases} \beta_1, & x_1 < x_0, \\ \beta_2, & x_1 > x_0, \end{cases}$$

for some $x_0 \in (0,1)$. To the west of x_0 the resistance tensor $\kappa(k, \beta; |u|)$ has entries dependent on k_1 and β_1 while to the east of x_0 the entries depend on k_2 and β_2. The discontinuity in κ is of particular interest. The sensitivity to x_0 represents the sensitivity to the location of the discontinuity. In the same way as in Section 4.2 above, the system is equivalent to the following one dimensional system:

$$\begin{aligned} \frac{u_1}{k(x)} + \beta(x) |u_1| u_1 &= -\partial_{x_1} p, \\ \partial_{x_1} u_1 &= 0, \\ p(0) = 1, \qquad & p(1) = 0. \end{aligned}$$

The solution is

$$\begin{aligned} p(x) &= \begin{cases} 1 - \left(\frac{u_1}{k_1} + \beta_1 |u_1| u_1\right) x_1, & x_1 < x_0, \\ \left(\frac{u_1}{k_2} + \beta_2 |u_1| u_1\right) - \left(\frac{u_1}{k_2} + \beta_2 |u_1| u_1\right) x_1, & x_1 > x_0, \end{cases} \\ u_1 &= \frac{\sqrt{\left(k_1(1-x_0) + k_2 x_0\right)^2 + 4k_1^2 k_2^2 \left(\beta_2(1-x_0) + \beta_1 x_0\right)}}{2k_1 k_2 \left(\beta_2(1-x_0) + \beta_1 x_0\right)} \\ &\quad - \frac{k_1(1-x_0) + k_2 x_0}{2k_1 k_2 \left(\beta_2(1-x_0) + \beta_1 x_0\right)}, \\ u_2 &= 0. \end{aligned}$$

The flow solver should be able to compute the solution exactly, but there is one

Table 4.4.1: West to East Flow With Discontinuous Parameters: Flow Solver Error

	Error		
	p	u_1	u_2
N=10	6.38×10^{-04}	3.54×10^{-04}	9.73×10^{-12}
N=11	1.05×10^{-10}	1.18×10^{-10}	7.66×10^{-11}

possible problem. The grid should be aligned to the discontinuity. In Table 4.4.1, the error is compared with $N = 10$ and $N = 11$ grid points for the experiment with $k_1 = 1$, $k_2 = 2$, $\beta_1 = 3$, $\beta_2 = 4$, and $x_0 = 0.5$. With $N = 10$ grid points the grid is not aligned to the discontinuity while with $N = 11$ it is. The error in this case is the maximum pointwise difference between the analytic solution and the numerical solution. The true solution is

$$\begin{aligned} p(x) &= \begin{cases} 1-(u_1+3|u_1|u_1)x_1, & x_1 < \frac{1}{2}, \\ \left(\frac{u_1}{2}+4|u_1|u_1\right)(1-x_1), & x_1 > \frac{1}{2}, \end{cases} \\ u_1 &= \frac{\sqrt{233}-3}{28}, \\ u_2 &= 0. \end{aligned}$$

This example shows that an odd number of grid points are needed to accurately solve the system.

In the transport problem, let

$$\begin{aligned} c(x,0) &= 0, \\ c(0,x_2,t) &= 1, \end{aligned}$$

$$D \equiv 0.$$

The solution satisfies equation (4.2.4) and the quantity of interest satisfies (4.2.5).

4.4.1 Flow Sensitivity

Let

$$s = \sqrt{\left(k_1\left(1-x_0\right)+k_2x_0\right)^2 + 4k_1^2k_2^2\left(\beta_2\left(1-x_0\right)+\beta_1x_0\right)},$$

in order to simplify the expressions that follow. The analytic sensitivity of u_2 to any parameter is zero, the analytic sensitivity of u_1 is

$$\begin{aligned}
\{\partial_{k_1}u_1\} &= \frac{x_0\left(k_1\left(x_0-1\right)-k_2x_0+s\right)}{2k_1^2\left(\beta_1x_0+\beta_2\left(1-x_0\right)\right)s},\\
\{\partial_{k_2}u_1\} &= \frac{4k_1^2k_2\left(\beta_1x_0+\beta_2\left(1-x_0\right)\right)}{2k_1k_2\left(\beta_1x_0+\beta_2\left(1-x_0\right)\right)s}\\
&\quad+\frac{x_0\left(k_1\left(1-x_0\right)+k_2x_0\right)-x_0s}{2k_1k_2\left(\beta_1x_0+\beta_2\left(1-x_0\right)\right)s}\\
&\quad+\frac{k_1\left(1-x_0\right)+k_2x_0-s}{2k_1k_2^2\left(\beta_1x_0+\beta_2\left(-x_0+1\right)\right)},\\
\{\partial_{\beta_1}u_1\} &= \frac{k_1k_2x_0}{\left(\beta_1x_0-\beta_2\left(x_0-1\right)\right)s}\\
&\quad-\frac{x_0\left(k_1\left(x_0-1\right)-k_2x_0+s\right)}{2k_1k_2\left(\beta_1x_0-\beta_2\left(x_0-1\right)\right)^2},\\
\{\partial_{\beta_2}u_1\} &= \frac{k_1k_2\left(-x_0+1\right)}{\left(\beta_1x_0+\beta_2\left(-x_0+1\right)\right)s}\\
&\quad+\frac{\left(x_0-1\right)\left(k_1\left(x_0-1\right)-k_2x_0+s\right)}{2k_1k_2\left(\beta_1x_0+\beta_2\left(-x_0+1\right)\right)^2},
\end{aligned}$$

$$\begin{aligned}\{\partial_{x_0} u_1\} &= \frac{(\beta_2-\beta_1)\left(k_1\left(x_0-1\right)-k_2 x_0\right)+s\left(\beta_2-\beta_1\right)}{2k_1k_2\left(\beta_1 x_0+\beta_2\left(-x_0+1\right)\right)^2} \\ &+\frac{s\left(k_1-k_2\right)+2k_1^2k_2^2\left(\beta_1-\beta_2\right)}{2sk_1k_2\left(\beta_1 x_0+\beta_2\left(-x_0+1\right)\right)} \\ &+\frac{\left(k_2-k_1\right)\left(k_1\left(1-x_0\right)+k_2 x_0\right)}{2sk_1k_2\left(\beta_1 x_0+\beta_2\left(-x_0+1\right)\right)}\end{aligned}$$

and the sensitivity of p is

$$\{\partial_{k_1} p\} = \begin{cases} -x_1\left(\left(2\beta_1 u_1+\frac{1}{k_1}\right)\{\partial_{k_1} u_1\}-\frac{u_1}{k_1^2}\right), & x<x_0, \\ (1-x_1)\left(2\beta_2 u_1+\frac{1}{k_2}\right)\{\partial_{k_1} u_1\}, & x>x_0 \end{cases}$$

$$\{\partial_{k_2} p\} = \begin{cases} -x_1\left(2\beta_1 u_1+\frac{1}{k_1}\right)\{\partial_{k_2} u_1\}, & x<x_0, \\ (1-x_1)\left(\left(2\beta_2 u_1+\frac{1}{k_2}\right)\{\partial_{k_2} u_1\}-\frac{1}{k_2^2}u_1\right), & x>x_0, \end{cases}$$

$$\{\partial_{\beta_1} p\} = \begin{cases} -x_1\left(2\beta_1 u_1+u_1^2+\frac{1}{k_1}\right)\{\partial_{\beta_1} u_1\}, & x<x_0, \\ (1-x_1)\left(2\beta_2 u_1+\frac{1}{k_2}\right)\{\partial_{\beta_1} u_1\}, & x>x_0, \end{cases}$$

$$\{\partial_{\beta_2} p\} = \begin{cases} -x_1\left(2\beta_1 u_1+\frac{1}{k_1}\right)\{\partial_{\beta_2} u_1\}, & x<x_0, \\ (1-x_1)\left(\left(2\beta_2 u_1+\frac{1}{k_2}\right)\{\partial_{\beta_2} u_1\}+u_1^2\right), & x>x_0, \end{cases}$$

$$\{\partial_{x_0} p\} = \begin{cases} -x_1\left(2\beta_1 u_1+\frac{1}{k_1}\right)\{\partial_{x_0} u_1\}, & x<x_0, \\ (1-x_1)\left(2\beta_2 u_1+\frac{1}{k_2}\right)\{\partial_{x_0} u_1\}, & x>x_0. \end{cases}$$

The analytic sensitivities are useful for verifying the numerical results.

For the FS sensitivity, it is necessary to compute

$$\partial_{|u|}\kappa = \begin{cases} \begin{bmatrix} \beta_1 & \\ & \beta_1 \end{bmatrix}, & x < x_0, \\ \begin{bmatrix} \beta_2 & \\ & \beta_2 \end{bmatrix}, & x > x_0, \end{cases}$$

$$\{\partial_{k_1}\kappa\} = \begin{cases} \begin{bmatrix} -\frac{1}{k_1^2} & \\ & -\frac{1}{k_1^2} \end{bmatrix}, & x_1 < x_0, \\ \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, & x > x_0, \end{cases}$$

$$\{\partial_{k_2}\kappa\} = \begin{cases} \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, & x < x_0, \\ \begin{bmatrix} -\frac{1}{k_2^2} & \\ & -\frac{1}{k_2^2} \end{bmatrix}, & x > x_0, \end{cases}$$

$$\{\partial_{\beta_1}\kappa\} = \begin{cases} \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix}, & x_1 < x_0, \\ \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, & x_1 > x_0, \end{cases}$$

$$\{\partial_{\beta_2}\kappa\} = \begin{cases} \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, & x_1 < x_0, \\ \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix}, & x_1 > x_0, \end{cases}$$

$$\{\partial_{x_0}\kappa\} = \delta_{x_0} \begin{bmatrix} \frac{1}{k_2} - \frac{1}{k_1} + (\beta_2 - \beta_1)|u_1| & \\ & \frac{1}{k_2} - \frac{1}{k_1} + (\beta_2 - \beta_1)|u_2| \end{bmatrix}.$$

The resistance for the sensitivity equation is

$$\kappa_{\text{sens}} = \begin{bmatrix} \frac{1}{k(x)} + 2\beta(x)|u_1| & \\ & \frac{1}{k(x)} + 2\beta(x)|u_2| \end{bmatrix}.$$

The source term is computed as $\{\partial_\pi \kappa\} u$ for each parameter

$$\pi \in \{k_1, k_2, \beta_1, \beta_2, x_0\}.$$

The source terms all depend on space, but that presents no difficulties for the numerical implementation. The FS sensitivity is the solution to (4.2.6)-(4.2.9).

The most interesting sensitivity in this example is the sensitivity to x_0. Numerically, this is a difficult sensitivity to find since implementing the source term requires the use of a numerical δ function. An implementation based on [71] may be found in Listing 11 in the Appendix.

Numerical results for the FS flow are in Tables 4.4.2 and 4.4.3. Recall that the

Table 4.4.2: West to East Flow With Discontinuous Parameters: Error in FS Flow

		Forward Sensitivity Error			
	N	$\{\partial_{k_1} H_1\}$	$\{\partial_{k_1} H_2\}$	$\{\partial_{k_2} H_1\}$	$\{\partial_{k_2} H_2\}$
$\left\{ \begin{array}{l} k_1 = 2.0, k_2 = 1.0 \\ \beta_1 = 0.5, \beta_2 = 1.0 \end{array} \right\}$	5	3.47×10^{-17}	3.92×10^{-17}	2.50×10^{-16}	1.94×10^{-16}
	15	3.75×10^{-16}	7.84×10^{-16}	1.33×10^{-15}	1.19×10^{-15}
	45	1.21×10^{-11}	9.32×10^{-16}	2.32×10^{-11}	2.14×10^{-16}
$\left\{ \begin{array}{l} k_1 = 1.0, k_2 = 2.0 \\ \beta_1 = 1.0, \beta_2 = 0.5 \end{array} \right\}$	5	1.94×10^{-16}	7.08×10^{-17}	2.08×10^{-17}	2.86×10^{-16}
	15	5.55×10^{-17}	1.62×10^{-15}	5.55×10^{-16}	3.87×10^{-15}
	45	3.68×10^{-11}	2.63×10^{-17}	2.15×10^{-11}	7.13×10^{-16}

Table 4.4.3: West to East Flow With Discontinuous Parameters: Error in FS Flow

		Forward Sensitivity Error					
	N	$\{\partial_{\beta_1} H_1\}$	$\{\partial_{\beta_1} H_2\}$	$\{\partial_{\beta_2} H_1\}$	$\{\partial_{\beta_2} H_2\}$	$\{\partial_{x_0} H_1\}$	$\{\partial_{x_0} H_2\}$
$\left\{ \begin{array}{l} k_1 = 2.0, k_2 = 1.0 \\ \beta_1 = 0.5, \beta_2 = 1.0 \end{array} \right\}$	5	5.55×10^{-17}	1.22×10^{-16}	$0.00\times10^{+00}$	2.78×10^{-17}	3.33×10^{-16}	1.03×10^{-16}
	15	1.14×10^{-15}	2.73×10^{-16}	5.27×10^{-16}	3.03×10^{-16}	5.55×10^{-16}	4.99×10^{-14}
	45	7.60×10^{-11}	3.60×10^{-15}	2.71×10^{-11}	7.27×10^{-16}	8.91×10^{-10}	1.10×10^{-15}
$\left\{ \begin{array}{l} k_1 = 1.0, k_2 = 2.0 \\ \beta_1 = 1.0, \beta_2 = 0.5 \end{array} \right\}$	5	5.55×10^{-17}	2.78×10^{-18}	5.55×10^{-17}	2.71×10^{-16}	1.67×10^{-16}	2.93×10^{-16}
	15	9.44×10^{-16}	1.20×10^{-15}	5.55×10^{-16}	5.42×10^{-16}	9.99×10^{-16}	2.11×10^{-14}
	45	5.58×10^{-11}	3.39×10^{-18}	1.11×10^{-10}	7.31×10^{-18}	1.22×10^{-11}	3.36×10^{-17}

error is the absolute value of the difference between the numerical approximation of the sensitivity and the analytic sensitivity computed above.

For each quantity of interest with its corresponding weight function ξ, the AS sensitivity system is (4.2.10)-(4.2.13). Once the solution to the adjoint sensitivity system are found, the sensitivity to each parameter is computed using the integral

$$\{\partial_\pi H\} \quad = \quad -\int_\Omega \{\partial_\pi \kappa\}\, uv dx.$$

Since $\{\partial_{x_0} u\}$ is not computed, the singular source term is not needed and so the implementation is easier than the forward sensitivity flow problem. When the

Table 4.4.4: West to East Flow With Discontinuous Parameters: Error in AS Flow

		Adjoint Sensitivity Error			
	N	$\{\partial_{k_1} H_1\}$	$\{\partial_{k_1} H_2\}$	$\{\partial_{k_2} H_1\}$	$\{\partial_{k_2} H_2\}$
$\left\{ \begin{array}{l} k_1 = 2.0, k_2 = 1.0 \\ \beta_1 = 0.5, \beta_2 = 1.0 \end{array} \right\}$	5	4.16×10^{-17}	1.73×10^{-18}	2.22×10^{-16}	1.38×10^{-18}
	15	5.34×10^{-16}	2.85×10^{-18}	1.22×10^{-15}	9.61×10^{-18}
	45	1.11×10^{-11}	8.34×10^{-16}	2.23×10^{-11}	2.68×10^{-16}
$\left\{ \begin{array}{l} k_1 = 1.0, k_2 = 2.0 \\ \beta_1 = 1.0, \beta_2 = 0.5 \end{array} \right\}$	5	1.39×10^{-16}	6.36×10^{-18}	$0.00\times10^{+00}$	2.96×10^{-18}
	15	1.67×10^{-16}	6.85×10^{-17}	6.94×10^{-18}	3.10×10^{-17}
	45	3.61×10^{-11}	2.18×10^{-17}	1.81×10^{-11}	2.92×10^{-18}

Table 4.4.5: West to East Flow With Discontinuous Parameters: Error in AS Flow

		Adjoint Sensitivity Error					
	N	$\{\partial_{\beta_1} H_1\}$	$\{\partial_{\beta_1} H_2\}$	$\{\partial_{\beta_2} H_1\}$	$\{\partial_{\beta_2} H_2\}$	$\{\partial_{x_0} H_1\}$	$\{\partial_{x_0} H_2\}$
$\left\{ \begin{array}{l} k_1 = 2.0, k_2 = 1.0 \\ \beta_1 = 0.5, \beta_2 = 1.0 \end{array} \right\}$	5	2.78×10^{-17}	8.67×10^{-18}	8.33×10^{-17}	3.47×10^{-18}	2.78×10^{-16}	5.77×10^{-17}
	15	1.61×10^{-15}	2.40×10^{-17}	8.88×10^{-16}	4.17×10^{-18}	4.44×10^{-16}	1.38×10^{-17}
	45	7.20×10^{-11}	3.45×10^{-15}	3.60×10^{-11}	2.08×10^{-16}	9.03×10^{-10}	5.99×10^{-16}
$\left\{ \begin{array}{l} k_1 = 1.0, k_2 = 2.0 \\ \beta_1 = 1.0, \beta_2 = 0.5 \end{array} \right\}$	5	2.78×10^{-17}	6.94×10^{-18}	8.33×10^{-17}	6.94×10^{-18}	1.11×10^{-16}	1.91×10^{-17}
	15	1.11×10^{-16}	3.71×10^{-17}	2.78×10^{-17}	5.64×10^{-17}	3.33×10^{-16}	1.65×10^{-16}
	45	5.54×10^{-11}	2.16×10^{-17}	1.11×10^{-10}	1.61×10^{-17}	1.13×10^{-11}	1.60×10^{-18}

sensitivity of the quantity of interest to x_0 is computed, the term with δ_{x_0} simplifies in the integral so that a function evaluation is used. Numerical results for the AS flow are presented in Tables 4.4.4 and 4.4.5. Recall that the error is the absolute value of the difference between the numerical approximation of the sensitivity and the analytic sensitivity computed above.

The FS flow and AS flow sensitivity calculations perform very well and have error smaller than the tolerance used for the solver.

Table 4.4.6: West to East Transport With Discontinuous Parameters: Error in FS Transport

		Forward Sensitivity Error				
	N	$\{\partial_{k_1} G\}$	$\{\partial_{k_2} G\}$	$\{\partial_{\beta_1} G\}$	$\{\partial_{\beta_2} G\}$	$\{\partial_{x_0} G\}$
$\left\{ \begin{array}{l} k_1 = 2.0, k_2 = 1.0 \\ \beta_1 = 0.5, \beta_2 = 1.0 \end{array} \right\}$	5	9.02×10^{-04}	3.61×10^{-03}	2.74×10^{-03}	2.74×10^{-03}	6.35×10^{-03}
	15	2.94×10^{-04}	1.18×10^{-03}	8.93×10^{-04}	8.93×10^{-04}	2.07×10^{-03}
	45	9.74×10^{-05}	3.90×10^{-04}	2.95×10^{-04}	2.95×10^{-04}	6.85×10^{-04}
	Order	1.01	1.01	1.01	1.01	1.01
$\left\{ \begin{array}{l} k_1 = 1.0, k_2 = 2.0 \\ \beta_1 = 1.0, \beta_2 = 0.5 \end{array} \right\}$	5	3.61×10^{-03}	9.02×10^{-04}	2.74×10^{-03}	2.74×10^{-03}	6.35×10^{-03}
	15	1.18×10^{-03}	2.94×10^{-04}	8.93×10^{-04}	8.93×10^{-04}	2.07×10^{-03}
	45	3.90×10^{-04}	9.74×10^{-05}	2.95×10^{-04}	2.95×10^{-04}	6.85×10^{-04}
	Order	1.01	1.01	1.01	1.01	1.01

4.4.2 Transport Sensitivity

The sensitivity for the quantity of interest in the transport system satisfies (4.2.14). The FS transport system satisfies (3.3.10)-(3.3.13). Numerical results for the FS transport are in Table 4.4.6.

The adjoint transport system (3.4.12)-(3.4.14) is solved backwards in time using the technique suggested in Remark 3.6. With the solution to the adjoint transport system, the adjoint flow problem with weight functions from (3.4.15) and (3.4.16) is solved. Finally, (3.4.17) is used to compute the sensitivity of the quantity of interest. Numerical results for the AS transport are in Table 4.4.7. Recall that the error is the absolute value of the difference between the numerical approximation of the sensitivity and the analytic sensitivity computed above. The FS transport sensitivities perform worse than the AS transport sensitivities. The FS transport sensitivities converge to the true solution with order approximately 1 while the AS transport converge with order approximately 2.

Table 4.4.7: West to East Transport With Discontinuous Parameters: Error in AS Transport

		Adjoint Sensitivity Error				
	N	$\{\partial_{k_1} G\}$	$\{\partial_{k_2} G\}$	$\{\partial_{\beta_1} G\}$	$\{\partial_{\beta_2} G\}$	$\{\partial_{x_0} G\}$
$\left\{ \begin{array}{c} k_1 = 2.0, k_2 = 1.0 \\ \beta_1 = 0.5, \beta_2 = 1.0 \end{array} \right\}$	5	2.91×10^{-05}	1.16×10^{-04}	8.83×10^{-05}	8.83×10^{-05}	2.05×10^{-04}
	15	3.23×10^{-06}	1.29×10^{-05}	9.81×10^{-06}	9.81×10^{-06}	2.27×10^{-05}
	45	3.59×10^{-07}	1.44×10^{-06}	1.09×10^{-06}	1.09×10^{-06}	2.53×10^{-06}
	Order	2.00	2.00	2.00	2.00	2.00
$\left\{ \begin{array}{c} k_1 = 1.0, k_2 = 2.0 \\ \beta_1 = 1.0, \beta_2 = 0.5 \end{array} \right\}$	5	1.16×10^{-04}	2.91×10^{-05}	8.83×10^{-05}	8.83×10^{-05}	2.05×10^{-04}
	15	1.29×10^{-05}	3.23×10^{-06}	9.81×10^{-06}	9.81×10^{-06}	2.27×10^{-05}
	45	1.44×10^{-06}	3.59×10^{-07}	1.09×10^{-06}	1.09×10^{-06}	2.53×10^{-06}
	Order	2.00	2.00	2.00	2.00	2.00

Chapter 5 Homotopy and Continuation

In this chapter homotopy and numerical continuation will be used to solve the nonlinear flow model. Two functions are said to be *homotopic* if one can be continuously deformed into the other [50]. If two functions are homotopic, then the continuous deformation of one into the other is called a *homotopy* [50]. *Numerical continuation* refers to methods for approximating a homotopy in order to find an approximation to the solution of a system [1, 21, 12]. In this chapter, several novel methods for numerical continuation are presented in which the derivative is interpreted as the sensitivity of the model to a parameter and the direct inversion of an operator is avoided.

In Section 5.1, numerical continuation is introduced. In Section 5.2 numerical continuation is applied to the algebraic model in (2.2.1). Several algorithms for numerical continuation are developed and applied to the problem. A novel continuation method is shown in Algorithm 5.3 and bounds for its error are given in Theorem 5.4. Finally, in Section 5.3, the continuation methods developed in Section 5.2 are applied to the nonlinear flow problem.

5.1 Background

Consider first a simple example, an equation on $u \in \mathbf{R}$ of the form of Equation (2.0.1).

In order to construct a homotopy, a parameter may be added so that the equation

$$F(\lambda, u) \equiv Lu(\lambda) + \lambda N(u(\lambda)) - f, \tag{5.1.1}$$

is solved for $\lambda \in [0,1]$. The function F provides a homotopy from the solution with $\lambda = 0$ to the solution with $\lambda = 1$. At $\lambda = 0$, the solution $u(0)$ may be found by solving the linear problem

$$Lu(0) = f. \tag{5.1.2}$$

At $\lambda = 1$, the solution $u(1)$ is the solution to equation

$$Lu(1) + \lambda N(u(1)) = f. \tag{5.1.3}$$

The continuation problem is to find a way to proceed from the easily known solution $u_0 = u(0)$ of $F(0,u) = 0$ to the solution $u(1)$ of $F(1,u) = 0$. Assume that $u(\lambda)$ is the unique solution to the equation $F(\lambda, u) = 0$ for each $\lambda \in [0,1]$. The set $\{u(\lambda) : 0 \leq \lambda \leq 1\}$ can be viewed as a curve in $\mathbf{R}^n$ from $u(0)$ to $u(1)$ parameterized by λ. A continuation method finds a sequence of steps along this curve.

If $u(\lambda)$ and $F(\lambda, u)$ are differentiable with respect to λ, then formally differ-

entiating (5.1.1) with respect to λ gives

$$0 = (L + \lambda N'(u)) \{\partial_\lambda u\} + N(u),$$

and solving for $\{\partial_\lambda u\}$ gives

$$\{\partial_\lambda u\} = -(L + \lambda N'(u))^{-1} N(u). \tag{5.1.4}$$

Now, given $\{\partial_\lambda u\}$ we can think of finding $u(\lambda)$ by constructing a pseudo-transient problem, a differential equation to lead us from $u(0)$ to $u(\lambda)$ for a given λ. In this pseudo-transient problem λ plays a role of the "time" variable.

Given the solution to (5.1.2), we arrive at an initial value problem in $\mathbf{R}^n$:

$$\begin{aligned} u_0 &= u(0), \\ \{\partial_\lambda u\} &= -(L + \lambda N'(u))^{-1} N(u). \end{aligned}$$

The solution to this ordinary differential equation will follow the curve

$$\{u(\lambda) : 0 \leq \lambda \leq 1\}$$

and will give the solution $u(1)$ to the nonlinear system in (5.1.3) which is sought. This is called *numerical continuation* [1, 21, 12]. The difficulty in this method is computing the derivative since it appears to involve a nonlinear operator.

Below we present a method in which the derivative is nothing but the sensitivity

of the model to a parameter. With this approach, the direct inversion of the nonlinear operator is avoided.

5.2 Application: An Algebraic Model

In Section 2.2 an algebraic model was given in (2.2.1) which has a solution given in (2.2.2). Consider the homotopy

$$\lambda\beta u(\lambda)|u(\lambda)| + u(\lambda) = -kp, \tag{5.2.1}$$

for $\lambda \in [0,1]$. To construct an initial value problem, it is necessary to differentiate (5.2.1) with respect to λ and to find the initial value $u_0 = u(0)$. The initial value is given by

$$\begin{aligned} -kp &= 0 \cdot \beta u(0)|u(0)| + u(0), \\ &= u(0). \end{aligned}$$

Differentiating and solving for $\{\partial_\lambda u\}$ yields the initial value problem

$$\{\partial_\lambda u\} = -\frac{\beta u(\lambda)|u(\lambda)|}{2\lambda\beta|u(\lambda)| + 1}, \tag{5.2.2}$$

$$u_0 = u(0) = -kp. \tag{5.2.3}$$

This problem is well posed since $2\lambda\beta|u(\lambda)| + 1 \neq 0$. It is possible to use numerical methods to approximate the solution to this initial value problem. Several variants

Algorithm 5.1 (Forward Euler Algebraic Model Continuation)**.** *Given* $N \geq 1$*, let* $\Delta\lambda = \frac{1}{N}$*. Let* $u_0 = -kp$*. For* $i = 1, \ldots, N$*:*

1. *Let* $\lambda_i = i\Delta\lambda$ *and* $\{\partial_\lambda u\}_i = -\frac{\beta u_{i-1}|u_{i-1}|}{2\lambda_{i-1}\beta|u_{i-1}|+1}$.
2. *Let* $u_i = u_{i-1} + \Delta\lambda \{\partial_\lambda u\}_i$.

are developed below. The variants will be based on the Forward Euler method, an explicit Runge-Kutta method, the Newton continuation method, and the novel Quasi-Newton continuation method.

These four methods are tested below. In most circumstances the Quasi-Newton continuation method has performance rivaling the Newton method but at a significantly lower computational cost.

Forward Euler based approach. A numerical method which is easy to implement is the Forward Euler method [12]. The Forward Euler method is a first order accurate method that approximates $u_{i+1} \approx u_i + \Delta\lambda \{\partial_\lambda u\}(u_i)$ where $\Delta\lambda$ is a given step size. This method is presented because it is simple to implement and understand. The implementation, given in Algorithm 5.1, can help guide the use of more complicated approximations. The error from solving using the Forward Euler method is first order, $O(\Delta\lambda)$. To choose $\Delta\lambda$, pick an acceptable tolerance, τ, and pick $\Delta\lambda < \tau$.

Runge-Kutta based approach. An obvious way to improve the continuation method above is to use a higher order method. One possibility is to use an explicit

Runge-Kutta method [12]. High order explicit Runge-Kutta methods are available in many software packages including Scipy. To use the Runge-Kutta method one need only provide the initial value u_0 and a function which will compute the derivative at given u and λ. An advantage of the Runge-Kutta methods provided in software packages like Scipy [34, 29] is their ability to adaptively control the step size.

Newton continuation. Another improvement on the Forward Euler continuation method is the Newton continuation method. See [21] for a discussion of this method. This method allows us to take larger step sizes at the cost of solving a nonlinear system. The method can be viewed as a predictor-corrector algorithm. In the predictor step, the Forward Euler approximation is found. In the corrector step, a nonlinear solver (like Newton's method) is used to find the true solution. Since the Forward Euler method should give a reasonable approximation to the solution sought with the nonlinear solver, fewer iterations should be necessary. The solution, at $\lambda = 1$, that is obtained using the Newton continuation method is as numerically accurate as solving only with the nonlinear solver since that is the final step of the algorithm. Newton continuation for a general model of the form (5.1.1) is shown in Algorithm 5.2.

Quasi-Newton continuation. A simplification of the Newton continuation method, which will be referred to as Quasi-Newton continuation, is to solve a related linear system rather than the full nonlinear system in the last step. Theorem 5.4 shows that Quasi-Newton continuation will maintain accuracy. It is demonstrated, below,

---

Algorithm 5.2 (Newton Continuation). *Given $N \geq 1$, let $\Delta\lambda = \frac{1}{N}$. Let $u(0) = u_0$ be the solution to the linear problem $Lu(0) = f$. For $i = 1, \ldots, N$:*

1. *Let $\lambda_i = i\Delta\lambda$ and $\{\partial_\lambda u\}_i = -\left(L + \lambda_i N'(u_i)\right)^{-1} N(u_i)$.*
2. *Let $\overline{u}_i = u_{i-1} + \Delta\lambda \{\partial_\lambda u\}_i$.*
3. *Solve the nonlinear problem $Lu_i + \lambda_i N(u_i) = f$ using $\overline{u}_i$ as the initial guess.*

Algorithm 5.3 (Quasi-Newton Continuation). *Given $N \geq 1$, let $\Delta\lambda = \frac{1}{N}$. Let $u(0) = u_0$ be the solution to the linear problem $Lu(0) = f$. For $i = 1, \ldots, N$:*

1. *Let $\lambda_i = i\Delta\lambda$ and $\{\partial_\lambda u\}_i = -\left(L + \lambda_i N'(u_i)\right)^{-1} N(u_i)$.*
2. *Let $\overline{u}_i = u_{i-1} + \Delta\lambda \{\partial_\lambda u\}_i$.*
3. *Solve the linear problem $Lu_i = f - \lambda_i N(\overline{u}_i)$ for u_i.*

---

that it performs almost as well as Newton continuation. Quasi-Newton continuation for a general model of the form (5.1.1) is shown in Algorithm 5.3.

Theorem 5.4. *Suppose $N(\cdot)$ is Lipschitz continuous with constant $K \geq 0$. Fix $\lambda \in [0, 1]$ and let u be the solution to*

$$Lu + \lambda N(u) = f. \tag{5.2.4}$$

If u_{QN} is the Quasi-Newton approximation to u, then $\|u - u_{QN}\| = O(h)$. Further, if $K\|L^{-1}\| \leq 1$, then $\|u - u_{QN}\| \leq \|u - u_{FE}\|$ where u_{FE} is the Forward Euler approximation to u.

Proof. Let u_{FE} be the Forward Euler approximation to u, then $u_{FE} = u + O(h)$.

By definition of the Quasi-Newton approximation to u, u_{QN} satisfies

$$Lu_{QN} + \lambda N\left(u_{FE}\right) = f. \tag{5.2.5}$$

Subtracting (5.2.5) from (5.2.4) and rearranging yields

$$u - u_{QN} = \lambda L^{-1}\left(N\left(u_{FE}\right) - N\left(u\right)\right).$$

Since N is Lipschitz continuous,

$$\left\|u - u_{QN}\right\| \leq \lambda K \left\|L^{-1}\right\| \left\|u_{FE} - u\right\|.$$

Since $u_{FE} = u + O\left(h\right)$, $\left\|u_{FE} - u\right\| = O\left(h\right)$. If $K\left\|L^{-1}\right\| \leq 1$, then since $0 \leq \lambda \leq 1$,

$$\left\|u - u_{QN}\right\| \leq \left\|u_{FE} - u\right\|.$$

□

5.2.1 A Simple Algebraic Example

Let $\beta = 2$ and $kp = -1$, then the solution u is sought to

$$2u\left|u\right| + u = 1.$$

The exact solution, from (2.2.2), is clearly $u = \frac{1}{2}$.

To find the solution using the present methods, the initial value problem

$$\{\partial_\lambda u\} = -\frac{\beta u\,|u|}{2\lambda\beta\,|u|+1},$$
$$u\,(0) = 1,$$

will be solved. In Figure 5.2.1, the error in the Forward Euler method and Quasi-Newton method for $N \in [0, 10]$ is shown. The error decreases rapidly with increasing N for both. The Forward Euler method performs better for smaller values of N while the Quasi-Newton method performs better for larger values.

The solution path for the Forward Euler, Quasi-Newton, and the Scipy ODE suite are shown in Figure 5.2.2. All methods use $N = 10$ steps. All solution paths are close to each other, but the Scipy ODE suite and Quasi-Newton methods perform slightly better than the Forward Euler method.

5.3 Continuation Method with Sensitivity for Fluid Flow in Porous Media

Recall the equations governing fluid flow in porous media (2.3.5)-(2.3.8). For clarity of presentation, let κ_L be the linear part of the resistance and let $\kappa_{NL} = \kappa_{NL}\,(|u|)$ be the non-linear part of the resistance. Then equations (2.3.5)-(2.3.8) may have the parameter λ added so that

$$\nabla \cdot u = f\,(x)\,, \qquad \text{in } \Omega, \tag{5.3.1}$$

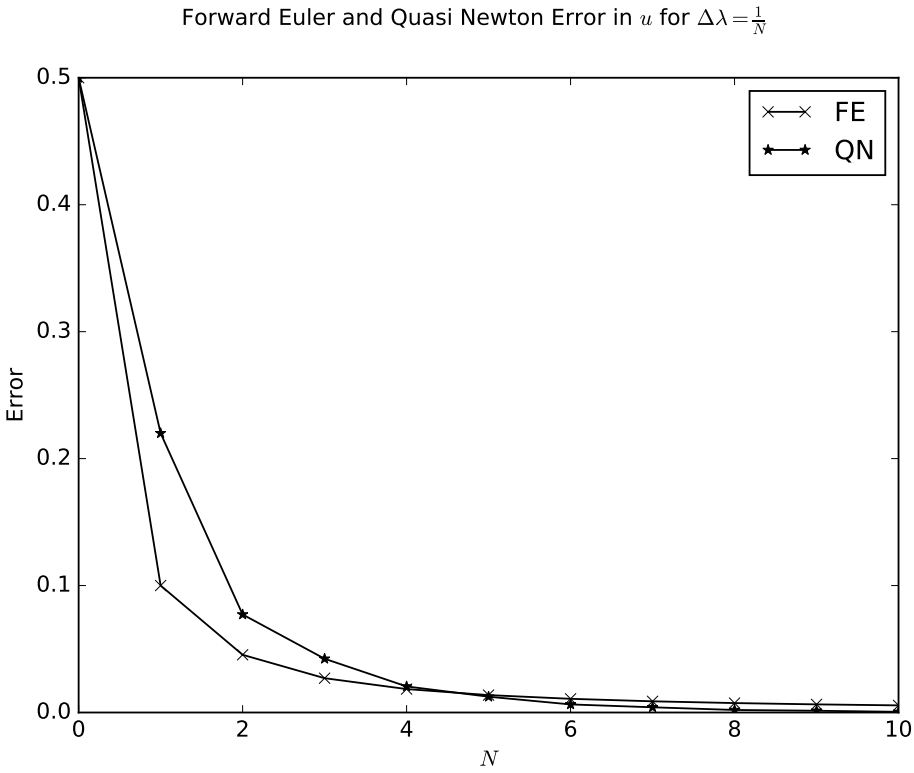

Figure 5.2.1: Error in Forward Euler and Quasi-Newton Methods for Algebraic Example: Forward Euler performs better for smaller N while Quasi-Newton performs better for larger N.

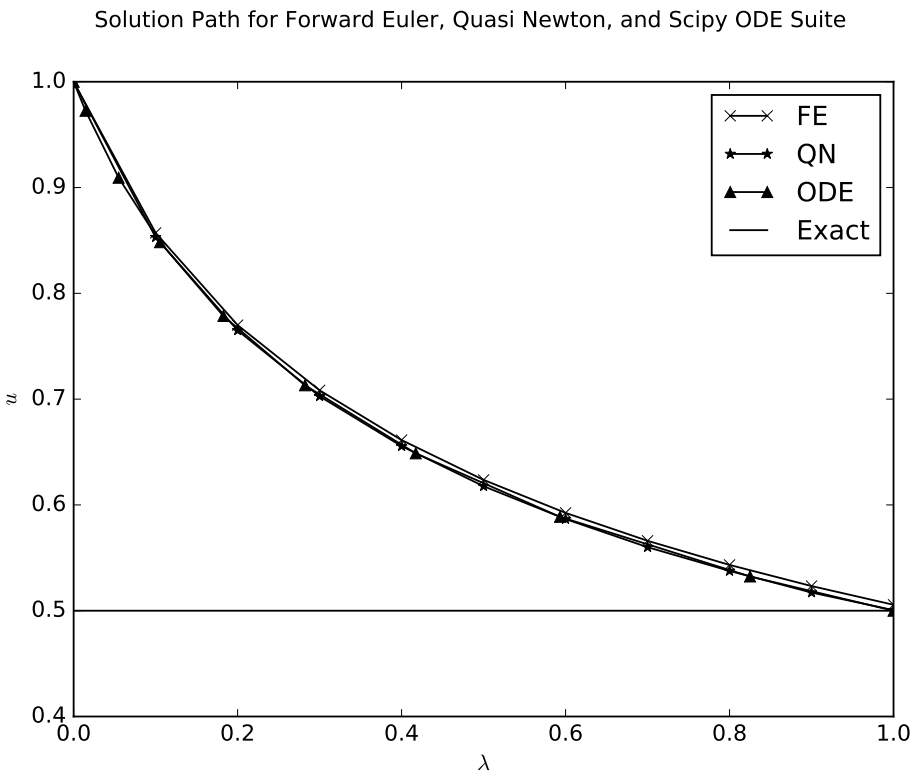

Figure 5.2.2: Solution Paths for Forward Euler, Quasi-Newton, and the Scipy ODE Suite on the Algebraic Example: Quasi-Newton and Scipy ODE Suite perform better than Forward Euler

$$\kappa_L u + \lambda \kappa_{NL}(|u|) u = -\nabla p + g(x), \qquad \text{in } \Omega, \tag{5.3.2}$$

$$p = p_D(x), \qquad \text{on } \Gamma^D, \tag{5.3.3}$$

$$u \cdot n = u_N(x), \qquad \text{on } \Gamma^N, \tag{5.3.4}$$

where $\lambda \in [0,1]$. Equations (5.3.1)-(5.3.4) define a homotopy between a linear Darcy flow problem at $\lambda = 0$ and the original nonlinear flow model problem at $\lambda = 1$.

5.3.1 Sensitivity to λ

We find the sensitivity to the parameter λ, formally, by differentiating with respect to λ. Let

$$\kappa_S(u, |u|, \lambda) = \kappa_L + \lambda \kappa_{NL}(|u|) + \lambda \partial_{|u|} \kappa_{NL}(\partial_u |u|(u)),$$

then the resulting system is

$$\nabla \cdot \{\partial_\lambda u\} = 0, \qquad \text{in } \Omega, \tag{5.3.5}$$

$$\kappa_S(u, |u|, \lambda) \{\partial_\lambda u\} = -\nabla \{\partial_\lambda p\} - \kappa_{NL}(|u|) u, \qquad \text{in } \Omega, \tag{5.3.6}$$

$$\{\partial_\lambda p\} = 0, \qquad \text{on } \Gamma^D, \tag{5.3.7}$$

$$\{\partial_\lambda u\} \cdot n = 0, \qquad \text{on } \Gamma^N. \tag{5.3.8}$$

This is a flow system for the sensitivities $\{\partial_\lambda u\}$ and $\{\partial_\lambda p\}$. An important fact about that system is that it is linear. Also, there is no explicit inversion of an operator.

5.3.2 Numerical Continuation

As described above, the sensitivity to λ will be used to find the solution to the nonlinear flow model. Let u_0, p_0 solve

$$\nabla \cdot u_0 = f(x), \qquad \text{in } \Omega, \tag{5.3.9}$$

$$\kappa_L u_0 = -\nabla p_0 + g(x), \qquad \text{in } \Omega, \tag{5.3.10}$$

$$p_0 = p_D(x), \qquad \text{on } \Gamma^D, \tag{5.3.11}$$

$$u_0 \cdot n = u_N(x), \qquad \text{on } \Gamma^N. \tag{5.3.12}$$

The task at hand is to find a path from u_0, p_0 to the solution to the nonlinear flow model. We will approximate $\{\partial_\lambda u\}$ and $\{\partial_\lambda p\}$ using (5.3.5)-(5.3.8). As explained before in Section 5.2, there are several possibilities for solving the resulting initial value problem including the Forward Euler method, Runge-Kutta methods, Quasi-Newton, and Newton Continuation. The results below will show that the Quasi-Newton is a superior method for this problem.

The Forward Euler method to trace the curves from the solution to the linear problem p_0, u_0 to the solution to the nonlinear problem p_N, u_N is presented in Algorithm 5.5. An important consideration in choosing N, is to ensure an acceptable level of accuracy. Since the Forward Euler method is first order accurate (in $\Delta\lambda$) and we expect the error in the pressures to be second order accurate (in Δx and Δy), it is necessary to use $\Delta\lambda$ sufficiently small so that the continuation error is not larger than the error from solving the flow problem. In short, it is sufficient

Algorithm 5.5 (Forward Euler Flow System Continuation)**.** *Given $N \geq 1$, let $\Delta\lambda = \frac{1}{N}$. Let p_0, u_0 solve the linear flow system* (5.3.9)-(5.3.12)*. For $i = 1, \ldots, N$:*

1. *Let $\lambda_i = i\Delta\lambda$, $\kappa_i = \left[\kappa_L + \lambda_{i-1}\kappa_{NL}\left(|u_{i-1}|\right) + \lambda_{i-1}\partial_{|u|}\kappa_{NL}\left(\partial_u |u| \left(u_{i-1}\right)\right)\right]$, and $g_i = -\kappa_{NL}\left(|u_{i-1}|\right) u_{i-1}$ and solve*

$$\begin{aligned} \nabla \cdot \left\{\partial_\lambda u\right\}_i &= 0, && \text{in } \Omega, \\ \kappa_i \left\{\partial_\lambda u\right\}_i &= -\nabla \left\{\partial_\lambda p\right\}_i + g_i, && \text{in } \Omega, \\ \left\{\partial_\lambda p\right\}_i &= 0, && \text{on } \Gamma^D, \\ \left\{\partial_\lambda u\right\}_i \cdot n &= 0, && \text{on } \Gamma^N, \end{aligned}$$

   for $\left\{\partial_\lambda p\right\}_i, \left\{\partial_\lambda u\right\}_i$.

2. *Let $p_i = p_{i-1} + \Delta\lambda \left\{\partial_\lambda p\right\}_i$ and $u_i = u_{i-1} + \Delta\lambda \left\{\partial_\lambda u\right\}_i$.*

that $\Delta\lambda \leq \min\left\{(\Delta x)^2, (\Delta y)^2\right\}$. If $N > \max\left\{N_x^2, N_y^2\right\}$, then

$$\Delta\lambda = \frac{1}{N} < \frac{1}{\max\left\{N_x^2, N_y^2\right\}} = \min\left\{(\Delta x)^2, (\Delta y)^2\right\}.$$

This small step size might remove any advantage of solving the nonlinear problem in this way. Fortunately, there are higher order methods which allow larger steps and offer adaptive step sizes. Also, a larger step size might still yield a reasonable result as an initial guess for a nonlinear solver.

The Runge-Kutta method is also implemented to solve the flow system. The initial values will be as above and an algorithm to compute the derivative is given in Algorithm 5.6.

The Newton continuation method is given in Algorithm 5.7. The Quasi-Newton continuation method is demonstrated in Algorithm 5.8. A reasonable tolerance can

Algorithm 5.6 (Flow System Derivative)**.** *Given u, p, and λ let $\kappa_\lambda = \left[\kappa_L + \lambda\kappa_{NL}\left(|u|\right) + \lambda\partial_{|u|}\kappa_{NL}\left(\partial_u\left|u\right|\left(u\right)\right)\right]$ and $g_\lambda = -\kappa_{NL}\left(|u|\right)u$ and solve*

$$\begin{aligned} \nabla\cdot\left\{\partial_\lambda u\right\} &= 0, && \text{in } \Omega, \\ \kappa_\lambda\left\{\partial_\lambda u\right\} &= -\nabla\left\{\partial_\lambda p\right\} + g_\lambda, && \text{in } \Omega, \\ \left\{\partial_\lambda p\right\} &= 0, && \text{on } \Gamma^D, \\ \left\{\partial_\lambda u\right\}\cdot n &= 0, && \text{on } \Gamma^N, \end{aligned}$$

for $\left\{\partial_\lambda p\right\}, \left\{\partial_\lambda u\right\}$.

be selected using similar derivations as for the Forward Euler continuation since this is a minor modification to that method. If the error in the linear or nonlinear solver is kept less than the expected error in the pressures and the error from the Forward Euler method, then the approximation should be reasonable. Since the residual is used rather than the true error, it is safer to use a smaller tolerance to assure a reasonable level of error.

5.3.3 Example Problem

To evaluate the algorithms developed above, an an example problem is defined by

$$\begin{aligned} \kappa\left(u\right) =& \begin{bmatrix} 2+\cos\left(xy\right) & \\ & 2+\sin\left(xy\right) \end{bmatrix} + \begin{bmatrix} 10\left|u_1\right| & \\ & 15\left|u_2\right| \end{bmatrix}, \\ p\left(1,y\right) =& p\left(x,1\right) = p\left(0,y\right) = p\left(x,0\right) = 0, \\ f\left(x,y\right) =& 4\pi y\cos\left(4\pi xy\right) + 3\pi x\cos\left(3\pi xy\right), \end{aligned}$$

Algorithm 5.7 (Newton Flow System Continuation)**.** *Given $N \geq 1$, let $\Delta\lambda = \frac{1}{N}$. Let p_0, u_0 solve the linear flow system* (5.3.9)-(5.3.12)*. For $i = 1, \ldots, N$:*

1. *Let $\lambda_i = i\Delta\lambda$, $\kappa_i = \left[\kappa_L + \lambda_i \kappa_{NL}\left(|u_{i-1}|\right) + \lambda_i \partial_{|u|}\kappa_{NL}\left(\partial_u |u|\left(u_{i-1}\right)\right)\right]$, and $g_i = -\kappa_{NL}\left(|u_{i-1}|\right) u_{i-1}$ and solve*

$$\begin{aligned} \nabla \cdot \left\{\partial_\lambda u\right\}_i &= 0, && \text{in } \Omega, \\ \kappa_i \left\{\partial_\lambda u\right\}_i &= -\nabla \left\{\partial_\lambda p\right\}_i + g_i, && \text{in } \Omega, \\ \left\{\partial_\lambda p\right\}_i &= 0, && \text{on } \Gamma^D, \\ \left\{\partial_\lambda u\right\}_i \cdot n &= 0, && \text{on } \Gamma^N, \end{aligned}$$

   for $\left\{\partial_\lambda p\right\}_i, \left\{\partial_\lambda u\right\}_i$.

2. *Let $p_i^* = p_{i-1} + \Delta\lambda \left\{\partial_\lambda p\right\}_i$ and $u_i^* = u_{i-1} + \Delta\lambda \left\{\partial_\lambda u\right\}_i$.*

3. *Solve the nonlinear system*

$$\begin{aligned} \nabla \cdot u_i &= f\left(x\right), && \text{in } \Omega, \\ \kappa_L u_i + \lambda_i \kappa_{NL}\left(|u_i|\right) u_i &= -\nabla p_i + g\left(x\right), && \text{in } \Omega, \\ p_i &= p_D\left(x\right), && \text{on } \Gamma^D, \\ u_i \cdot n &= u_N\left(x\right), && \text{on } \Gamma^N, \end{aligned}$$

   for u_i and p_i using the initial guess p_i^* and u_i^*.

Algorithm 5.8 (Quasi-Newton Flow System Continuation)**.** *Given $N \geq 1$, let $\Delta\lambda = \frac{1}{N}$. Let p_0, u_0 solve the linear flow system* (5.3.9)-(5.3.12)*. For $i = 1, \ldots, N$:*

1. *Let $\lambda_i = i\Delta\lambda$, $\kappa_i = \left[\kappa_L + \lambda_i \kappa_{NL}\left(|u_{i-1}|\right) + \lambda_i \partial_{|u|}\kappa_{NL}\left(\partial_u |u|\left(u_{i-1}\right)\right)\right]$, and $g_i = -\kappa_{NL}\left(|u_{i-1}|\right) u_{i-1}$ and solve*

$$\begin{aligned} \nabla \cdot \left\{\partial_\lambda u\right\}_i &= 0, && \text{in } \Omega, \\ \kappa_i \left\{\partial_\lambda u\right\}_i &= -\nabla \left\{\partial_\lambda p\right\}_i + g_i, && \text{in } \Omega, \\ \left\{\partial_\lambda p\right\}_i &= 0, && \text{on } \Gamma^D, \\ \left\{\partial_\lambda u\right\}_i \cdot n &= 0, && \text{on } \Gamma^N, \end{aligned}$$

   for $\left\{\partial_\lambda p\right\}_i, \left\{\partial_\lambda u\right\}_i$.

2. *Let $p_i^* = p_{i-1} + \Delta\lambda \left\{\partial_\lambda p\right\}_i$ and $u_i^* = u_{i-1} + \Delta\lambda \left\{\partial_\lambda u\right\}_i$.*

3. *Solve the linear system*

$$\begin{aligned} \nabla \cdot u_i &= f\left(x\right), && \text{in } \Omega, \\ \kappa_L u_i + \lambda_i \kappa_{NL}\left(|u_i^*|\right) u_i &= -\nabla p_i + g\left(x\right), && \text{in } \Omega, \\ p_i &= p_D\left(x\right), && \text{on } \Gamma^D, \\ u_i \cdot n &= u_N\left(x\right), && \text{on } \Gamma^N, \end{aligned}$$

   for u_i and p_i. If an iterative method is used to solve the linear system, use the initial guess p_i^* and u_i^*.

$$g(x,y) = \begin{bmatrix} (2+\cos(xy)+10\,|\sin(4\pi xy)|)\sin(4\pi xy) \\ (2+\sin(xy)+15\,|\sin(3\pi xy)|)\sin(3\pi xy) \end{bmatrix} + \begin{bmatrix} 2\pi\cos(2\pi x)\sin(2\pi y) \\ 2\pi\sin(2\pi x)\cos(2\pi y) \end{bmatrix}.$$

Consider the system, as usual,

$$\kappa(u)\,u + \nabla p = g,$$
$$\nabla \cdot u = f,$$

with the given Dirichlet boundary conditions. It is easy to verify that

$$u = [\sin(4\pi xy), \sin(3\pi xy)]^T,$$
$$p = \sin(2\pi x)\sin(2\pi y),$$

is the solution to this problem. Also, note that

$$\kappa_{NL}(u)\,u = \begin{bmatrix} 10\,|u_1| & \\ & 15\,|u_2| \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix},$$
$$\partial_{|u|}\kappa_{NL}(\partial_u |u|(u)) = \begin{bmatrix} 10 & \\ & 15 \end{bmatrix} \begin{bmatrix} |u_1| & \\ & |u_2| \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}.$$

Further, the resistance term and gravity-like source term for the continuation method are given by

$$\kappa_\lambda(\lambda, u) = \begin{bmatrix} 2+\cos(xy) & \\ & 2+\sin(xy) \end{bmatrix} + 2\lambda \begin{bmatrix} 10|u_1| & \\ & 15|u_2| \end{bmatrix},$$

$$g_\lambda == -\kappa_{NL}(|u|)\, u,$$

respectively.

The methods and example problem are implemented in Listing 13 in the Appendix. The example problem is solved with $N_x = N_y = 50$. In this section the error from the numerical method is not the focus. Instead, the modeling error is the primary interest. Unfortunately, there will be numerical error in the solution which is related to the grid spacing. The solution is shown in Figure 5.3.1. Note that since this problem has forcing functions, it is expected that the velocity is not perpendicular to the level lines of the pressure.

To compare the methods, the L^2 error in u and p is computed using various $\Delta\lambda$. A plot is shown with logarithmic scale in Figure 5.3.2. The Forward Euler method performs far worse than the other methods with more error with $\Delta\lambda = \frac{1}{49}$ than the Quasi-Newton method has with $\Delta\lambda = \frac{1}{7}$. The Newton method converges immediately to the solution since the full nonlinear system is solved regardless of the choice of $\Delta\lambda$. The error in the Newton method is set by the stopping criteria for the nonlinear solver which is set to $\frac{1}{h^2}$, the expected error due to the discretization. The nonlinear solver will only converge quickly if the initial guess is close enough

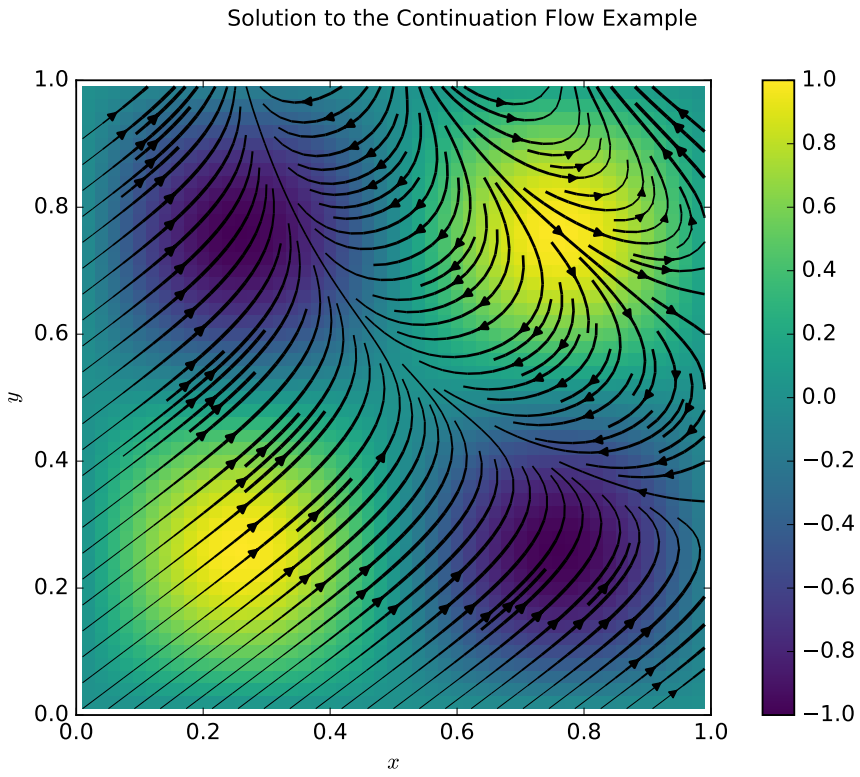

Figure 5.3.1: Solution to Flow Example Problem with $N_x = N_y = 50$.

to the solution. The Quasi-Newton method performs very well. With $\Delta\lambda = \frac{1}{12}$, the Quasi-Newton method has converged to the accuracy expected for the spatial discretization.

Another comparison is the relative computational cost, shown in Figure 5.3.3. In this figure, for a given $N = \frac{1}{\Delta\lambda}$, the total number of systems that are solved is shown. For the Forward Euler method, there is one linear system to solve for each iteration. The Quasi-Newton method requires two linear systems to be solved (one for the predictor step and another for the corrector step). The Newton method requires that a nonlinear system to be solved. Something that is obscured by this figure is that the cost of the Newton iterations is higher than the cost of the Quasi-Newton and Forward Euler iterations. The Newton method relies on a nonlinear solver at each step which includes extra overhead as well as solving a linear system. Since the number of iterations for the Quasi-Newton method is smaller than the Newton and the cost per iteration is less and since the accuracy is comparable, it seems that the Quasi-Newton method is a superior method. It is also worth noting that the accuracy can be brought to the same level as the Newton method by using the output as an initial guess in the nonlinear solver. Since a small number of iterations of the Quasi-Newton method leads to a small error, the nonlinear solver should converge quickly.

To compare the Scipy ODE suite, the error is compared with similar $\Delta\lambda$ in Table 5.3.1. The Scipy ODE suite performs slightly better than the Forward Euler method but is greatly out-performed by the Newton and Quasi-Newton methods.

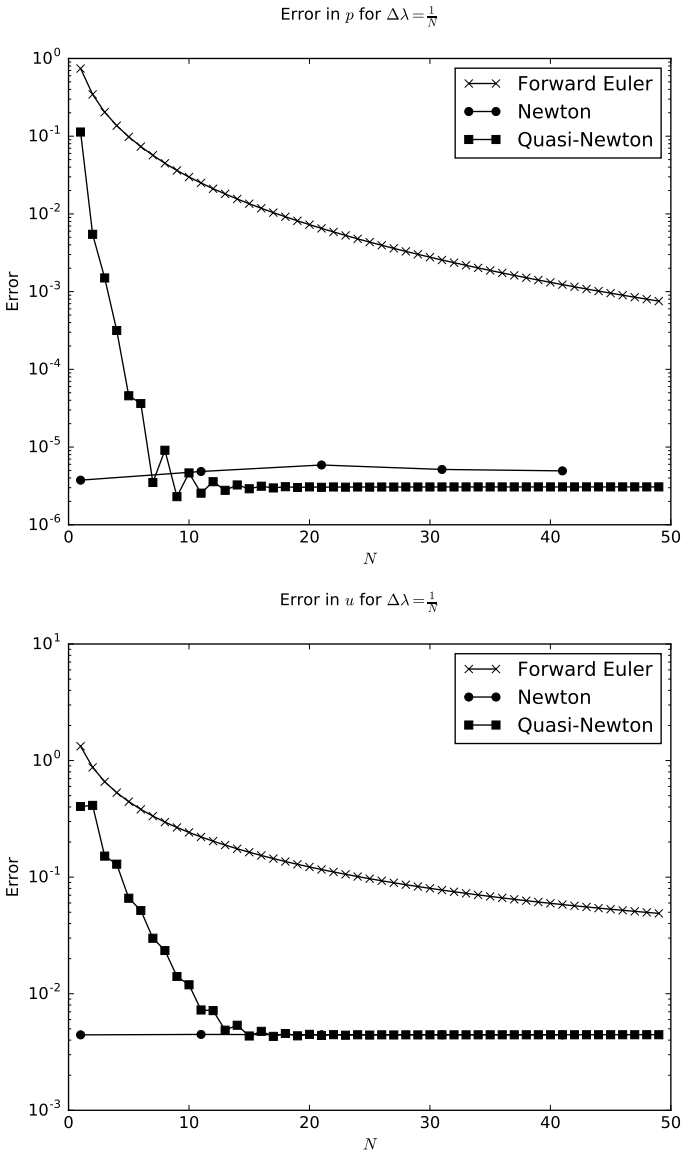

Figure 5.3.2: L^2 Error in p and u.
The Newton method solves the full system with tolerance $\frac{1}{h^2}$ for the pressure, where h is the grid spacing, regardless of N so the error is approximately constant. The linear flow solver is expected to give the same error as the Newton solver's tolerance.

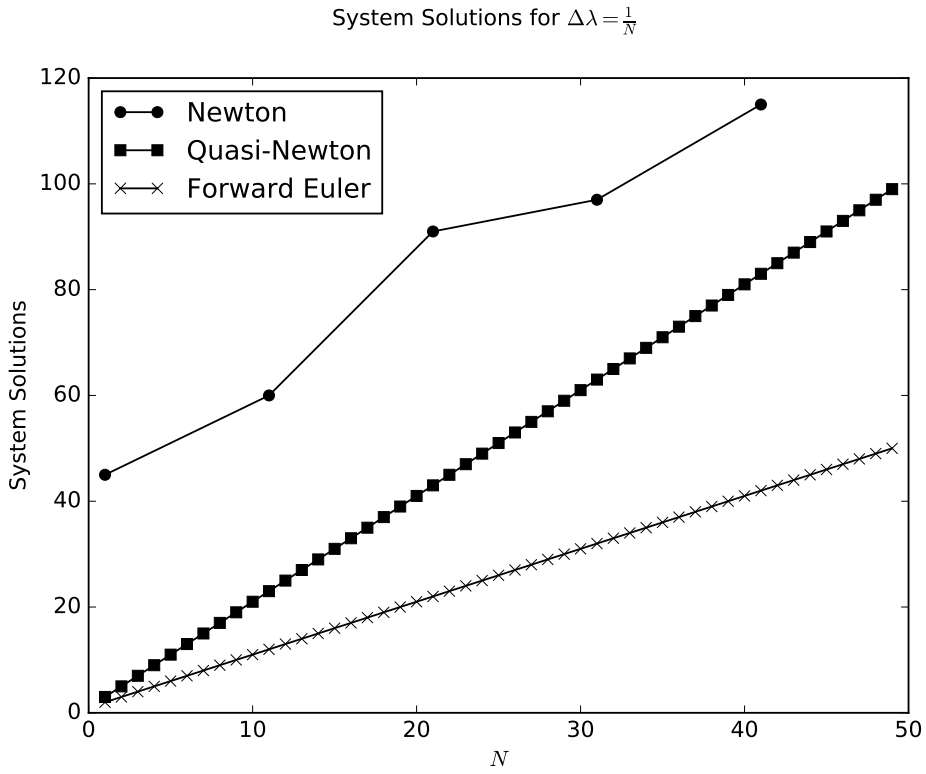

Figure 5.3.3: Number of System Solutions.
The Newton method takes more system solutions and the solutions are more costly than the Quasi-Newton and the Forward Euler methods. The Quasi-Newton method takes approximately double the number of solutions as the Forward Euler method.

	Error in p	Error in u
Forward Euler	8.86×10^{-3}	0.393
Scipy ODE	7.71×10^{-3}	0.350
Quasi-Newton	1.08×10^{-3}	0.0678
Newton	1.12×10^{-3}	0.0677

Table 5.3.1: Error with $N = 66$ Iterations.
The Forward Euler method and the Scipy ODE suite produce similar error. The Quasi-Newton and Newton methods produce similar error.

5.4 Summary

In this chapter, numerical continuation was discussed. In Section 5.1, a homotopy was presented and an initial value problem for following the homotopy was given. In Section 5.2, several algorithms were developed to solve the initial value problem from Section 5.1. The methods for solving the initial value problem are referred to as numerical continuation.

The simplest method, based on the forward Euler method, is presented in Algorithm 5.1 for a general nonlinear problem and it is applied to the algebraic example from Chapter 2 in Section 5.2.1. In Section 5.3, the method is applied to the nonlinear flow model from Chapter 2. The algorithm is shown in Algorithm 5.5. An advantage of the method is that it has easily computable error bounds, although the accuracy is far less than some of the other methods.

Another method, based on the same idea as the forward Euler method, is to use a Runge-Kutta method and software libraries. Commonly available software libraries will solve an initial value problem given a derivative function and the initial value. The numerical experiments show that the results are similar to the forward Euler method.

The standard method for numerical continuation, Newton continuation, is presented in 5.2. The Newton continuation will converge if the nonlinear flow solver from Chapter 2 converges since the last step is solving the full nonlinear system. The disadvantage is that the computational cost for so many nonlinear solutions may be higher than necessary.

Finally, a novel Quasi-Newton continuation method is provided in Algorithm 5.3 and it is applied to the algebraic problem in Section 5.2.1. In Section 5.3, the method is applied to the nonlinear flow model from Chapter 2. The algorithm is shown in Algorithm 5.8. In Theorem 5.4, the error in the Quasi-Newton method is shown to be no worse than the forward Euler method. In practical experiments, the error is much less than the error from the forward Euler method. In the practical experiments, this method is clearly superior. Since only linear systems are solved, each iteration is relatively fast when compared to the Newton continuation method while it also has much faster convergence than the forward Euler method.

Chapter 6 Model Adaptivity Using Sensitivity Analysis

When modeling a physical problem, such as fluid flow and transport in porous media, there are several possible sources of error. First, measurements need to be made which can only have a finite resolution and often present processing problems [14, 36, 60]. There is also numerical error due to the choice of numerical method for approximating the solution as discussed in Chapter 2. The topic of this chapter is *model adaptivity*, which is concerned with the error due to the choice of physical model [56, 53, 7, 54, 55, 52]. An example of such a choice is the choice between the linear Darcy model for fluid flow and the nonlinear flow model presented in Chapter 2. Approximating the *modeling error*, that is the error incurred from the choice of model, aids in the choice of model. This chapter will show one method for approximating the modeling error.

In [52, 55, 54, 7, 53, 56] residual based error approximations for a quantity of interest are discussed. One difficulty in the methods in those works is that the error in the solution to the problem and an adjoint problem have to be estimated. No general framework for finding the estimates is presented. In this section, sensitivity analysis, as in Chapter 3.4, will be applied to the problem of approximating the error in a quantity of interest.

In Section 6.1, the framework for approximating the error in a quantity of interest is developed. It is proposed that a homotopy between a linear model and

a nonlinear model be followed as in Chapter 5. It is proposed that the sensitivity of the quantity of interest is computed for various values of the continuation parameter and the values are integrated with respect to the continuation parameter. The sensitivity of the quantity of interest may be computed using either FS or AS from Chapter 3. The integration with respect to the continuation parameter may be carried out using techniques such as the left hand rule, the trapezoidal rule, or Gaussian quadrature. In Section 6.2, the methods developed in Section 6.1 are applied to the nonlinear flow model from Chapter 2.

6.1 Background

Recall, a homotopy for a system of the form (2.0.1) is given in (5.1.1). Suppose that $G(u)$ is a quantity of interest. The error in the quantity of interest $G(\cdot)$ is given by

$$E_G = G(u(1)) - G(u(0)),$$

where $u(0)$ is the solution to (5.1.2) and $u(1)$ is the solution to (5.1.3). Let

$$\overline{G}(\lambda) = G(u(\lambda)),$$

then $\overline{G}$ is a real valued function defined on $[0,1]$. Then

$$\frac{d\overline{G}}{d\lambda} = \partial_u G \{\partial_\lambda u\},$$

which may be computed using either FS from Section 3.3 or AS from Section 3.4.

Theorem 6.1. *If* $\overline{G}(\cdot)$ *is differntiable on* $[0,1]$ *and* $\frac{d\overline{G}}{d\lambda}$ *is Riemann integrable on* $[0,1]$*, then*

$$\begin{aligned} E_G &= \overline{G}(1) - \overline{G}(0) \\ &= \int_0^1 \frac{d\overline{G}}{d\lambda} d\lambda. \end{aligned} \tag{6.1.1}$$

Proof. Apply the Fundamental Theorem of Calculus [17]. □

The methods in this chapter are based on integrating $\frac{d\overline{G}}{d\lambda}$ over the interval $\lambda \in [0,1]$. The simplest method for approximating the integral is the left hand rule which results in

$$E_G \approx \frac{d\overline{G}}{d\lambda}(0).$$

In the following section the trapezoidal rule will also be applied. Another option for approximating the integral is to use Gaussian quadrature. First, observe that

$$\int_0^1 \frac{d\overline{G}}{d\lambda}(\lambda)\, d\lambda = \frac{1}{2}\int_{-1}^1 \frac{d\overline{G}}{d\lambda}\left(\frac{\lambda+1}{2}\right) d\lambda.$$

Gaussian quadrature approximates

$$\int_{-1}^1 f(x)\, dx \approx \sum_{i=1}^n w_i f(x_i),$$

for some weights w_i and nodes x_i. The weights w_i and nodes x_i are chosen so that the approximation is exact for polynomials of degree $2n-1$ or less. A complete

treatment of Gaussian quadrature may be found in [3, 12]. The nodes correspond with the roots of orthogonal polynomials. One example of orthogonal polynomials that may be used are the Legendre polynomials. In that case, the nodes and the weights are available through calls to library functions [34]. The only difficulty is evaluating at the correct points $\lambda_i = \frac{x_i+1}{2}$ where x_i are the nodes above.

6.2 Fluid Flow in Porous Media

Recall the equations governing fluid flow in porous media (2.3.5)-(2.3.8). For clarity of presentation, let κ_L be the linear part of the resistance and let $\kappa_{NL} = \kappa_{NL}(|u|)$ be the non-linear part of the resistance. Recall that a homotopy between the linear Darcy model at $\lambda = 0$ and the nonlinear flow model at $\lambda = 1$ may be constructed as in (5.3.1)-(5.3.4). Further, recall that the sensitivity to λ satisfies (5.3.5)-(5.3.8).

Let $|\Omega|$ represent the area of Ω. Let

$$H_1(u) = \int_\Omega \xi_1(x)\, u(x)\, dx$$

where $\xi_1(x) = \left[\frac{1}{|\Omega|}, 0\right]^T$ and

$$H_2(u) = \int_\Omega \xi_2(x)\, u(x)\, dx$$

where $\xi_2(x) = \left[0, \frac{1}{|\Omega|}\right]^T$. These quantities of interest correspond, respectively, to the average value of u_1 and u_2 in Ω.

Two methods to compute the sensitivity of the quantities of interest will be used: forward sensitivity analysis (FS) and adjoint sensitivity analysis (AS). Recall that in FS the sensitivity of the solution is found by solving (3.3.5)-(3.3.8) while in AS the system (3.4.1)-(3.4.4) is solved which avoids the direct computation of the sensitivity of the solution. Both methods use the solution to the system (5.3.1)-(5.3.4) when computing the sensitivity. The left hand rule, the trapezoidal rule, and Gaussian quadrature will be applied to approximate the integral in (6.1.1).

The left hand rule approximates

$$E_{H_i} \approx \{\partial_\lambda H_i\}|_{\lambda=0}\,.$$

The left hand rule requires the solution $u\,(0)$ to the Darcy system. In addition to the solution $u\,(0)$ the FS method will require the solution to (5.3.1)-(5.3.4) with $\lambda = 0$ whereas the AS method will require the solution to (3.4.1)-(3.4.4) with $\lambda = 0$. The FS left hand rule method is outlined in Algorithm 6.2 and the AS left hand rule method is outlined in Algorithm 6.3.

The trapezoidal rule and Gaussian quadrature will be used to approximate the integral in (6.1.1). The FS method will require the value of $\{\partial_\lambda u\}\,(\lambda)$ which are found by solving (5.3.1)-(5.3.4) for several values of λ while the AS method will require the solution to the adjoint problem (3.4.1)-(3.4.4) for several values of λ. Both of the systems (5.3.1)-(5.3.4) and (3.4.1)-(3.4.4) use the solution $u\,(\lambda)$. To get an approximation for $u\,(\lambda)$, the Quasi-Newton method from Chapter 5 will be

Algorithm 6.2. *Let p_0, u_0 solve the linear flow system (5.3.9)-(5.3.12).*

1. *Solve for $\{\partial_\lambda p\}$ and $\{\partial_\lambda u\}$ the linear system*

$$\begin{aligned} \nabla \cdot \{\partial_\lambda u\} &= 0, && \text{in } \Omega, \\ \left[\partial_{|u|}\kappa\left(\partial_u |u|\left(u_0\right)\right) + \kappa\left(0; |u_0|\right)\right]\{\partial_\lambda u\} &= -\nabla\{\partial_\lambda p\} - \{\partial_\pi \kappa\} u, && \text{in } \Omega, \\ \{\partial_\lambda p\} &= 0, && \text{on } \Gamma^D, \\ \{\partial_\lambda u\} \cdot n &= 0, && \text{on } \Gamma^N. \end{aligned}$$

2. *Compute $\{\partial_\lambda H\}(0) = \int_\Omega \xi(x)\{\partial_\lambda u\}\, dx$.*

Approximate $E_H \approx \{\partial_\lambda H\}(0)$.

Algorithm 6.3. *Let p_0, u_0 solve the linear flow system (5.3.9)-(5.3.12).*

1. *Solve for μ and v the linear system*

$$\begin{aligned} \nabla \cdot v &= 0, && \text{in } \Omega, \\ \left[\partial_{|u|}\kappa\left(\partial_u |u|\left(u_0\right)\right) + \kappa\left(0; |u_0|\right)\right] v &= -\nabla\mu + \xi, && \text{in } \Omega, \\ \mu &= -\xi_\partial, && \text{on } \Gamma^D, \\ v \cdot n &= 0, && \text{on } \Gamma^N. \end{aligned}$$

2. *Compute $\{\partial_\lambda H\}(0) = -\int_\Omega \{\partial_\lambda \kappa\} u_0 v dx$.*

Approximate $E_{H_i} \approx \{\partial_\lambda H\}(0)$.

used. The FS trapezoidal rule, AS trapezoidal rule, FS Gaussian quadrature, and AS Gaussian quadrature methods are outlined in Algorithms 6.4-6.7, respectively.

An important comparison is the computational complexity where the FS methods are superior. The FS methods allow the reuse of the computed sensitivities from the Quasi-Newton method and so fewer system solutions are needed for a given number of integration points. Algorithms 6.4-6.7 are implemented in Listing 14 and are applied to the example problem from Section 5.3.3. The estimated error in the x-velocity is shown in Figure 6.2.1 and the estimated error in the y-velocity is shown in Figure 6.2.2. Again, the FS methods perform better than the AS methods. Surprisingly, the Gaussian quadrature methods do not perform as well as the trapezoidal rule.

6.3 Summary

In this chapter a method for approximating the modeling error was proposed. The method uses the techniques developed in Chapter 3 for sensitivity analysis and the homotopy from Chapter 5. Having a good approximation for the error in a quantity of interest may aid in choosing a model. If the error is known to be small, then a simpler model may be used.

In Section 6.1, the general framework was given. The Fundamental Theorem of Calculus was employed to give a simple relationship between the error in the quantity of interest E_G and the integral of the sensitivity of the objective function with respect to the continuation parameter. The sensitivity of the objective

Algorithm 6.4. *Given $N \geq 1$, let $\Delta\lambda = \frac{1}{N}$. Let p_0, u_0 solve the linear flow system (5.3.9)-(5.3.12). For $i = 1, \ldots, N$:*

1. *Let $\lambda_i = i\Delta\lambda$, $\kappa_i = \left[\kappa_L + \lambda_i \kappa_{NL}\left(|u_{i-1}|\right) + \lambda_i \partial_{|u|}\kappa_{NL}\left(\partial_u |u|\left(u_{i-1}\right)\right)\right]$, and $g_i = -\kappa_{NL}\left(|u_{i-1}|\right) u_{i-1}$ and solve for $\{\partial_\lambda p\}_i, \{\partial_\lambda u\}_i$*

$$\begin{aligned} \nabla \cdot \{\partial_\lambda u\}_i &= 0, && \text{in } \Omega, \\ \kappa_i \{\partial_\lambda u\}_i &= -\nabla \{\partial_\lambda p\}_i + g_i, && \text{in } \Omega, \\ \{\partial_\lambda p\}_i &= 0, && \text{on } \Gamma^D, \\ \{\partial_\lambda u\}_i \cdot n &= 0, && \text{on } \Gamma^N. \end{aligned}$$

2. *Let $p_i^* = p_{i-1} + \Delta\lambda \{\partial_\lambda p\}_i$ and $u_i^* = u_{i-1} + \Delta\lambda \{\partial_\lambda u\}_i$.*

3. *Solve for u_i and p_i the linear system*

$$\begin{aligned} \nabla \cdot u_i &= f(x), && \text{in } \Omega, \\ \kappa_L u_i + \lambda_i \kappa_{NL}\left(|u_i^*|\right) u_i &= -\nabla p_i + g(x), && \text{in } \Omega, \\ p_i &= p_D(x), && \text{on } \Gamma^D, \\ u_i \cdot n &= u_N(x), && \text{on } \Gamma^N. \end{aligned}$$

4. *Compute $\{\partial_\lambda H\}(\lambda_i) = -\int_\Omega \xi(x) \{\partial_\lambda u\}_i \, dx$.*

Approximate $\int_0^1 \{\partial_\lambda H\} \, d\lambda = -\int_0^1 \{\partial_\lambda H\} \, d\lambda$ using the trapezoidal rule:

$$\frac{\Delta\lambda}{2} \sum_{i=1}^{N} \left(\{\partial_\lambda H\}(\lambda_i) + \{\partial_\lambda H\}(\lambda_{i-1})\right).$$

Algorithm 6.5. *Given $N \geq 1$, let $\Delta\lambda = \frac{1}{N}$. Let p_0, u_0 solve the linear flow system (5.3.9)-(5.3.12). For $i = 1, \ldots, N$:*

1. *Let $\lambda_i = i\Delta\lambda$, $\kappa_i = \left[\kappa_L + \lambda_i \kappa_{NL}\left(|u_{i-1}|\right) + \lambda_i \partial_{|u|}\kappa_{NL}\left(\partial_u |u|\left(u_{i-1}\right)\right)\right]$, and $g_i = -\kappa_{NL}\left(|u_{i-1}|\right) u_{i-1}$ and solve for $\{\partial_\lambda p\}_i, \{\partial_\lambda u\}_i$*

$$\begin{aligned} \nabla \cdot \{\partial_\lambda u\}_i &= 0, && \text{in } \Omega, \\ \kappa_i \{\partial_\lambda u\}_i &= -\nabla \{\partial_\lambda p\}_i + g_i, && \text{in } \Omega, \\ \{\partial_\lambda p\}_i &= 0, && \text{on } \Gamma^D, \\ \{\partial_\lambda u\}_i \cdot n &= 0, && \text{on } \Gamma^N. \end{aligned}$$

2. *Let $p_i^* = p_{i-1} + \Delta\lambda \{\partial_\lambda p\}_i$ and $u_i^* = u_{i-1} + \Delta\lambda \{\partial_\lambda u\}_i$.*

3. *Solve for u_i and p_i the linear system*

$$\begin{aligned} \nabla \cdot u_i &= f(x), && \text{in } \Omega, \\ \kappa_L u_i + \lambda_i \kappa_{NL}\left(|u_i^*|\right) u_i &= -\nabla p_i + g(x), && \text{in } \Omega, \\ p_i &= p_D(x), && \text{on } \Gamma^D, \\ u_i \cdot n &= u_N(x), && \text{on } \Gamma^N. \end{aligned}$$

4. *Solve for μ_i and v_i the linear system*

$$\begin{aligned} \nabla \cdot v_i &= 0, && \text{in } \Omega, \\ \left[\partial_{|u|}\kappa\left(\partial_u |u|\left(u_i\right)\right) + \kappa\left(\lambda_i; |u_i|\right)\right] v_i &= -\nabla \mu_i + \xi, && \text{in } \Omega, \\ \mu_i &= -\xi_\partial, && \text{on } \Gamma^D, \\ v_i \cdot n &= 0, && \text{on } \Gamma^N. \end{aligned}$$

5. *Compute $\{\partial_\lambda H\}(\lambda_i) = -\int_\Omega \{\partial_\lambda \kappa\} u_i v_i dx$.*

Approximate $\int_0^1 \{\partial_\lambda H\} d\lambda = -\int_0^1 \{\partial_\lambda H\} d\lambda$ using the trapezoidal rule:

$$\frac{\Delta\lambda}{2} \sum_{i=1}^{N} \left(\{\partial_\lambda H\}(\lambda_i) + \{\partial_\lambda H\}(\lambda_{i-1})\right).$$

Algorithm 6.6. *Given a choice of orthogonal polynomials and $N \geq 1$, let x_i be the N nodes in $[-1,1]$ and w_i be the corresponding weights. Let $\lambda_0 = 0$ and let p_0, u_0 solve the linear flow system (5.3.9)-(5.3.12). For $i = 1, \ldots, N$:*

1. *Let $\lambda_i = \frac{x_i+1}{2}$.*
2. *Let $\kappa_i = \left[\kappa_L + \lambda_i \kappa_{NL}\left(|u_{i-1}|\right) + \lambda_i \partial_{|u|}\kappa_{NL}\left(\partial_u |u|\left(u_{i-1}\right)\right)\right]$, and $g_i = -\kappa_{NL}\left(|u_{i-1}|\right) u_{i-1}$ and solve for $\{\partial_\lambda p\}_i, \{\partial_\lambda u\}_i$*

$$\begin{aligned} \nabla \cdot \{\partial_\lambda u\}_i &= 0, && \text{in } \Omega, \\ \kappa_i \{\partial_\lambda u\}_i &= -\nabla \{\partial_\lambda p\}_i + g_i, && \text{in } \Omega, \\ \{\partial_\lambda p\}_i &= 0, && \text{on } \Gamma^D, \\ \{\partial_\lambda u\}_i \cdot n &= 0, && \text{on } \Gamma^N. \end{aligned}$$

3. *Let $\Delta\lambda = \lambda_i - \lambda_{i-1}$ and let $p_i^* = p_{i-1} + \Delta\lambda \{\partial_\lambda p\}_i$ and $u_i^* = u_{i-1} + \Delta\lambda \{\partial_\lambda u\}_i$.*
4. *Solve for u_i and p_i the linear system*

$$\begin{aligned} \nabla \cdot u_i &= f(x), && \text{in } \Omega, \\ \kappa_L u_i + \lambda_i \kappa_{NL}\left(|u_i^*|\right) u_i &= -\nabla p_i + g(x), && \text{in } \Omega, \\ p_i &= p_D(x), && \text{on } \Gamma^D, \\ u_i \cdot n &= u_N(x), && \text{on } \Gamma^N. \end{aligned}$$

Approximate $\int_0^1 \{\partial_\lambda H\} d\lambda = -\int_0^1 \int_\Omega \xi(x) \{\partial_\lambda u\}_i dx d\lambda$ using Gaussian quadrature:

$$\sum_{i=1}^{N} w_i \{\partial_\lambda \kappa\} u_i v_i.$$

Algorithm 6.7. *Given a choice of orthogonal polynomials and $N \geq 1$, let x_i be the N nodes in $[-1,1]$ and w_i be the corresponding weights. Let $\lambda_0 = 0$ and let p_0, u_0 solve the linear flow system (5.3.9)-(5.3.12). For $i = 1, \ldots, N$:*

1. *Let $\lambda_i = \frac{x_i+1}{2}$.*
2. *Let $\kappa_i = \left[\kappa_L + \lambda_i \kappa_{NL}\left(|u_{i-1}|\right) + \lambda_i \partial_{|u|}\kappa_{NL}\left(\partial_u |u|\left(u_{i-1}\right)\right)\right]$, and $g_i = -\kappa_{NL}\left(|u_{i-1}|\right) u_{i-1}$ and solve for $\left\{\partial_\lambda p\right\}_i, \left\{\partial_\lambda u\right\}_i$*

$$\begin{aligned} \nabla \cdot \left\{\partial_\lambda u\right\}_i &= 0, && \text{in } \Omega, \\ \kappa_i \left\{\partial_\lambda u\right\}_i &= -\nabla \left\{\partial_\lambda p\right\}_i + g_i, && \text{in } \Omega, \\ \left\{\partial_\lambda p\right\}_i &= 0, && \text{on } \Gamma^D, \\ \left\{\partial_\lambda u\right\}_i \cdot n &= 0, && \text{on } \Gamma^N. \end{aligned}$$

3. *Let $\Delta\lambda = \lambda_i - \lambda_{i-1}$ and let $p_i^* = p_{i-1} + \Delta\lambda \left\{\partial_\lambda p\right\}_i$ and $u_i^* = u_{i-1} + \Delta\lambda \left\{\partial_\lambda u\right\}_i$.*
4. *Solve for u_i and p_i the linear system*

$$\begin{aligned} \nabla \cdot u_i &= f\left(x\right), && \text{in } \Omega, \\ \kappa_L u_i + \lambda_i \kappa_{NL}\left(|u_i^*|\right) u_i &= -\nabla p_i + g\left(x\right), && \text{in } \Omega, \\ p_i &= p_D\left(x\right), && \text{on } \Gamma^D, \\ u_i \cdot n &= u_N\left(x\right), && \text{on } \Gamma^N. \end{aligned}$$

5. *Solve for μ_i and v_i the linear system*

$$\begin{aligned} \nabla \cdot v_i &= 0, && \text{in } \Omega, \\ \left[\partial_{|u|}\kappa\left(\partial_u |u|\left(u_i\right)\right) + \kappa\left(\lambda_i; |u_i|\right)\right] v_i &= -\nabla \mu_i + \xi, && \text{in } \Omega, \\ \mu_i &= -\xi_\partial, && \text{on } \Gamma^D, \\ v_i \cdot n &= 0, && \text{on } \Gamma^N. \end{aligned}$$

Approximate $\int_0^1 \left\{\partial_\lambda H\right\} d\lambda = -\int_0^1 \int_\Omega \left\{\partial_\lambda \kappa\right\} uv dx d\lambda$ using Gaussian quadrature:

$$\sum_{i=1}^{N} w_i \left\{\partial_\lambda \kappa\right\} u_i v_i.$$

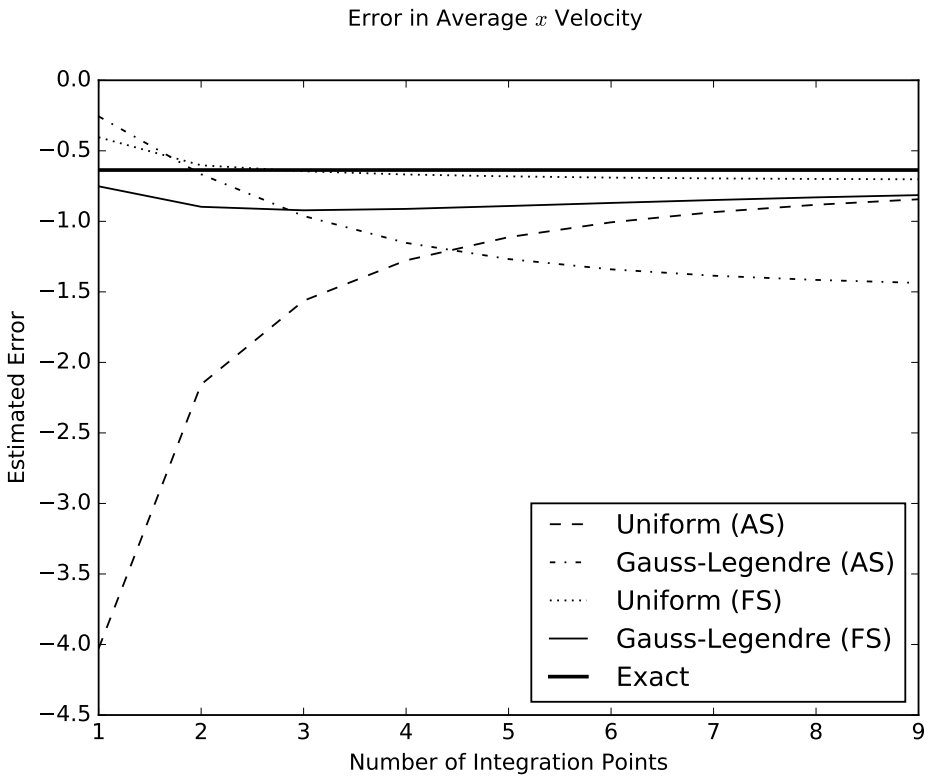

Figure 6.2.1: Estimated Error in Average Velocity in x-direction

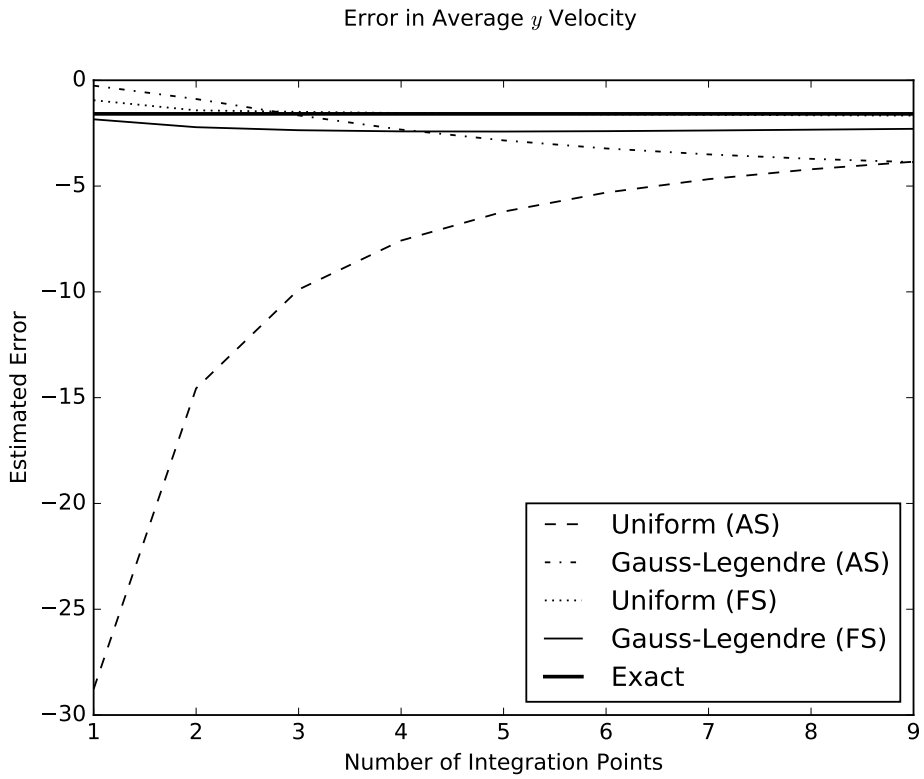

Figure 6.2.2: Estimated Error in Average Velocity in y-direction

function may be computed using either FS or AS from Chapter 3. The integration with respect to the continuation parameter may be done using techniques such as the left hand rule, the trapezoidal rule, or Gaussian quadrature.

In Section 6.2, the methods developed in Section 6.1 were applied to the nonlinear flow model from Chapter 2. Algorithms were developed using FS and AS from Chapter 3 and integration techniques such as the left hand rule, the trapezoidal rule, and Gaussian quadrature. When applied to an example problem the FS methods emerged as the superior methods since the error in the quantity of interest was more accurately predicted and the computational cost was less. Surprisingly, the trapezoidal rule performed better than Gaussian quadrature for the integration with respect to the continuation parameter.

Besides the advantages of lower computational complexity and better accuracy, which are enough to warrant its use, the FS methods have an additional advantage since they are inherently local in nature. If the modeling error were used to choose the model locally, the local error could be computed using FS with only the cost of additional integration whereas the AS method would require the solutions to additional systems. This is a demonstration of the general principle that the FS method is preferred when there are many quantities of interest and few parameters since there are potentially many quantities of interest and only one parameter.

Chapter 7 Conclusions and Future Work

In this work several new ideas were applied to a problem of coupled fluid flow and transport in porous media. Without access to closed form solutions, it is important to have computational algorithms to approximate the solutions. This work explored several methods related to approximating those solutions.

Before implementing a numerical method to approximate the solution, it is important to verify that the problem is, indeed, well-posed. In this work, the nonlinear non-Darcy model was shown to be well-posed. It is often much more difficult to show that a nonlinear problem is well-posed than a linear model. The proof of the well-posedness of the non-Darcy model relied on recently published analysis results. To our knowledge, this is the first proof of well-posedness in this setting.

With the well-posedness of the model problems established, a numerical algorithm was developed to approximate the solution to the problem. Cell-centered finite difference methods were connected to an expanded mixed finite element method for the linear Darcy model. Cell-centered finite difference methods offer the advantage that they are relatively easy to implement, but they impose more severe restrictions on the type of data that is allowed. Meanwhile, finite element methods are less restrictive for the type of data that are allowed. The mixed finite element method provides conservative approximations to the velocity

which are desirable. The expanded mixed finite element method allowed for the media to be anisotropic. It was proposed that Newton's method could be applied to use the Darcy solver to get the solution to the nonlinear non-Darcy system.

Next, the theoretical framework for sensitivity analysis was established. In this work, sensitivity analysis was discussed as finding the derivative of the solution to a system with respect to model parameters. There are various applications beyond the scope of this work for sensitivity analysis including model reduction, statistical analysis, and gaining insight into how accurately model parameters must be estimated. The problem of finding sensitivities can be quite delicate since theory usually does not guarantee the existence of such derivatives. In this work, forward and adjoint methods for finding sensitivities were developed. In particular, we believe this is the first example of sensitivity analysis for a coupled system. With the theoretical framework in place, several examples demonstrating techniques of sensitivity analysis were shown.

In Chapter 5, a set of novel numerical continuation algorithms were developed. Numerical continuation methods trace a path between an easy to solve problem and a more difficult problem. The methods developed in this work combine the idea of tracing a path with sensitivity analysis. To do this, an initial value problem was set up with initial value at a linear problem and, using sensitivity analysis, the derivative with respect to a continuation parameter was found. Several methods for solving the initial value problem were proposed. In the end, a predictor-corrector algorithm which we called the Quasi-Newton method was shown to work quite well.

Finally, a method for model adaptivity was developed. Model adaptivity is concerned with approximating the modeling error so that an appropriate model may be chosen. The continuation methods developed in Chapter 5 were leveraged to give an estimate of the modeling error from choosing the Darcy model over the non-Darcy model. It was demonstrated that the Quasi-Newton continuation method may be used to estimate the error in a quantity of interest.

There are many directions for future work to proceed. The simplest would be to apply the results to slightly more complicated systems than the problems in this work. For instance, the examples in this work did not have anisotropy in the medium. Since the Darcy solver has a larger stencil in the presence of anisotropy, the continuation method may show even more use in that setting. Also, the examples in Chapter 4, even though implemented in 2D, were essentially equivalent to one dimensional problems. Demonstrating that the techniques work on bona fide two dimensional problems would be an easy extension of the present work.

Another open question is what other related fluid flow models may be shown to be well-posed. It was pointed out that the simple addition of anisotropy in the nonlinear term may pose a problem for the present theory and it was shown that the form for the magnitude of velocity also deserves delicate treatment. These and other related models may be well-posed, but the present theory does not show that they are.

Some extensions to the continuation method may also be possible. One possible extension is to models with more than one part. An example is a system of the

form

$$K^{-1}u + \beta |u| u + \gamma |u|^{\alpha} u = -\nabla p.$$

In such a case, the continuation and modeling error may be applied by adding two continuation parameters, $\lambda_1, \lambda_2 \in [0, 1]$, so that the equation reads

$$K^{-1}u + \lambda_1 \beta |u| u + \lambda_2 \gamma |u|^{\alpha} u = -\nabla p.$$

For the modeling error, the extension is immediate, but for numerical continuation the extension is less obvious. An open question is how to follow a path from $\lambda_1 = \lambda_2 = 0$ to $\lambda_1 = \lambda_2 = 1$. There are many possible paths and defining and tracing an optimal path is an open question.

Another extension is to explore the composition of models further. For instance, suppose that a coupled system of flow, transport, and reaction is to be solved which may be written as

$$R(T(F)).$$

It is possible that each system may be modeled in several ways. For instance, the flow model may be solved using the Darcy flow model or the non-Darcy flow model, the transport may include or exclude diffusion, and the reaction system may be simple if only a few reactive species are present while it may be more complicated if there are many reactive species present. In such a system, it may be advantageous to estimate the modeling error using several continuation parameters at each level.

The system might look something like

$$\sum_{i=1}^{N_R} \lambda_i^R R_i \left(\sum_{j=1}^{N_T} \lambda_j^T T_j \left(\sum_{k=1}^{N_F} \lambda_k^F F_k \right) \right),$$

where N_R is the number of reaction systems, N_T is the number of transport systems, and N_F is the number of flow systems.

Finally, rather than a continuation method that works in a global way, it may be possible to develop a continuation method that works more locally. The idea would be to progress the continuation more quickly in places where the sensitivity to the continuation parameter is low and more slowly where the sensitivity to the continuation parameter is high.

Bibliography

[1] Eugene L. Allgower and Kurt Georg. *Introduction to numerical continuation methods*, volume 45 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2003. Reprint of the 1990 edition [Springer-Verlag, Berlin; MR1059455 (92a:65165)].

[2] Todd Arbogast, Mary F. Wheeler, and Ivan Yotov. Mixed finite elements for elliptic problems with tensor coefficients as cell-centered finite differences. *SIAM J. Numer. Anal.*, 34(2):828–852, 1997.

[3] Kendall Atkinson and Weimin Han. *Theoretical numerical analysis*, volume 39 of *Texts in Applied Mathematics*. Springer, New York, second edition, 2005. A functional analysis framework.

[4] Eugenio Aulisa, Lidia Bloshanskaya, Luan Hoang, and Akif Ibragimov. Analysis of generalized Forchheimer flows of compressible fluids in porous media. *J. Math. Phys.*, 50(10):103102, 44, 2009.

[5] A. H. Baker, E. R. Jessup, and T. Manteuffel. A technique for accelerating the convergence of restarted GMRES. *SIAM J. Matrix Anal. Appl.*, 26(4):962–984, 2005.

[6] Jacob Bear. *Dynamics of fluids in porous media*. Courier Dover Publications, 1972.

[7] Roland Becker and Rolf Rannacher. An optimal control approach to a posteriori error estimation in finite element methods. *Acta Numer.*, 10:1–102, 2001.

[8] Roland Becker and Boris Vexler. Mesh refinement and numerical sensitivity analysis for parameter calibration of partial differential equations. *J. Comput. Phys.*, 206(1):95–110, 2005.

[9] Dietrich Braess. *Finite elements*. Cambridge University Press, Cambridge, third edition, 2007. Theory, fast solvers, and applications in elasticity theory, Translated from the German by Larry L. Schumaker.

[10] Franco Brezzi and Michel Fortin. *Mixed and hybrid finite element methods*, volume 15 of *Springer Series in Computational Mathematics*. Springer-Verlag, New York, 1991.

[11] Peter N. Brown, George D. Byrne, and Alan C. Hindmarsh. VODE: a variable-coefficient ODE solver. *SIAM J. Sci. Statist. Comput.*, 10(5):1038–1051, 1989.

[12] Richard L Burden and J. Douglas Faires. *Numerical Analysis*. Brooks/Cole, Cengage Learning, Boston, MA, Boston, MA, 9th edition, 2011.

[13] G. D. Byrne and A. C. Hindmarsh. A polyalgorithm for the numerical solution of ordinary differential equations. *ACM Trans. Math. Software*, 1(1):71–96, 1975.

[14] Rong Cai, W Brent Lindquist, Wooyong Um, and Keith W Jones. Tomographic analysis of reactive flow induced pore structure changes in column experiments. *Advances in water resources*, 32(9):1396–1403, 2009.

[15] Yang Cao, Shengtai Li, and Linda Petzold. Adjoint sensitivity analysis for differential-algebraic equations: algorithms and software. *J. Comput. Appl. Math.*, 149(1):171–191, 2002. Scientific and engineering computations for the 21st century—methodologies and applications (Shizuoka, 2001).

[16] Yang Cao, Shengtai Li, Linda Petzold, and Radu Serban. Adjoint sensitivity analysis for differential-algebraic equations: the adjoint DAE system and its numerical solution. *SIAM J. Sci. Comput.*, 24(3):1076–1089 (electronic), 2002.

[17] N. L. Carothers. *Real analysis*. Cambridge University Press, Cambridge, 2000.

[18] Karen Chan, Stefano Tarantola, Andrea Saltelli, and Ilya M. Sobol′. Variance-based methods. In *Sensitivity analysis*, Wiley Ser. Probab. Stat., pages 167–197. Wiley, Chichester, 2000.

[19] J. Crank. *The mathematics of diffusion*. Clarendon Press, Oxford, second edition, 1975.

[20] Veronica Czitrom. One-factor-at-a-time versus designed experiments. *The American Statistician*, 53(2):126–131, 1999.

[21] Peter Deuflhard. *Newton methods for nonlinear problems*, volume 35 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 2004. Affine invariance and adaptive algorithms.

[22] J. Dieudonné. *Foundations of modern analysis*. Academic Press, New York, 1969. Enlarged and corrected printing, Pure and Applied Mathematics, Vol. 10-I.

[23] Jim Douglas Jr, Paulo Jorge Paes-Leme, and Tiziana Giorgi. Generalized forchheimer flow in porous media. *Boundary value problems for partial differential equations and applications: dedicated to E. Magenes*, 29:99, 1993.

[24] LJ Durfolsky. Numerical calculation of equivalent grid block permeability tensors of heterogeneous porous media: Water resour res v27, n5, may 1991, p299–708. In *International Journal of Rock Mechanics and Mining Sciences & Geomechanics Abstracts*, volume 28, page A350. Pergamon, 1991.

[25] Pierre Fabrie and Michel Langlais. Mathematical analysis of miscible displacement in porous medium. *SIAM J. Math. Anal.*, 23(6):1375–1392, 1992.

[26] Philipp Forchheimer. Wasserbewegung durch boden. *Z. Ver. Deutsch. Ing*, 45(1782):1788, 1901.

[27] C. R. Garibotti and M. Peszyńska. Upscaling non-Darcy flow. *Transp. Porous Media*, 80(3):401–430, 2009.

[28] Andreas Griewank and Andrea Walther. *Evaluating derivatives*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2008. Principles and techniques of algorithmic differentiation.

[29] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving ordinary differential equations. I*, volume 8 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, second edition, 1993. Nonstiff problems.

[30] Alexander Hay, Imran Akhtar, and Jeff T. Borggaard. On the use of sensitivity analysis in model reduction to predict flows for varying inflow conditions. *Internat. J. Numer. Methods Fluids*, 68(1):122–134, 2012.

[31] Chuanping He, Michael G. Edwards, and Louis J. Durlofsky. Numerical calculation of equivalent cell permeability tensors for general quadrilateral control volumes. *Comput. Geosci.*, 6(1):29–47, 2002.

[32] Alan C. Hindmarsh. ODEPACK, a systematized collection of ODE solvers. In *Scientific computing (Montreal, Que., 1982)*, IMACS Trans. Sci. Comput., I, pages 55–64. IMACS, New Brunswick, NJ, 1983.

[33] K. R. Jackson and R. Sacks-Davis. An alternative implementation of variable step-size multistep formulas for stiff ODEs. *ACM Trans. Math. Software*, 6(3):295–318, 1980.

[34] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2017-03-17].

[35] C. T. Kelley. *Iterative methods for linear and nonlinear equations*, volume 16 of *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1995. With separately available software.

[36] D Kim, CA Peters, and WB Lindquist. Upscaling geochemical reaction rates accompanying acidic co2-saturated brine flow in sandstone aquifers. *Water Resources Research*, 47(1), 2011.

[37] D. A. Knoll and D. E. Keyes. Jacobian-free Newton-Krylov methods: a survey of approaches and applications. *J. Comput. Phys.*, 193(2):357–397, 2004.

[38] O. A. Ladyženskaja, V. A. Solonnikov, and N. N. Ural′ceva. *Linear and quasilinear equations of parabolic type*. Translated from the Russian by S. Smith. Translations of Mathematical Monographs, Vol. 23. American Mathematical Society, Providence, R.I., 1968.

[39] Randall J. LeVeque. *Finite volume methods for hyperbolic problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, 2002.

[40] Randall J. LeVeque. *Finite difference methods for ordinary and partial differential equations*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2007. Steady-state and time-dependent problems.

[41] Shengtai Li and Linda Petzold. Software and algorithms for sensitivity analysis of large-scale differential algebraic systems. *J. Comput. Appl. Math.*, 125(1-2):131–145, 2000. Numerical analysis 2000, Vol. VI, Ordinary differential equations and integral equations.

[42] Shengtai Li and Linda Petzold. Adjoint sensitivity analysis for time-dependent partial differential equations with adaptive mesh refinement. *J. Comput. Phys.*, 198(1):310–325, 2004.

[43] Shengtai Li and Linda Petzold. Adjoint sensitivity analysis for time-dependent partial differential equations with adaptive mesh refinement. *J. Comput. Phys.*, 198(1):310–325, 2004.

[44] Shengtai Li, Linda Petzold, and Wenjie Zhu. Sensitivity analysis of differential-algebraic equations: a comparison of methods on a special problem. *Appl. Numer. Math.*, 32(2):161–174, 2000.

[45] Shengtai Li, Linda Petzold, and Wenjie Zhu. Sensitivity analysis of differential-algebraic equations: a comparison of methods on a special problem. *Appl. Numer. Math.*, 32(2):161–174, 2000.

[46] Shengtai Li, Linda R. Petzold, and James M. Hyman. Solution adapted mesh refinement and sensitivity analysis for parabolic partial differential equation systems. In *Large-scale PDE-constrained optimization (Santa Fe, NM, 2001)*, volume 30 of *Lect. Notes Comput. Sci. Eng.*, pages 117–132. Springer, Berlin, 2003.

[47] Timothy Maly and Linda R. Petzold. Numerical methods and software for sensitivity analysis of differential-algebraic systems. *Appl. Numer. Math.*, 20(1-2):57–79, 1996. Workshop on the method of lines for time-dependent problems (Lexington, KY, 1995).

[48] John N. McDonald and Neil A. Weiss. *A course in real analysis.* Academic Press Inc., San Diego, CA, 1999. Biographies by Carol A. Weiss.

[49] A. R. Meenakshi and C. Rajian. On a product of positive semidefinite matrices. *Linear Algebra Appl.*, 295(1-3):3–6, 1999.

[50] J.R. Munkres. *Topology.* Featured Titles for Topology Series. Prentice Hall, Incorporated, 2000.

[51] James M Murphy, David MH Sexton, David N Barnett, Gareth S Jones, Mark J Webb, Matthew Collins, and David A Stainforth. Quantification of modelling uncertainties in a large ensemble of climate change simulations. *Nature*, 430(7001):768–772, 2004.

[52] J. T. Oden and S. Prudhomme. Goal-oriented error estimation and adaptivity for the finite element method. *Comput. Math. Appl.*, 41(5-6):735–756, 2001.

[53] J. Tinsley Oden, Ivo Babuška, Fabio Nobile, Yusheng Feng, and Raul Tempone. Theory and methodology for estimation and control of errors due to modeling, approximation, and uncertainty. *Comput. Methods Appl. Mech. Engrg.*, 194(2-5):195–204, 2005.

[54] J. Tinsley Oden and Serge Prudhomme. Estimation of modeling error in computational mechanics. *J. Comput. Phys.*, 182(2):496–515, 2002.

[55] J. Tinsley Oden, Serge Prudhomme, Daniel C. Hammerand, and Mieczyslaw S. Kuczma. Modeling error and adaptivity in nonlinear continuum mechanics. *Comput. Methods Appl. Mech. Engrg.*, 190(49-50):6663–6684, 2001.

[56] J. Tinsley Oden, Serge Prudhomme, Albert Romkes, and Paul T. Bauman. Multiscale modeling of physical phenomena: adaptive control of models. *SIAM J. Sci. Comput.*, 28(6):2359–2389, 2006.

[57] M Peszyńska, Anna Trykozko, and Ken Kennedy. Sensitivity to parameters in non-darcy flow model from porescale through mesoscale. In J. Carrera, editor, *Proceedings of XVIII International Conference on Water Resources*, 2010.

[58] Malgorzata Peszynska and Anna Trykozko. Pore-to-core simulations of flow with large velocities using continuum models and imaging data. *Computational Geosciences*, 17(4):623–645, 2013.

[59] Małgorzata Peszyńska, Anna Trykozko, and Kyle Augustson. *Computational Upscaling of Inertia Effects from Porescale to Mesoscale*, pages 695–704. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[60] Catherine A Peters. Accessibilities of reactive minerals in consolidated sedimentary rock: An imaging study of three sandstones. *Chemical Geology*, 265(1):198–208, 2009.

[61] Linda Petzold, Shengtai Li, Yang Cao, and Radu Serban. Sensitivity analysis of differential-algebraic equations and partial differential equations. *Computers & chemical engineering*, 30(10):1553–1559, 2006.

[62] Thomas F Russell and Mary Fanett Wheeler. Finite element and finite difference methods for continuous flows in porous media. *The mathematics of reservoir simulation*, 1:35–106, 1983.

[63] Andrea Saltelli and Paola Annoni. How to avoid a perfunctory sensitivity analysis. *Environmental Modelling & Software*, 25(12):1508–1517, 2010.

[64] Andrea Saltelli and Stefano Tarantola. On the relative importance of input factors in mathematical models: safety assessment for nuclear waste disposal. *J. Amer. Statist. Assoc.*, 97(459):702–709, 2002.

[65] R. E. Showalter. *Monotone operators in Banach space and nonlinear partial differential equations*, volume 49 of *Mathematical Surveys and Monographs*. American Mathematical Society, Providence, RI, 1997.

[66] R. E. Showalter. Nonlinear degenerate evolution equations in mixed formulation. *SIAM J. Math. Anal.*, 42(5):2114–2131, 2010.

[67] John R. Singler. Differentiability with respect to parameters of weak solutions of linear parabolic equations. *Math. Comput. Modelling*, 47(3-4):422–430, 2008.

[68] Lisa G. Stanley and Dawn L. Stewart. *Design sensitivity analysis*, volume 25 of *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2002. Computational issues of sensitivity equation methods.

[69] Walter A. Strauss. *Partial differential equations*. John Wiley & Sons Ltd., Chichester, second edition, 2008. An introduction.

[70] Raimond A Struble. *Nonlinear differential equations*, volume 267. McGraw-Hill New York, 1962.

[71] Anna-Karin Tornberg and Björn Engquist. Numerical approximations of singular source terms in differential equations. *J. Comput. Phys.*, 200(2):462–488, 2004.

APPENDIX

The following source code demonstrates the methods used in this thesis.

Listing 1: flowSolver.py

```
""" A flow solver based on [Arbogast, Wheeler]. """

import numpy as np
import numexpr as ne
from numpy import zeros,ones,outer
from numpy.linalg import inv
from scipy.linalg import solve
from scipy.optimize import newton_krylov
from scipy.interpolate import interp1d
from scipy.sparse.linalg import LinearOperator,lgmres

class DarcySolver(object):
    def __init__(self,x,y,Boundary,K=None,f=None,g=None,
                 solverOptions=None,NDSolverOptions=None):
        # Set up the class variables.
        self.x,self.y = x,y
        self.Nx,self.Ny = len(x)-1,len(y)-1
        self.X = (x[1:]+x[:-1])/2.
        self.Y = (y[1:]+y[:-1])/2.
        # Dx,Dy are the cell dimensions.
        self.Dx,self.Dy = x[1:] - x[:-1],y[1:]-y[:-1]
        # dx,dy are the distances between cell centers.
        self.dx = (self.Dx[1:]+self.Dx[:-1])/2.
        self.dy = (self.Dy[1:]+self.Dy[:-1])/2.
        # K is the inverse of the resistance tensor.
        if K is not None:
            self.K = K
        self.F = f
        self.g = g
        self.Boundary = Boundary
        self._Kgx = zeros((self.Nx+1,self.Ny))
        self._Kgy = zeros((self.Nx,self.Ny+1))
        if solverOptions is not None:
            self.solverOptions = solverOptions
        else:
            self.solverOptions = {'tol':1e-10}
        if NDSolverOptions is not None:
            self.NDSolverOptions = NDSolverOptions
        else:
            self.NDSolverOptions = {'f_tol':1e-10}

    def get_F(self):
        return self._F

    def set_F(self,f):
        self._F = f
        if self._F is not None:
            self.Fdxdy = self._F*outer(self.Dx,self.Dy)
        else:
            self.Fdxdy = zeros((self.Nx,self.Ny))

    F = property(get_F,set_F)
```

```
def get_K(self):
    return self._K

def set_K(self,K):
    self._K = K

K = property(get_K,set_K)

def get_kappa(self):
    raise ValueError("Only K is stored.")

def set_kappa(self,kappa):
    Nx,Ny = self.Nx,self.Ny
    K = {'11':zeros((Nx+1,Ny+1)),'12':zeros((Nx+1,Ny+1)),
         '22':zeros((Nx+1,Ny+1))}
    Kij = zeros((2,2))
    for i in xrange(Nx+1):
        for j in xrange(Ny+1):
            Kij[0,0] = kappa['11'][i,j]
            Kij[1,0] = kappa['12'][i,j]
            Kij[0,1] = kappa['12'][i,j]
            Kij[1,1] = kappa['22'][i,j]
            Kinv = inv(Kij)
            K['11'][i,j] = Kinv[0,0]
            K['12'][i,j] = Kinv[1,0]
            K['22'][i,j] = Kinv[1,1]
    self.K = K

kappa = property(get_kappa,set_kappa)

def getGravitySourceEdges(self):
    Nx,Ny = self.Nx,self.Ny
    Kgx,Kgy = self._Kgx,self._Kgy
    if self.g is not None:
        Dx,Dy,dx,dy = self.Dx,self.Dy,self.dx,self.dy
        K11,K22,K12 = self.K['11'],self.K['22'],self.K['12']
        gx,gy,gyx,gxy = self.g['1x'],self.g['2y'],self.g['2x'],self.g['1y']
        RHS = zeros((Nx,Ny))
        # Set up RHS from gravity.
        Kgx = (gx*(K11[:,:-1]+K11[:,1:])/2.
               + gyx*(K12[:,:-1]+K12[:,1:])/2.)
        Kgy = (gy*(K22[:-1,:]+K22[1:,:])/2.
               + gxy*(K12[:-1,:]+K12[1:,:])/2.)
    return Kgx,Kgy

def getGravitySource(self):
    Nx,Ny = self.Nx,self.Ny
    Kgx,Kgy = self.getGravitySourceEdges()
    return self.Divergence(Kgx,Kgy)

def getWellSource(self):
    return self.Fdxdy

def SetUpRHS(self):
    Kgdiv = self.getGravitySource()
    Fdxdy = self.getWellSource()
```

```
        return Kgdiv - Fdxdy

    def GetVelocity(self,p):
        N,D = self.Boundary['Neumann'],self.Boundary['Dirichlet']
        Bv = self.Boundary['Values']
        if self.g is not None:
            # Approximate velocity: u = K*g - K*grad(p)
            Kgx,Kgy = self.getGravitySourceEdges()
            kpx,kpy = self.KGrad(p)
            ux,uy = Kgx-kpx,Kgy-kpy
        else:
            kpx,kpy = self.KGrad(p)
            ux,uy = -kpx,-kpy
        ux[0,N['West']] = Bv['West'][N['West']]
        ux[-1,N['East']] = Bv['East'][N['East']]
        uy[N['South'],0] = Bv['South'][N['South']]
        uy[N['North'],-1] = Bv['North'][N['North']]
        return ux,uy

    def GetVelocityNonDarcy(self,p):
        Nx,Ny = self.Nx,self.Ny
        self._p = p
        ux,uy = self.GetVelocity(p)
        u = np.append(ux.flat,uy.flat)
        u = newton_krylov(self.velocityResidual,u,
                          f_tol=self.NDSolverOptions['f_tol']/2.)
        ux = u[:(Nx+1)*Ny].reshape((Nx+1,Ny))
        uy = u[(Nx+1)*Ny:].reshape((Nx,Ny+1))
        return ux,uy

    def velocityResidual(self,u):
        Nx,Ny = self.Nx,self.Ny
        ux = u[:(Nx+1)*Ny].reshape((Nx+1,Ny))
        uy = u[(Nx+1)*Ny:].reshape((Nx,Ny+1))
        self.updateKappa(ux,uy)
        vx,vy = self.GetVelocity(self._p)
        return np.append((vx-ux).flat,(vy-uy).flat)

    def KGrad(self,p):
        Nx,Ny = self.Nx,self.Ny
        Dx,Dy,dx,dy = self.Dx,self.Dy,self.dx,self.dy
        K11,K22,K12 = self.K['11'],self.K['22'],self.K['12']
        N,D = self.Boundary['Neumann'],self.Boundary['Dirichlet']
        Bv = self.Boundary['Values']
        # Approximate K*grad(p)
        ux,uy = zeros((Nx+1,Ny)),zeros((Nx,Ny+1))
        px,py = self.Gradient(p)
        DX = 2.*(Dx[1:]+Dx[:-1])
        uxc,pxc = ux[1:-1,:],px[1:-1,:]
        k11b,k11t = K11[1:-1,:-1],K11[1:-1,1:]
        Dxr,Dxl = Dx[1:],Dx[:-1]
        Dxr,Dxl = Dx[1:]/DX,Dx[:-1]/DX
        Dxr,Dxl = Dxr[:,np.newaxis],Dxl[:,np.newaxis]
        k12b,k12t = K12[1:-1,:-1],K12[1:-1,1:]
        pyb,pyt = py[1:,:-1],py[1:,1:]
        pyb2,pyt2 = py[:-1,:-1],py[:-1,1:]
        e="pxc*(k11b+k11t)/2.+Dxr*(k12b*pyb+k12t*pyt)+Dxl*(k12b*pyb2+k12t*pyt2)"
```

```
        ne.evaluate(e,out=uxc)
        '''
        ux[1:-1,:] = (px[1:-1,:]*(K11[1:-1,:-1]+K11[1:-1,1:])/2.
          +(Dx[1:]*(K12[1:-1,:-1]*py[1:,:-1]+K12[1:-1,1:]*py[1:,1:]).T/DX).T
          +(Dx[:-1]*(K12[1:-1,:-1]*py[:-1,:-1]+K12[1:-1,1:]*py[:-1,1:]).T/DX).T)
        '''
        ux[0,N['West']] = Bv['West'][N['West']]
        DW = D['West']
        DWp1 = DW + 1
        ux[0,DW] = ((K11[0,DW]+K11[0,DWp1])*px[0,DW]/2. +
                    (K12[0,DW]*py[0,DW]+K12[0,DWp1]*py[0,DWp1]))
        ux[-1,N['East']] = Bv['East'][N['East']]
        DE = D['East']
        DEp1 = DE + 1
        ux[-1,DE] = ((K11[-1,DE]+K11[-1,DEp1])*px[-1,DE]/2. +
                    (K12[-1,DE]*py[-1,DE]+K12[-1,DEp1]*py[-1,DEp1]))
        DY = 2.*(Dy[1:]+Dy[:-1])
        uyc,pyc = uy[:,1:-1],py[:,1:-1]
        k22l,k22r = K22[:-1,1:-1],K22[1:,1:-1]
        Dyt,Dyb = Dy[1:],Dy[:-1]
        Dyt,Dyb = Dy[1:]/DY,Dy[:-1]/DY
        k12l,k12r = K12[:-1,1:-1],K12[1:,1:-1]
        pxl,pxr = px[:-1,1:],px[1:,1:]
        pxl2,pxr2 = px[:-1,:-1],px[1:,:-1]
        e="pyc*(k22l+k22r)/2.+Dyt*(k12l*pxl+k12r*pxr)+Dyb*(k12l*pxl2+k12r*pxr2)"
        ne.evaluate(e,out=uyc)
        '''
        uy[:,1:-1] = (py[:,1:-1]*(K22[:-1,1:-1]+K22[1:,1:-1])/2.
              + Dy[1:]*(K12[:-1,1:-1]*px[:-1,1:]+K12[1:,1:-1]*px[1:,1:])/DY
              + Dy[:-1]*(K12[:-1,1:-1]*px[:-1,:-1]+K12[1:,1:-1]*px[1:,:-1])/DY)
        '''
        uy[N['South'],0] = Bv['South'][N['South']]
        DS = D['South']
        DSp1 = DS + 1
        uy[DS,0] = ((K22[DS,0]+K22[DSp1,0])*py[DS,0]/2. +
                    (K12[DS,0]*px[DS,0]+K12[DSp1,0]*px[DSp1,0]))
        uy[N['North'],-1] = Bv['North'][N['North']]
        DN = D['North']
        DNp1 = DN + 1
        uy[DN,-1] = ((K22[DN,-1]+K22[DNp1,-1])*py[DN,-1]/2. +
                    (K12[DN,-1]*px[DN,-1]+K12[DNp1,-1]*px[DNp1,-1]))
        return ux,uy

    def Gradient(self,p):
        Dx,Dy,dx,dy = self.Dx,self.Dy,self.dx,self.dy
        Nx,Ny = self.Nx,self.Ny
        K11,K22,K12 = self.K['11'],self.K['22'],self.K['12']
        N,D = self.Boundary['Neumann'],self.Boundary['Dirichlet']
        Bv = self.Boundary['Values']
        # Approximate grad(p)
        px,py = zeros((Nx+1,Ny)),zeros((Nx,Ny+1))
        dx2 = dx[:,np.newaxis]
        pxc = px[1:-1,:]
        pr,pl = p[1:,:],p[:-1,:]
        ne.evaluate("(pr-pl)/dx2",out=pxc)
        # px[1:-1,:] = ((p[1:,:] - p[:-1,:]).T/dx).T
        px[0,D['West']] = 2.*(p[0,D['West']]-Bv['West'][D['West']])/Dx[0]
```

```
px[-1,D['East']] = 2.*(Bv['East'][D['East']]-p[-1,D['East']])/Dx[-1]
pyc = py[:,1:-1]
pt,pb = p[:,1:],p[:,:-1]
ne.evaluate("(pt-pb)/dy",out=pyc)
#py[:,1:-1] = (p[:,1:] - p[:,:-1])/dy
py[D['South'],0] = 2.*(p[D['South'],0]-Bv['South'][D['South']])/Dy[0]
py[D['North'],-1] =2.*(Bv['North'][D['North']]-p[D['North'],-1])/Dy[-1]
BvW = Bv['West']
for j in N['West']:
    if j == 0:
        if 0 in D['South']:
            px[0,j]=(2.*(BvW[j]-K12[0,j]*py[0,j]-K12[0,j+1]*py[0,j+1])/
                    (K11[0,j+1]+K11[0,j]))
    elif j == Ny-1:
        if 0 in D['North']:
            px[0,j]=(2.*(BvW[j]-K12[0,j]*py[0,j]-K12[0,j+1]*py[0,j+1])/
                    (K11[0,j+1]+K11[0,j]))
    else:
        px[0,j]=(2.*(BvW[j]-K12[0,j]*py[0,j]-K12[0,j+1]*py[0,j+1])/
                    (K11[0,j+1]+K11[0,j]))
BvE = Bv['East']
for j in N['East']:
    if j == 0:
        if Nx in D['South']:
            px[-1,j]=(2.*(BvE[j]-K12[-1,j]*py[-1,j]-
                      K12[-1,j+1]*py[-1,j+1])/
                    (K11[-1,j+1]+K11[-1,j]))
    elif j == Ny-1:
        if Nx in D['North']:
            px[-1,j]=(2.*(BvE[j]-K12[-1,j]*py[-1,j]-
                      K12[-1,j+1]*py[-1,j+1])/
                    (K11[-1,j+1]+K11[-1,j]))
    else:
        px[-1,j]=(2.*
                  (BvE[j]-K12[-1,j]*py[-1,j]-K12[-1,j+1]*py[-1,j+1])/
                    (K11[-1,j+1]+K11[-1,j]))
BvS = Bv['South']
for j in N['South']:
    if j == 0:
        if 0 in D['West']:
            py[j,0]=(2.*(BvS[j]-K12[j,0]*px[j,0]-K12[j+1,0]*py[j+1,0])/
                    (K22[j+1,0]+K22[j,0]))
    elif j == Nx-1:
        if 0 in D['East']:
            py[j,0]=(2.*(BvS[j]-K12[j,0]*px[j,0]-K12[j+1,0]*px[j+1,0])/
                    (K22[j+1,0]+K22[j,0]))
    else:
        py[j,0]=(2.*(BvS[j]-K12[j,0]*px[j,0]-K12[j+1,0]*px[j+1,0])/
                    (K22[j+1,0]+K22[j,0]))
BvN = Bv['North']
for j in N['North']:
    v = BvN[j]
    if j == 0:
        if 0 in D['West']:
            py[j,-1]=(2.*(v-K12[j,-1]*px[j,-1]-K12[j+1,-1]*px[j+1,-1])/
                    (K22[j+1,-1]+K22[j,-1]))
    elif j == Nx-1:
```

```
            if 0 in D['East']:
                py[j,-1]=(2.*(v-K12[j,-1]*px[j,-1]-K12[j+1,-1]*px[j+1,-1])/
                        (K22[j+1,-1]+K22[j,-1]))
        else:
            py[j,-1]=(2.*(v-K12[j,-1]*px[j,-1]-K12[j+1,-1]*px[j+1,-1])/
                        (K22[j+1,-1]+K22[j,-1]))
    # SW corner
    if (0 in N['West']) and (0 in N['South']):
        M = [[(K11[0,0]+K11[0,1])/2.,K12[0,0]],
             [K12[0,0],(K22[0,0]+K22[1,0])/2.]]
        rhs = [[Bv['West'][0]-K12[0,1]*py[0,1]],
               [Bv['South'][0]-K12[1,0]*px[1,0]]]
        ps = solve(M,rhs)
        px[0,0] = ps[0]
        py[0,0] = ps[1]
    # SE corner
    if (0 in N['East']) and (Nx in N['South']):
        M = [[(K11[-1,0]+K11[-1,1])/2.,K12[-1,0]],
             [K12[-1,0],(K22[-1,0]+K22[-2,0])/2.]]
        rhs = [[Bv['East'][0]-K12[-1,1]*py[-1,1]],
               [Bv['South'][-1]-K12[-2,0]*px[-2,0]]]
        ps = solve(M,rhs)
        px[-1,0] = ps[0]
        py[-1,0] = ps[1]
    # NW corner
    if (Ny in N['West']) and (0 in N['North']):
        M = [[(K11[0,-1]+K11[0,-2])/2.,K12[0,-1]],
             [K12[0,-1],(K22[0,-1]+K22[1,-1])/2.]]
        rhs = [[Bv['West'][-1]-K12[0,-2]*py[0,-2]],
               [Bv['North'][0]-K12[1,-1]*px[1,-1]]]
        ps = solve(M,rhs)
        px[0,-1] = ps[0]
        py[0,-1] = ps[1]
    # NE corner
    if (Ny in N['East']) and (Nx in N['North']):
        M = [[(K11[-1,-1]+K11[-1,-2])/2.,K12[-1,-1]],
             [K12[-1,-1],(K22[-1,-1]+K22[-2,-1])/2.]]
        rhs = [[Bv['East'][-1]-K12[-1,-2]*py[-1,-2]],
               [Bv['North'][-1]-K12[-2,-1]*px[-2,-1]]]
        ps = solve(M,rhs)
        px[-1,-1] = ps[0]
        py[-1,-1] = ps[1]
    return px,py

def Divergence(self,ux,uy):
    Dx,Dy,dx,dy = self.Dx,self.Dy,self.dx,self.dy
    # Approximate div(u)
    uxr,uxl = ux[1:,:],ux[:-1,:]
    uyt,uyb = uy[:,1:],uy[:,:-1]
    Dx2 = Dx[:,np.newaxis]
    div = ne.evaluate("Dy*(uxr-uxl)+Dx2*(uyt-uyb)")
    #div = Dy*(ux[1:,:]-ux[:-1,:]) + (Dx*(uy[:,1:]-uy[:,:-1]).T).T
    return div

def residual(self,p):
    Nx,Ny = self.Nx,self.Ny
    ux,uy = self.GetVelocity(p.reshape((Nx,Ny)))
```

```
    return self.getWellSource() - self.Divergence(ux,uy)

def solve(self,p0=None):
    Nx,Ny = self.Nx,self.Ny
    if p0 is None:
        p0 = zeros((Nx,Ny))
    self.RHS = self.SetUpRHS()
    r0 = self.residual(zeros((Nx,Ny))).flatten()
    def A(p):
        return self.residual(p).flatten() - r0
    Aop = LinearOperator((Nx*Ny,Nx*Ny),A,dtype=np.float)
    p = lgmres(Aop,-r0,p0.flat,**self.solverOptions)
    if p[1]:
        print "No convergence of GMRES."
    p = p[0].reshape((Nx,Ny))
    ux,uy = self.GetVelocity(p)
    return p,ux,uy

def residualNonDarcy(self,p):
    ux,uy = self.GetVelocityNonDarcy(p)
    return self.getWellSource() - self.Divergence(ux,uy)

def velocityToCorners(self,ux,uy):
    Nx,Ny = self.Nx,self.Ny
    UX,UY = zeros((Nx+1,Ny+1)),zeros((Nx+1,Ny+1))
    uxfun = interp1d(self.Y,ux)
    uyfun = interp1d(self.X,uy,axis=0)
    UX[:,1:-1] = uxfun(self.y[1:-1])
    UX[:,0] = ux[:,0]
    UX[:,-1] = ux[:,-1]
    UY[1:-1,:] = uyfun(self.x[1:-1])
    UY[0,:] = uy[0,:]
    UY[-1,:] = uy[-1,:]
    return UX,UY

def updateKappa(self,ux,uy=None):
    if uy is not None:
        UX,UY = self.velocityToCorners(ux,uy)
    else:
        Nx,Ny = self.Nx,self.Ny
        uxl = ux[:(Nx+1)*Ny].reshape((Nx+1,Ny))
        uyl = ux[(Nx+1)*Ny:].reshape((Nx,Ny+1))
        UX,UY = self.velocityToCorners(uxl,uyl)
    self.kappa = self.kappa_function(np.abs(UX),np.abs(UY))
    self.RHS = self.SetUpRHS()

def solveNonDarcy(self,kappa_function,p0=None,
                  kappa0=None,DarcyInitial=True,
                  ux0=None,uy0=None):
    Nx,Ny = self.Nx,self.Ny
    if p0 is None:
        p0 = zeros((Nx,Ny))
    self.kappa_function = kappa_function
    if kappa0 is not None:
        self.kappa = kappa0
        self.RHS = self.SetUpRHS()
    elif (ux0 is not None) and (uy0 is None):
```

```
            self.updateKappa(ux0)
            self.RHS = self.SetUpRHS()
        elif (ux0 is not None) and (uy0 is not None):
            self.updateKappa(ux0,uy0)
            self.RHS = self.SetUpRHS()
        else:
            self.updateKappa(zeros((Nx+1,Ny)),zeros((Nx,Ny+1)))
            self.RHS = self.SetUpRHS()
        if DarcyInitial:
            p0,ux,uy = self.solve(p0)
            self.updateKappa(ux,uy)
            if isinstance(DarcyInitial,int) and DarcyInitial > 1:
                for j in xrange(DarcyInitial-1):
                    self.updateKappa(ux,uy)
                    p0,ux,uy = self.solve(p0)
        elif (ux0 is not None) and (uy0 is None):
            ux = ux0[:(Nx+1)*Ny].reshape((Nx+1,Ny))
            uy = ux0[(Nx+1)*Ny:].reshape((Nx,Ny+1))
        elif (ux0 is not None) and (uy0 is not None):
            ux,uy = ux0,uy0
        else:
            ux,uy = self.GetVelocity(p0)
            self.updateKappa(ux,uy)
        self._u = np.append(ux.flat,uy.flat)
        p = newton_krylov(self.residualNonDarcy,p0,
                          **self.NDSolverOptions)
        ux,uy = self.GetVelocityNonDarcy(p)
        return p,ux,uy

if __name__ == '__main__':
    from numpy import linspace,ones,zeros,arange
    Nx,Ny = 10,7
    x,y = linspace(0.,1.,Nx+1),linspace(0.,1.,Ny+1)
    ea = np.array([],dtype=np.int)
    Boundary = {'Neumann':
                {'West':ea,'East':ea,'North':arange(Nx),'South':arange(Nx)},
                'Dirichlet':
                {'West':arange(Ny),'East':arange(Ny),'North':ea,'South':ea},
                'Values':
                {'West':ones(Ny),'East':zeros(Ny),'North':zeros(Nx),
                 'South':zeros(Nx)}}
    Boundary = {'Neumann':
                {'West':arange(Ny),'East':arange(Ny),'North':ea,'South':ea},
                'Dirichlet':
                {'West':ea,'East':ea,'North':arange(Nx),'South':arange(Nx)},
                'Values':
                {'West':zeros(Ny),'East':zeros(Ny),'North':zeros(Nx),
                 'South':ones(Nx)}}
    F = zeros((Nx,Ny))
    g = {'1x':zeros((Nx+1,Ny)),'2y':zeros((Nx,Ny+1)),'2x':zeros((Nx+1,Ny)),
         '1y':zeros((Nx,Ny+1))}
    K = {'11':ones((Nx+1,Ny+1)),'22':ones((Nx+1,Ny+1)),'12':zeros((Nx+1,Ny+1))}
    ds = DarcySolver(x,y,Boundary,K,F,g)
    p,ux,uy = ds.solve()
    print p
    print ux
    print uy
```

Listing 2: transportSolver.py

```
""" The transport solver. """

import scipy.sparse.linalg as la
from transportLinearOperators import advectionOperator,diffusionOperator
from transportLinearOperators import Side,Boundary,getInflowOutflow
from scipy.sparse.linalg import LinearOperator as LO
import numpy as np
from numpy import arange,zeros

class advectionSolver(object):
    def __init__(self,dx,dy,ux,uy,boundary,source=None,sourcet=False):
        self.Nx,self.Ny = len(dx),len(dy)
        self.dx,self.dy = dx,dy
        self.dxdy = np.outer(dx,dy)
        self.ux,self.uy = ux,uy
        self.boundary = boundary
        if sourcet:
            self.source = source
            self.timestep = self.timestep_with_sourcet
            self.timestep_T = self.timestep_with_sourcet_T
        elif source is not None:
            self.source = source
            self.timestep = self.timestep_with_source
            self.timestep_T = self.timestep_with_source_T
        else:
            self.timestep = self.timestep_without_source
            self.timestep_T = self.timestep_without_source_T
        self.advOp = advectionOperator(dx,dy,ux,uy,boundary)
        self.adv = self.advOp.linOp()

    def timestep_without_source(self,c0,dt,Nt,savesteps=False):
        Nx,Ny = self.Nx,self.Ny
        if savesteps:
            times = dt*arange(Nt+1)
            steps = zeros((Nt+1,Nx,Ny))
            steps[0,:,:] = c0
            for j in xrange(Nt):
                step = steps[j,:,:].flat
                steps[j+1,:,:] = steps[j,:,:]
                steps[j+1,:,:] -= dt*(self.adv*step).reshape((Nx,Ny))
            return zip(times,steps)
        else:
            c = c0.copy().flat
            for j in xrange(Nt):
                c -= dt*(self.adv*c)
            return c.reshape((Nx,Ny))

    def timestep_with_source(self,c0,dt,Nt,savesteps=False):
        Nx,Ny = self.Nx,self.Ny
        source = self.source
        dxdy = self.dxdy
        if savesteps:
            times = dt*arange(Nt+1)
            steps = zeros((Nt+1,Nx,Ny))
            steps[0,:,:] = c0
```

```
        for j in xrange(Nt):
            step = steps[j,:,:].flat
            steps[j+1,:,:] = steps[j,:,:]
            steps[j+1,:,:] -= dt*(self.adv*step).reshape((Nx,Ny))
            steps[j+1,:,:] += dt*source/dxdy
        return zip(times,steps)
    else:
        c = c0.copy().flat
        for j in xrange(Nt):
            c -= dt*(self.adv*c)
            c += dt*(source/dxdy).flat
        return c.reshape((Nx,Ny))

def timestep_with_sourcet(self,c0,dt,Nt,savesteps=False):
    Nx,Ny = self.Nx,self.Ny
    source = self.source
    dxdy = self.dxdy
    if savesteps:
        times = dt*arange(Nt+1)
        steps = zeros((Nt+1,Nx,Ny))
        steps[0,:,:] = c0
        for j in xrange(Nt):
            step = steps[j,:,:].flat
            steps[j+1,:,:] = steps[j,:,:]
            steps[j+1,:,:] -= dt*(self.adv*step).reshape(Nx,Ny)
            steps[j+1,:,:] += dt*(source[j]/dxdy)
        return zip(times,steps)
    else:
        c = c0.copy().flat
        for j in xrange(Nt):
            c -= dt*(self.adv*c)
            c += dt*(source[j]/dxdy).flat
        return c.reshape((Nx,Ny))

def timestep_without_source_T(self,c0,dt,Tfinal,savesteps=False):
    Nx,Ny = self.Nx,self.Ny
    Nt = int(np.ceil(Tfinal/dt)) - 1
    if dt*Nt < Tfinal:
        times = dt*arange(Nt+2)
        times[-1] = Tfinal
        steps = zeros((Nt+2,Nx,Ny))
        steps[0,:,:] = c0
    elif dt*Nt >= Tfinal:
        times = dt*arange(Nt+1)
        times[-1] = Tfinal
        steps = zeros((Nt+1,Nx,Ny))
        steps[0,:,:] = c0
    if savesteps:
        for j in xrange(Nt):
            step = steps[j,:,:].flat
            steps[j+1,:,:] = steps[j,:,:]
            steps[j+1,:,:] -= dt*(self.adv*step).reshape((Nx,Ny))
        if dt*Nt < Tfinal:
            dtF = Tfinal - dt*Nt
            step = steps[-2,:,:].flat
            steps[-1,:,:] = steps[-2,:,:]
            steps[-1,:,:] -= dtF*(self.adv*step).reshape((Nx,Ny))
```

```
            return zip(times,steps)
        else:
            c = c0.copy().flat
            for j in xrange(Nt):
                c -= dt*(self.adv*c)
            if dt*Nt < Tfinal:
                dtF = Tfinal - dt*Nt
                c -= dtF*(self.adv*c)
            return c.reshape((Nx,Ny))

    def timestep_with_source_T(self,c0,dt,Tfinal,savesteps=False):
        Nx,Ny = self.Nx,self.Ny
        source = self.source
        dxdy = self.dxdy
        Nt = int(np.ceil(Tfinal/dt)) - 1
        if dt*Nt < Tfinal:
            times = dt*arange(Nt+2)
            times[-1] = Tfinal
            steps = zeros((Nt+2,Nx,Ny))
            steps[0,:,:] = c0
        elif dt*Nt >= Tfinal:
            times = dt*arange(Nt+1)
            times[-1] = Tfinal
            steps = zeros((Nt+1,Nx,Ny))
            steps[0,:,:] = c0
        if savesteps:
            for j in xrange(Nt):
                step = steps[j,:,:].flat
                steps[j+1,:,:] = steps[j,:,:]
                steps[j+1,:,:] -= dt*(self.adv*step).reshape((Nx,Ny))
                steps[j+1,:,:] += dt*source/dxdy
            if dt*Nt < Tfinal:
                dtF = Tfinal - dt*Nt
                step = steps[-2,:,:].flat
                steps[-1,:,:] = steps[-2,:,:]
                steps[-1,:,:] -= dtF*(self.adv*step).reshape((Nx,Ny))
                steps[-1,:,:] += dtF*source/dxdy
            return zip(times,steps)
        else:
            c = c0.copy().flat
            for j in xrange(Nt):
                c -= dt*(self.adv*c)
                c += dt*(source/dxdy).flat
            if dt*Nt < Tfinal:
                dtF = Tfinal - dt*Nt
                c -= dtF*(self.adv*c)
                c += dtF*(source/dxdy).flat
            return c.reshape((Nx,Ny))

    def timestep_with_sourcet_T(self,c0,dt,Tfinal,savesteps=False):
        Nx,Ny = self.Nx,self.Ny
        source = self.source
        dxdy = self.dxdy
        Nt = int(np.ceil(Tfinal/dt)) - 1
        if dt*Nt < Tfinal:
            times = dt*arange(Nt+2)
            times[-1] = Tfinal
```

```
            steps = zeros((Nt+2,Nx,Ny))
            steps[0,:,:] = c0
        elif dt*Nt >= Tfinal:
            times = dt*arange(Nt+1)
            times[-1] = Tfinal
            steps = zeros((Nt+1,Nx,Ny))
            steps[0,:,:] = c0
        if savesteps:
            times = dt*arange(Nt+1)
            steps = zeros((Nt+1,Nx,Ny))
            steps[0,:,:] = c0
            for j in xrange(Nt):
                step = steps[j,:,:].flat
                steps[j+1,:,:] = steps[j,:,:]
                steps[j+1,:,:] -= dt*(self.adv*step).reshape(Nx,Ny)
                steps[j+1,:,:] += dt*(source[j]/dxdy)
            if dt*Nt < Tfinal:
                dtF = Tfinal - dt*Nt
                step = steps[-2,:,:].flat
                steps[-1,:,:] = steps[-2,:,:]
                steps[-1,:,:] -= dtF*(self.adv*step).reshape((Nx,Ny))
                steps[-1,:,:] += dtF*source[-1]/dxdy
            return zip(times,steps)
        else:
            c = c0.copy().flat
            for j in xrange(Nt):
                c -= dt*(self.adv*c)
                c += dt*(source[j]/dxdy).flat
            if dt*Nt < Tfinal:
                dtF = Tfinal - dt*Nt
                c -= dtF*(self.adv*c)
                c += dtF*(source/dxdy).flat
            return c.reshape((Nx,Ny))

class advectionDiffusionSolver(object):
    def __init__(self,dx,dy,ux,uy,Dx,Dy,boundary,source=None,sourcet=False,
                 slvrname='gmres',slvrops=None):
        self.Nx,self.Ny = len(dx),len(dy)
        Nx,Ny = len(dx),len(dy)
        self.dx,self.dy = dx,dy
        self.dxdy = np.outer(dx,dy)
        self.ux,self.uy = ux,uy
        self.Dx,self.Dy = Dx,Dy
        self.boundary = boundary
        if sourcet:
            self.source = source
            self.timestep = self.timestep_with_sourcet
        elif source:
            self.source = source.flat
            self.timestep = self.timestep_with_source
        else:
            self.timestep = self.timestep_without_source
        self.advOp = advectionOperator(dx,dy,ux,uy,boundary)
        self.adv = self.advOp.linOp()
        self.diffOp = diffusionOperator(dx,dy,Dx,Dy,boundary)
        self.diff = self.diffOp.linOp()
        if slvrops is None:
```

```
            self.slvrops = {'tol':1e-12}
        self.slvr = {'gmres':la.gmres,
                     'bicg':la.bicg,
                     'biststab':la.bicgstab,
                     'cg':la.cg,
                     'cgs':la.cgs,
                     'lgmres':la.lgmres,
                     'minres':la.minres,
                     'qmr':la.qmr}.get(slvrname,la.gmres)

    def timestep_without_source(self,c0,dt,Nt,savesteps=False):
        Nx,Ny = self.Nx,self.Ny
        c = c0.copy().flat
        def lhs(x):
            return x + dt*(self.diff*x)
        self.op = LO((Nx*Ny,Nx*Ny),lhs,dtype=np.float64)
        if savesteps:
            steps = [(0.,c0.copy())]
        for j in xrange(Nt):
            rhs = c - dt*(self.adv*c)
            c,success = self.slvr(self.op,rhs,rhs,**self.slvrops)
            if savesteps:
                steps.append((j*dt+dt,c.copy().reshape((Nx,Ny))))
        if savesteps:
            return steps
        else:
            return c.reshape((Nx,Ny))

    def timestep_with_source(self,c0,dt,Nt,savesteps=False):
        Nx,Ny = self.Nx,self.Ny
        dxdy = self.dxdy
        c = c0.copy().flat
        source = self.source
        def lhs(x):
            return x - dt*(self.diff*x)
        self.op = LO((Nx*Ny,Nx*Ny),lhs,dtype=np.float64)
        if savesteps:
            steps = [(0.,c0.copy())]
        for j in xrange(Nt):
            rhs = c - dt*(self.adv*c) + dt*source/dxdy
            c,success = self.slvr(self.op,rhs,rhs,**self.slvrops)
            if savesteps:
                steps.append((j*dt+dt,c.copy().reshape((Nx,Ny))))
        if savesteps:
            return steps
        else:
            return c.reshape((Nx,Ny))

    def timestep_with_sourcet(self,c0,dt,Nt,savesteps=False):
        Nx,Ny = self.Nx,self.Ny
        dxdy = self.dxdy
        c = c0.copy().flat
        source = self.source
        def lhs(x):
            return x - dt*(self.diff*x)
        self.op = LO((Nx*Ny,Nx*Ny),lhs,dtype=np.float64)
        if savesteps:
```

```
            steps = [(0.,c0.copy())]
        for j in xrange(Nt):
            rhs = c - dt*(self.adv*c) + dt*(source[j]/dxdy).flat
            c,success = self.slvr(self.op,rhs,rhs,**self.slvrops)
            if savesteps:
                steps.append((j*dt+dt,c.copy().reshape((Nx,Ny))))
        if savesteps:
            return steps
        else:
            return c.reshape((Nx,Ny))

def zeroInitialOneInflow(x,y,ux,uy,dt,Nt,savesteps=True):
    ones = np.ones
    zeros = np.zeros
    Nx = uy.shape[0]
    Ny = ux.shape[1]
    dx = x[1:] - x[:-1]
    dy = y[1:] - y[:-1]
    wi,wo,ei,eo,si,so,ni,no = getInflowOutflow(ux,uy)
    west = Side(wi,wo,ones(len(wi)),ones(len(wo)))
    east = Side(ei,eo,ones(len(ei)),ones(len(eo)))
    north = Side(ni,no,ones(len(ni)),ones(len(no)))
    south = Side(si,so,ones(len(si)),ones(len(so)))
    bnd = Boundary(west,east,south,north)
    slvr = advectionSolver(dx,dy,ux,uy,bnd)
    cfinal = slvr.timestep(zeros((Nx,Ny)),dt,Nt,savesteps)
    return cfinal,bnd

def zeroInitialOneInflow_T(x,y,ux,uy,dt,Tfinal,savesteps=True):
    ones = np.ones
    zeros = np.zeros
    Nx = uy.shape[0]
    Ny = ux.shape[1]
    dx = x[1:] - x[:-1]
    dy = y[1:] - y[:-1]
    wi,wo,ei,eo,si,so,ni,no = getInflowOutflow(ux,uy)
    west = Side(wi,wo,ones(len(wi)),ones(len(wo)))
    east = Side(ei,eo,ones(len(ei)),ones(len(eo)))
    north = Side(ni,no,ones(len(ni)),ones(len(no)))
    south = Side(si,so,ones(len(si)),ones(len(so)))
    bnd = Boundary(west,east,south,north)
    slvr = advectionSolver(dx,dy,ux,uy,bnd)
    cfinal = slvr.timestep_T(zeros((Nx,Ny)),dt,Tfinal,savesteps)
    return cfinal,bnd
```

Listing 3: transportLinearOperators.py

```
""" Linear operators for the advection-diffusion equation. """

from collections import namedtuple
from numpy import zeros,diff,outer,append,float64,zeros_like,flatnonzero
from scipy.sparse.linalg import LinearOperator as LO

class advectionOperator(object):
    def __init__(self,dx,dy,ux,uy,boundary):
```

```
        self.Nx,self.Ny = len(dx),len(dy)
        self.dx = dx
        self.dy = dy
        self.boundary = boundary
        self.ux = ux
        self.uy = uy
        self.uxp = ux.copy()
        self.uxp[ux <= 0.] = 0.
        self.uxn = ux.copy()
        self.uxn[ux > 0.] = 0.
        self.uyp = uy.copy()
        self.uyp[uy <= 0.] = 0.
        self.uyn = uy.copy()
        self.uyn[uy > 0.] = 0.
        self._fx = zeros_like(ux)
        self._fy = zeros_like(uy)

    def Flux(self,c):
        b = self.boundary
        uxp,uxn = self.uxp,self.uxn
        uyp,uyn = self.uyp,self.uyn
        self._fx[1:-1,:] = uxp[1:-1,:]*c[:-1,:] + uxn[1:-1,:]*c[1:,:]
        self._fx[0,b.W.I] = uxp[0,b.W.I]*b.W.Iv
        self._fx[0,b.W.O] = uxn[0,b.W.O]*c[0,b.W.O]
        self._fx[-1,b.E.I] = uxn[-1,b.E.I]*b.E.Iv
        self._fx[-1,b.E.O] = uxp[-1,b.E.O]*c[-1,b.E.O]
        self._fy[:,1:-1] = uyp[:,1:-1]*c[:,:-1] + uyn[:,1:-1]*c[:,1:]
        self._fy[b.S.I,0] = uyp[b.S.I,0]*b.S.Iv
        self._fy[b.S.O,0] = uyn[b.S.O,0]*c[b.S.O,0]
        self._fy[b.N.I,-1] = uyn[b.N.I,-1]*b.N.Iv
        self._fy[b.N.O,-1] = uyp[b.N.O,-1]*c[b.N.O,-1]
        return self._fx,self._fy

    def matvec(self,c):
        Nx,Ny = self.Nx,self.Ny
        dx,dy = self.dx,self.dy
        self.Flux(c.reshape((Nx,Ny)))
        F = (diff(self._fx,axis=0).T/dx).T + diff(self._fy,axis=1)/dy
        return F.flat

    def linOp(self):
        Nx,Ny = self.Nx,self.Ny
        self.lo = LO((Nx*Ny,Nx*Ny),self.matvec,dtype=float64)
        return self.lo

class diffusionOperator(object):
    def __init__(self,dx,dy,Dx,Dy,boundary):
        self.Nx,self.Ny = len(dx),len(dy)
        self.dx = dx
        self.dy = dy
        self.ddx = (dx[1:] + dx[:-1])/2.
        self.ddy = (dy[1:] + dy[:-1])/2.
        self.Dx = Dx
        self.Dy = Dy
        self.boundary = boundary
        self._fx = zeros_like(Dx)
        self._fy = zeros_like(Dy)
```

```
    def Flux(self,c):
        Nx,Ny = self.Nx,self.Ny
        b = self.boundary
        C = c.reshape((Nx,Ny))
        self._fx[1:-1,:] = (diff(C,axis=0).T/self.ddx).T
        self._fx[0,b.W.I] = 2.*(C[0,b.W.I] - b.W.Iv)/self.dx[0]
        self._fx[-1,b.E.I] = 2.*(b.E.Iv - C[-1,b.E.I])/self.dx[-1]
        self._fx *= self.Dx
        self._fx[0,b.W.O] = -b.W.Ov
        self._fx[-1,b.E.O] = b.E.Ov
        self._fy[:,1:-1] = diff(C,axis=1)/self.ddy
        self._fy[b.S.I,0] = 2.*(C[b.S.I,0] - b.S.Iv)/self.dy[0]
        self._fy[b.N.I,-1] = 2.*(b.N.Iv - C[b.N.I,-1])/self.dy[-1]
        self._fy *= self.Dy
        self._fy[b.S.O,0] = -b.S.Ov
        self._fy[b.N.O,-1] = b.N.Ov
        return self._fx,self._fy

    def matvec(self,c):
        fx,fy = self.Flux(c)
        dx,dy = self.dx,self.dy
        F = fx[:-1,:]/dx - fx[1:,:]/dx + fy[:,:-1]/dy - fy[:,1:]/dy
        return F.flat

    def linOp(self):
        Nx,Ny = self.Nx,self.Ny
        self.lo = LO((Nx*Ny,Nx*Ny),self.matvec,dtype=float64)
        return self.lo

Side = namedtuple('Side',['I','O','Iv','Ov'])

Boundary = namedtuple('Boundary',['W','E','S','N'])

def getInflowOutflow(ux,uy):
    wi = flatnonzero(ux[0,:]>0)
    wo = flatnonzero(ux[0,:]<=0)
    ei = flatnonzero(ux[-1,:]<0)
    eo = flatnonzero(ux[-1,:]>=0)
    si = flatnonzero(uy[:,0]>0)
    so = flatnonzero(uy[:,0]<=0)
    ni = flatnonzero(uy[:,-1]<0)
    no = flatnonzero(uy[:,-1]>=0)
    return wi,wo,ei,eo,si,so,ni,no
```

Listing 4: experimentDriver.py

```
""" A script to run the experiments. """

import matplotlib
matplotlib.use('PDF')
import matplotlib.pyplot as plt
from flowSolver import DarcySolver
from transportSolver import zeroInitialOneInflow,advectionSolver
from transportSolver import zeroInitialOneInflow_T
```

```
from transportLinearOperators import getInflowOutflow
from transportLinearOperators import Side,Boundary
import numpy as np
from numpy import append,zeros_like,abs,ceil,ones,zeros,log,where
from utility import integrateVelocity
from matrix2latex import matrix2latex
from scipy.sparse.linalg import LinearOperator as LO
from numpy import float64,diff
from scipy.integrate import simps

def runFlowExperiment(x,y,Kappa,KappaS,boundary,boundaryS,parameters,
                      darcyInitial=True,tol=1e-10):
    """ x,y are spatial discretizations
        Kappa is the resistance operator
        KappaS is the resistance for the sensitivity equation
        boundary is the boundary conditions
        boundaryS is the boundary conditions for the sensitivity equation
        parameters is a container of dicts. Each parameter should have keys:
            'name', 'gravity', 'dKdp10x', 'dKdp10y', 'dKdp01x', 'dKdp01y'
        darcyInitial controls whether a Darcy initial guess is used

    """
    Nx,Ny = len(x)-1,len(y)-1
    slvr = DarcySolver(x,y,boundary,NDSolverOptions={'f_tol':tol,
                                                     'verbose':True})
    p,ux,uy = slvr.solveNonDarcy(Kappa,DarcyInitial=darcyInitial)
    UX,UY = slvr.velocityToCorners(ux,uy)
    KS = KappaS(UX,UY)
    """ Set up and solve the AS systems. """
    g10 = {'1x':ones((Nx+1,Ny)),'2y':zeros((Nx,Ny+1)),'2x':zeros((Nx+1,Ny)),
           '1y':ones((Nx,Ny+1))}
    ASsolver = DarcySolver(x,y,boundaryS,KS,g=g10,
                           solverOptions={'tol':tol})
    ASsolver.kappa = KS
    mu10,vx10,vy10 = ASsolver.solve()
    g01 = {'1x':zeros((Nx+1,Ny)),'2y':ones((Nx,Ny+1)),'2x':ones((Nx+1,Ny)),
           '1y':zeros((Nx,Ny+1))}
    ASsolver = DarcySolver(x,y,boundaryS,KS,g=g01,
                           solverOptions={'tol':tol})
    ASsolver.kappa = KS
    mu01,vx01,vy01 = ASsolver.solve()
    """ For each parameter, set up and solve the FS system. Also integrate. """
    sensitivities = {}
    for pi in parameters:
        g = pi['gravity'](ux,uy)
        FSsolver = DarcySolver(x,y,boundaryS,KS,g=g,
                               solverOptions={'tol':tol})
        FSsolver.kappa = KS
        ps,uxs,uys = FSsolver.solve()
        """ Compute FS integrals. """
        H10FS = integrateVelocity(x,y,uxs,'x')
        H01FS = integrateVelocity(x,y,uys,'y')
        """ Compute AS integrals. """
        H10AS = -(integrateVelocity(x,y,pi['dKdp10x'](ux,uy)*vx10,'x') +
                  integrateVelocity(x,y,pi['dKdp10y'](ux,uy)*vy10,'y'))
        H01AS = -(integrateVelocity(x,y,pi['dKdp01x'](ux,uy)*vx01,'x') +
                  integrateVelocity(x,y,pi['dKdp01y'](ux,uy)*vy01,'y'))
```

```
            sensitivities.update({pi['name']:{'H10FS':H10FS,'H01FS':H01FS,
                                              'H10AS':H10AS,'H01AS':H01AS,
                                              'ps':ps,'uxs':uxs,'uys':uys}})
        return p,ux,uy,KS,sensitivities

    def runTransportExperiment(x,y,ux,uy,sensitivities,parameters,Tfinal,KS,
                               boundaryS,tol=1e-10):
        dx,dy = x[1:]-x[:-1],y[1:]-y[:-1]
        dxdy = np.outer(dx,dy)
        Nx,Ny = len(dx),len(dy)
        ''' Compute dt and Nt. '''
        h = min(min(dx),min(dy))
        UX = abs(ux).max()
        UY = abs(uy).max()
        dt = h/(UX+UY)
        Nt = int(ceil(Tfinal/dt)) - 1
        """ Solve the transport problem. """
        #c,bnd = zeroInitialOneInflow(x,y,ux,uy,dt,Nt)
        c,bnd = zeroInitialOneInflow_T(x,y,ux,uy,dt,Tfinal)
        Nt = len(c)-1
        """ For each parameter, set up and solve the FS transport problem. """
        parameters2 = sensitivities.keys()
        sensint = zeros_like(c[0][1])
        for pii in parameters2:
            pi = sensitivities[pii]
            uxs,uys = pi['uxs'],pi['uys']
            """ Use the value of c on the boundary that was used in the computation
                of c. This should aid in keeping the system consistent. """
            upwindC = advectionUpwindC(dx,dy,ux,uy,bnd)
            pisrc = []
            #for j in xrange(Nt+1):
            for j in xrange(Nt):
                cx,cy = upwindC.Flux(c[j][1])
                pisrc.append(-((np.diff(cx*uxs,axis=0)*dy)
                               + (np.diff(cy*uys,axis=1).T*dx).T))
            wi,wo,ei,eo,si,so,ni,no = getInflowOutflow(ux,uy)
            west = Side(wi,wo,zeros(len(wi)),zeros(len(wo)))
            east = Side(ei,eo,zeros(len(ei)),zeros(len(eo)))
            north = Side(ni,no,zeros(len(ni)),zeros(len(no)))
            south = Side(si,so,zeros(len(si)),zeros(len(so)))
            sbnd = Boundary(west,east,south,north)
            slvr = advectionSolver(dx,dy,ux,uy,sbnd,pisrc,True)
            #cpi = slvr.timestep(zeros((Nx,Ny)),dt,Nt,True)
            cpi = slvr.timestep_T(zeros((Nx,Ny)),dt,Tfinal,True)
            """ Compute the sensitivity of the objective funtion. """
            #cpi2 = zeros(Nt+1)
            cpi2 = zeros(Nt)
            ts = []
            #for j in xrange(Nt+1):
            for j in xrange(Nt):
                ts.append(cpi[j][0])
                cpi2[j] = sum((cpi[j][1]*dxdy).flatten())
            sensint = simps(cpi2,ts)/Tfinal
            pi.update({'Gsfs':sensint})
            pi.update({'csfs':cpi,'pisrc':pisrc})
        """ Set up and solve the AS transport problem. """
        wi,wo,ei,eo,si,so,ni,no = getInflowOutflow(-ux,-uy)
```

```
    west = Side(wi,wo,zeros(len(wi)),zeros(len(wo)))
    east = Side(ei,eo,zeros(len(ei)),zeros(len(eo)))
    north = Side(ni,no,zeros(len(ni)),zeros(len(no)))
    south = Side(si,so,zeros(len(si)),zeros(len(so)))
    sbnd = Boundary(west,east,south,north)
    ssrc = dxdy/Tfinal
    asslvr = advectionSolver(dx,dy,-ux,-uy,sbnd,ssrc)
    #asc = asslvr.timestep(zeros((len(dx),len(dy))),dt,Nt+1,True)
    asc = asslvr.timestep_T(zeros((len(dx),len(dy))),dt,Tfinal,True)
    upwindC = advectionUpwindC(dx,dy,ux,uy,bnd)
    xitx = zeros((Nx+1,Ny,Nt))
    xity = zeros((Nx,Ny+1,Nt))
    xiw,xie = zeros((Ny,Nt)),zeros((Ny,Nt))
    xis,xin = zeros((Nx,Nt)),zeros((Nx,Nt))
    ts = []
    Dx,Dy = (dx[1:]+dx[:-1])/2.,(dy[1:]+dy[:-1])/2.
    for ct,qtau,ti in zip(c,asc[::-1],xrange(Nt)):
        qt = qtau[1]
        qx = ((qt[1:,:]-qt[:-1,:]).T/Dx).T
        qy = (qt[:,1:]-qt[:,:-1])/Dy
        cx,cy = upwindC.Flux(ct[1])
        xitx[1:-1,:,ti] = -cx[1:-1,:]*qx
        xity[:,1:-1,ti] = -cy[:,1:-1]*qy
        ts.append(ct[0])
        xiw[:,ti] = qt[0,:]*cx[0,:]
        xie[:,ti] = qt[-1,:]*cx[-1,:]
        xis[:,ti] = qt[:,0]*cy[:,0]
        xin[:,ti] = qt[:,-1]*cy[:,-1]
    xix = -simps(xitx,ts)
    xiy = -simps(xity,ts)
    xiw = -simps(xiw,ts)
    xie = -simps(xie,ts)
    xis = -simps(xis,ts)
    xin = -simps(xin,ts)
    xiw[boundaryS['Neumann']['West']] = 0.
    xie[boundaryS['Neumann']['East']] = 0.
    xis[boundaryS['Neumann']['South']] = 0.
    xin[boundaryS['Neumann']['North']] = 0.
    """ Use xi to solve the adjoint flow problem. """
    xi = {'1x':xix,'2x':zeros((Nx+1,Ny)),'1y':zeros((Nx,Ny+1)),'2y':xiy}
    bnd = {'Neumann':boundaryS['Neumann'],'Dirichlet':boundaryS['Dirichlet'],
           'Values':{'West':-xiw,'East':-xie,'South':-xis,'North':-xin}}
    ASsolver = DarcySolver(x,y,bnd,KS,g=xi,
                           solverOptions={'tol':tol})
    ASsolver.kappa = KS
    mu,vx,vy = ASsolver.solve()
    for Pi in parameters:
        GAS = -(integrateVelocity(x,y,Pi['dKdp10x'](ux,uy)*vx,'x') +
                integrateVelocity(x,y,Pi['dKdp10y'](ux,uy)*vy,'y'))
        sensitivities[Pi['name']].update({'GAS':GAS})
        sensitivities.update({'asc':asc,'xi':xi,'mu':mu,'vx':vx,'vy':vy,
                              'xitx':xitx,'xity':xity,'xix':xix,'xiy':xiy})
    return c,sensitivities

def generateFlowSensitivityErrorTable(filename,caption,experiments,parameters):
    hc = experiments.keys() # Names of the experiments.
    hr,ha = [],[] # hr will contain the names of the sensitivities.
```

```
    hr1,hr1a = [],[]
    st = '$\\left\\{{ \\partial_{{{}}}H_{{{}}}\\right\\}}$'
    for pi in parameters:
        hr1 += ['Forward Sensitivity Error']*2
        hr1a += ['Adjoint Sensitivity Error']*2
        hr.append(st.format(pi,'1'))
        hr.append(st.format(pi,'2'))
        hr.append(st.format(pi,'1'))
        hr.append(st.format(pi,'2'))
    hr += ha # Both FS and AS
    hr1 += hr1a
    hr = [hr1,hr]
    m = [] # Contains the entries in the table.
    for r in hc:
        rfs,ras = [],[]
        experiment = experiments[r]
        ss = experiment['sensitivities']
        for pi in parameters:
            sp = ss[pi]
            rfs.extend([sp['H10FS Error'],sp['H01FS Error']])
            ras.extend([sp['H10AS Error'],sp['H01AS Error']])
        rfs += ras
        m.append(rfs)
    forcol = ['%.2e']*4*len(parameters)
    t = matrix2latex(m,filename,'tabular',headerRow=hr,
                     headerColumn=hc,caption=caption,formatColumn=forcol)
    return t

def generateFlowFSErrorTable(filename,caption,experiments,parameters):
    hc = experiments.keys() # Names of the experiments.
    hr,ha = [],[] # hr will contain the names of the sensitivities.
    hr1,hr1a = [],[]
    st = '$\\left\\{{ \\partial_{{{}}}H_{{{}}}\\right\\}}$'
    for pi in parameters:
        hr1 += ['Forward Sensitivity Error']*2
        hr1a += ['Adjoint Sensitivity Error']*2
        hr.append(st.format(pi,'1'))
        hr.append(st.format(pi,'2'))
        #hr.append(st.format(pi,'1'))
        #hr.append(st.format(pi,'2'))
    #hr += ha # Both FS and AS
    #hr1 += hr1a
    hr = [hr1,hr]
    m = [] # Contains the entries in the table.
    for r in hc:
        rfs,ras = [],[]
        experiment = experiments[r]
        ss = experiment['sensitivities']
        for pi in parameters:
            sp = ss[pi]
            rfs.extend([sp['H10FS Error'],sp['H01FS Error']])
            ras.extend([sp['H10AS Error'],sp['H01AS Error']])
        #rfs += ras
        m.append(rfs)
    forcol = ['%.2e']*2*len(parameters)
    t = matrix2latex(m,filename,'tabular',headerRow=hr,
                     headerColumn=hc,caption=caption,formatColumn=forcol)
```

```
    return t

def generateFlowASErrorTable(filename,caption,experiments,parameters):
    hc = experiments.keys() # Names of the experiments.
    hr,ha = [],[] # hr will contain the names of the sensitivities.
    hr1,hr1a = [],[]
    st = '$\\left\\{{ \\partial_{{{}}}H_{{{}}}\\right\\}}$'
    for pi in parameters:
        hr1 += ['Forward Sensitivity Error']*2
        hr1a += ['Adjoint Sensitivity Error']*2
        hr.append(st.format(pi,'1'))
        hr.append(st.format(pi,'2'))
        #hr.append(st.format(pi,'1'))
        #hr.append(st.format(pi,'2'))
    #hr = ha # Both FS and AS
    hr1 = hr1a
    hr = [hr1,hr]
    m = [] # Contains the entries in the table.
    for r in hc:
        rfs,ras = [],[]
        experiment = experiments[r]
        ss = experiment['sensitivities']
        for pi in parameters:
            sp = ss[pi]
            rfs.extend([sp['H10FS Error'],sp['H01FS Error']])
            ras.extend([sp['H10AS Error'],sp['H01AS Error']])
        rfs = ras
        m.append(rfs)
    forcol = ['%.2e']*2*len(parameters)
    t = matrix2latex(m,filename,'tabular',headerRow=hr,
                     headerColumn=hc,caption=caption,formatColumn=forcol)
    return t

def generateFlowSensitivityErrorTableWithRefinement(filename,caption,
                                                    experiments,parameters,
                                                    orders):
    hc = [] # Names of the experiments.
    hr,ha = ['$N$'],[] # hr will contain the names of the sensitivities.
    hr1,hr1a = [''],[]
    st = '$\\left\\{{ \\partial_{{{}}}H_{{{}}}\\right\\}}$'
    for pi in parameters:
        hr1 += ['Forward Sensitivity Error']*2
        hr1a += ['Adjoint Sensitivity Error']*2
        hr.append(st.format(pi,'1'))
        hr.append(st.format(pi,'2'))
        ha.append(st.format(pi,'1'))
        ha.append(st.format(pi,'2'))
    hr += ha # Both FS and AS
    hr1 += hr1a
    hr = [hr1,hr]
    m = [] # Contains the entries in the table.
    errs = {}
    for pi in parameters:
        errs.update({pi:{'H10FS Error':[],'H10AS Error':[],
                         'H01FS Error':[],'H01AS Error':[]}})
    for r in experiments.keys():
        if orders:
```

```
    s=r'\multirow{{{}}}{{*}}{{{{'.format(len(experiments[r])+1)
else:
    s=r'\multirow{{{}}}{{*}}{{{{'.format(len(experiments[r]))
s = s + r + r' }}'
hc.append(s)
#hc.append(r)
hs = []
for pi in parameters:
    errs.update({pi:{'H10FS Error':[],'H10AS Error':[],
                     'H01FS Error':[],'H01AS Error':[]}})
for experiment in experiments[r]:
    hc += ['']
    rfs = ['{}'.format(experiment['N'])]
    ras = []
    hs += [experiment['h']]
    ss = experiment['sensitivities']
    for pi in parameters:
        sp = ss[pi]
        rfs.extend(['{:.2e}'.format(sp['H10FS Error']),
                    '{:.2e}'.format(sp['H01FS Error'])])
        ras.extend(['{:.2e}'.format(sp['H10AS Error']),
                    '{:.2e}'.format(sp['H01AS Error'])])
        errs[pi]['H10FS Error'].append(sp['H10FS Error'])
        errs[pi]['H10AS Error'].append(sp['H10AS Error'])
        errs[pi]['H01FS Error'].append(sp['H01FS Error'])
        errs[pi]['H01AS Error'].append(sp['H01AS Error'])
    rfs += ras
    m.append(rfs)
rfs = ['Order']
for pi in parameters:
    if 'H10FS' in orders:
        H10FSo = np.polyfit([log(h) for h in hs],
                            [log(e) for e in
                             errs[pi]['H10FS Error']],1)[0]
        rfs.append('{:.2f}'.format(H10FSo))
    else:
        rfs.append('-')
    if 'H01FS' in orders:
        H01FSo = np.polyfit([log(h) for h in hs],
                            [log(e) for e in
                             errs[pi]['H01FS Error']],1)[0]
        rfs.append('{:.2f}'.format(H01FSo))
    else:
        rfs.append('-')
    if 'H10AS' in orders:
        H10ASo = np.polyfit([log(h) for h in hs],
                            [log(e) for e in
                             errs[pi]['H10AS Error']],1)[0]
        rfs.append('{:.2f}'.format(H10ASo))
    else:
        rfs.append('-')
    if 'H01AS' in orders:
        H01ASo = np.polyfit([log(h) for h in hs],
                            [log(e) for e in
                             errs[pi]['H01AS Error']],1)[0]
        rfs.append('{:.2f}'.format(H01ASo))
    else:
```

```
                rfs.append('-')
            m.append(rfs)
        forcol = ['%s'] + ['%s']*4*len(parameters)
        t = matrix2latex(m,filename,'tabular',headerRow=hr,
                         headerColumn=hc,caption=caption,formatColumn=forcol)
        return t

def generateFlowFSErrorTableWithRefinement(filename,caption,
                                                    experiments,parameters,
                                                    orders):
        hc = [] # Names of the experiments.
        hr,ha = ['$N$'],[] # hr will contain the names of the sensitivities.
        hr1,hr1a = [''],[]
        st = '$\\left\\{{ \\partial_{{{}}}H_{{{}}}\\right\\}}$'
        for pi in parameters:
            hr1 += ['Forward Sensitivity Error']*2
            hr1a += ['Adjoint Sensitivity Error']*2
            hr.append(st.format(pi,'1'))
            hr.append(st.format(pi,'2'))
            ha.append(st.format(pi,'1'))
            ha.append(st.format(pi,'2'))
        #hr += ha # Both FS and AS
        #hr1 += hr1a
        hr = [hr1,hr]
        m = [] # Contains the entries in the table.
        errs = {}
        for pi in parameters:
            errs.update({pi:{'H10FS Error':[],'H10AS Error':[],
                             'H01FS Error':[],'H01AS Error':[]}})
        for r in experiments.keys():
            if orders:
                s=r'\multirow{{{}}}{{*}}{{{{'.format(len(experiments[r])+1)
            else:
                s=r'\multirow{{{}}}{{*}}{{{{'.format(len(experiments[r]))
            s = s + r + r' }}'
            hc.append(s)
            #hc.append(r)
            hs = []
            for pi in parameters:
                errs.update({pi:{'H10FS Error':[],'H10AS Error':[],
                                 'H01FS Error':[],'H01AS Error':[]}})
            for experiment in experiments[r]:
                hc += ['']
                rfs = ['{}'.format(experiment['N'])]
                ras = []
                hs += [experiment['h']]
                ss = experiment['sensitivities']
                for pi in parameters:
                    sp = ss[pi]
                    rfs.extend(['{:.2e}'.format(sp['H10FS Error']),
                                '{:.2e}'.format(sp['H01FS Error'])])
                    ras.extend(['{:.2e}'.format(sp['H10AS Error']),
                                '{:.2e}'.format(sp['H01AS Error'])])
                    errs[pi]['H10FS Error'].append(sp['H10FS Error'])
                    errs[pi]['H10AS Error'].append(sp['H10AS Error'])
                    errs[pi]['H01FS Error'].append(sp['H01FS Error'])
                    errs[pi]['H01AS Error'].append(sp['H01AS Error'])
```

```
            #rfs += ras
            m.append(rfs)
        hc.pop()
        if orders:
            hc += ['']
            rfs = ['Order']
            for pi in parameters:
                if 'H10FS' in orders:
                    H10FSo = np.polyfit([log(h) for h in hs],
                                        [log(e) for e in
                                         errs[pi]['H10FS Error']],1)[0]
                    rfs.append('{:.2f}'.format(H10FSo))
                else:
                    rfs.append('-')
                if 'H01FS' in orders:
                    H01FSo = np.polyfit([log(h) for h in hs],
                                        [log(e) for e in
                                         errs[pi]['H01FS Error']],1)[0]
                    rfs.append('{:.2f}'.format(H01FSo))
                else:
                    rfs.append('-')
                '''
                if 'H10AS' in orders:
                    H10ASo = np.polyfit([log(h) for h in hs],
                                        [log(e) for e in
                                         errs[pi]['H10AS Error']],1)[0]
                    rfs.append('{:.2f}'.format(H10ASo))
                else:
                    rfs.append('-')
                if 'H01AS' in orders:
                    H01ASo = np.polyfit([log(h) for h in hs],
                                        [log(e) for e in
                                         errs[pi]['H01AS Error']],1)[0]
                    rfs.append('{:.2f}'.format(H01ASo))
                else:
                    rfs.append('-')
                '''
            m.append(rfs)
    forcol = ['%s'] + ['%s']*2*len(parameters)
    t = matrix2latex(m,filename,'tabular',headerRow=hr,
                     headerColumn=hc,caption=caption,formatColumn=forcol)
    return t

def generateFlowASErrorTableWithRefinement(filename,caption,
                                                    experiments,parameters,
                                                    orders):
    hc = [] # Names of the experiments.
    hr,ha = ['$N$'],['$N$'] # hr will contain the names of the sensitivities.
    hr1,hr1a = [''],['']
    st = '$\\left\\{{ \\partial_{{{}}}H_{{{}}}\\right\\}}$'
    for pi in parameters:
        hr1 += ['Forward Sensitivity Error']*2
        hr1a += ['Adjoint Sensitivity Error']*2
        hr.append(st.format(pi,'1'))
        hr.append(st.format(pi,'2'))
        ha.append(st.format(pi,'1'))
        ha.append(st.format(pi,'2'))
```

```
hr = ha # Both FS and AS
hr1 = hr1a
hr = [hr1,hr]
m = [] # Contains the entries in the table.
errs = {}
for pi in parameters:
    errs.update({pi:{'H10FS Error':[],'H10AS Error':[],
                     'H01FS Error':[],'H01AS Error':[]}})
for r in experiments.keys():
    if orders:
        s=r'\multirow{{{}}}{{*}}{{{{{'.format(len(experiments[r])+1)
    else:
        s=r'\multirow{{{}}}{{*}}{{{{{'.format(len(experiments[r]))
    s = s + r + r' }}'
    hc.append(s)
    #hc.append(r)
    hs = []
    for pi in parameters:
        errs.update({pi:{'H10FS Error':[],'H10AS Error':[],
                         'H01FS Error':[],'H01AS Error':[]}})
    for experiment in experiments[r]:
        hc += ['']
        rfs = ['{}'.format(experiment['N'])]
        ras = ['{}'.format(experiment['N'])]
        #ras = []
        hs += [experiment['h']]
        ss = experiment['sensitivities']
        for pi in parameters:
            sp = ss[pi]
            rfs.extend(['{:.2e}'.format(sp['H10FS Error']),
                        '{:.2e}'.format(sp['H01FS Error'])])
            ras.extend(['{:.2e}'.format(sp['H10AS Error']),
                        '{:.2e}'.format(sp['H01AS Error'])])
            errs[pi]['H10FS Error'].append(sp['H10FS Error'])
            errs[pi]['H10AS Error'].append(sp['H10AS Error'])
            errs[pi]['H01FS Error'].append(sp['H01FS Error'])
            errs[pi]['H01AS Error'].append(sp['H01AS Error'])
        rfs = ras
        m.append(rfs)
    hc.pop()
    if orders:
        hc += ['']
        rfs = ['Order']
        for pi in parameters:
            '''
            if 'H10FS' in orders:
                H10FSo = np.polyfit([log(h) for h in hs],
                                    [log(e) for e in
                                     errs[pi]['H10FS Error']],1)[0]
                rfs.append('{:.2f}'.format(H10FSo))
            else:
                rfs.append('-')
            if 'H01FS' in orders:
                H01FSo = np.polyfit([log(h) for h in hs],
                                    [log(e) for e in
                                     errs[pi]['H01FS Error']],1)[0]
                rfs.append('{:.2f}'.format(H01FSo))
```

```
                else:
                    rfs.append('-')
                '''
                if 'H10AS' in orders:
                    H10ASo = np.polyfit([log(h) for h in hs],
                                        [log(e) for e in
                                         errs[pi]['H10AS Error']],1)[0]
                    rfs.append('{:.2f}'.format(H10ASo))
                else:
                    rfs.append('-')
                if 'H01AS' in orders:
                    H01ASo = np.polyfit([log(h) for h in hs],
                                        [log(e) for e in
                                         errs[pi]['H01AS Error']],1)[0]
                    rfs.append('{:.2f}'.format(H01ASo))
                else:
                    rfs.append('-')
            m.append(rfs)
        '''
        else:
            m.append(' '*2*len(parameters))
        '''
    forcol = ['%s'] + ['%s']*2*len(parameters)
    t = matrix2latex(m,filename,'tabular',headerRow=hr,
                     headerColumn=hc,caption=caption,formatColumn=forcol)
    return t

def generateTranSensitivityErrorTable(filename,caption,experiments,parameters):
    hc = experiments.keys() # Names of the experiments.
    hr,ha = [],[] # hr will contain the names of the sensitivities.
    hr1,hr1a = [],[]
    st = '$\\left\\{{ \\partial_{{{}}}G\\right\\}}$'
    for pi in parameters:
        hr1 += ['Forward Sensitivity Error']
        hr1a += ['Adjoint Sensitivity Error']
        hr.append(st.format(pi))
        ha.append(st.format(pi))
    hr += ha # Both FS and AS
    hr1 += hr1a
    hr = [hr1,hr]
    m = [] # Contains the entries in the table.
    for r in hc:
        rfs,ras = [],[]
        experiment = experiments[r]
        ss = experiment['sensitivities']
        for pi in parameters:
            sp = ss[pi]
            rfs.extend([sp['Gsfs Error']])
            ras.extend([sp['GAS Error']])
        rfs += ras
        m.append(rfs)
    forcol = ['%.2e']*2*len(parameters)
    t = matrix2latex(m,filename,'tabular',headerRow=hr,
                     headerColumn=hc,caption=caption,formatColumn=forcol)
    return t

def generateTranFSErrorTable(filename,caption,experiments,parameters):
```

```
    hc = experiments.keys() # Names of the experiments.
    hr,ha = [],[] # hr will contain the names of the sensitivities.
    hr1,hr1a = [],[]
    st = '$\\left\\{{ \\partial_{{{}}}G\\right\\}}$'
    for pi in parameters:
        hr1 += ['Forward Sensitivity Error']
        hr1a += ['Adjoint Sensitivity Error']
        hr.append(st.format(pi))
        ha.append(st.format(pi))
    #hr += ha # Both FS and AS
    #hr1 += hr1a
    hr = [hr1,hr]
    m = [] # Contains the entries in the table.
    for r in hc:
        rfs,ras = [],[]
        experiment = experiments[r]
        ss = experiment['sensitivities']
        for pi in parameters:
            sp = ss[pi]
            rfs.extend([sp['Gsfs Error']])
            ras.extend([sp['GAS Error']])
        #rfs += ras
        m.append(rfs)
    forcol = ['%.2e']*len(parameters)
    t = matrix2latex(m,filename,'tabular',headerRow=hr,
                     headerColumn=hc,caption=caption,formatColumn=forcol)
    return t

def generateTranASErrorTable(filename,caption,experiments,parameters):
    hc = experiments.keys() # Names of the experiments.
    hr,ha = [],[] # hr will contain the names of the sensitivities.
    hr1,hr1a = [],[]
    st = '$\\left\\{{ \\partial_{{{}}}G\\right\\}}$'
    for pi in parameters:
        hr1 += ['Forward Sensitivity Error']
        hr1a += ['Adjoint Sensitivity Error']
        hr.append(st.format(pi))
        ha.append(st.format(pi))
    hr = ha # Both FS and AS
    hr1 = hr1a
    hr = [hr1,hr]
    m = [] # Contains the entries in the table.
    for r in hc:
        rfs,ras = [],[]
        experiment = experiments[r]
        ss = experiment['sensitivities']
        for pi in parameters:
            sp = ss[pi]
            rfs.extend([sp['Gsfs Error']])
            ras.extend([sp['GAS Error']])
        rfs = ras
        m.append(rfs)
    forcol = ['%.2e']*len(parameters)
    t = matrix2latex(m,filename,'tabular',headerRow=hr,
                     headerColumn=hc,caption=caption,formatColumn=forcol)
    return t
```

```
def generateTranSensitivityErrorTableWithRefinement(filename,caption,
                                                    experiments,parameters,
                                                    orders):
    hc = [] # Names of the experiments.
    hr,ha = ['$N$'],[] # hr will contain the names of the sensitivities.
    hr1,hr1a = [''],[]
    st = '$\\left\\{{ \\partial_{{{}}}G\\right\\}}$'
    for pi in parameters:
        hr1 += ['Forward Sensitivity Error']
        hr1a += ['Adjoint Sensitivity Error']
        hr.append(st.format(pi))
        hr.append(st.format(pi))
    hr += ha # Both FS and AS
    hr1 += hr1a
    hr = [hr1,hr]
    m = [] # Contains the entries in the table.
    errs = {}
    for pi in parameters:
        errs.update({pi:{'Gsfs Error':[],'GAS Error':[]}})
    for r in experiments.keys():
        if orders:
            s=r'\multirow{{{}}}{{*}}{{{{'.format(len(experiments[r])+1)
        else:
            s=r'\multirow{{{}}}{{*}}{{{{'.format(len(experiments[r]))
        s = s + r + r' }}'
        hc.append(s)
        #hc.append(r)
        hs = []
        for pi in parameters:
            errs.update({pi:{'Gsfs Error':[],'GAS Error':[]}})
        for experiment in experiments[r]:
            hc += ['']
            rfs = ['{}'.format(experiment['N'])]
            ras = []
            hs += [experiment['h']]
            ss = experiment['sensitivities']
            for pi in parameters:
                sp = ss[pi]
                rfs.extend(['{:.2e}'.format(sp['Gsfs Error'])])
                ras.extend(['{:.2e}'.format(sp['GAS Error'])])
                errs[pi]['Gsfs Error'].append(sp['Gsfs Error'])
                errs[pi]['GAS Error'].append(sp['GAS Error'])
            rfs += ras
            m.append(rfs)
        rfs = ['Order']
        for pi in parameters:
            if 'Gsfs' in orders:
                Gsfso = np.polyfit([log(h) for h in hs],
                                   [log(e) for e in
                                    errs[pi]['Gsfs Error']],1)[0]
                rfs.append('{:.2f}'.format(Gsfso))
            else:
                rfs.append('-')
            if 'GAS' in orders:
                GASo = np.polyfit([log(h) for h in hs],
                                   [log(e) for e in
                                    errs[pi]['GAS Error']],1)[0]
```

```
                rfs.append('{:.2f}'.format(GASo))
            else:
                rfs.append('-')
        m.append(rfs)
    forcol = ['%s'] + ['%s']*2*len(parameters)
    t = matrix2latex(m,filename,'tabular',headerRow=hr,
                     headerColumn=hc,caption=caption,formatColumn=forcol)
    return t

def generateTranFSErrorTableWithRefinement(filename,caption,
                                           experiments,parameters,
                                           orders):
    hc = [] # Names of the experiments.
    hr,ha = ['$N$'],[] # hr will contain the names of the sensitivities.
    hr1,hr1a = [''],[]
    st = '$\\left\\{{ \\partial_{{{}}}G\\right\\}}$'
    for pi in parameters:
        hr1 += ['Forward Sensitivity Error']
        hr1a += ['Adjoint Sensitivity Error']
        hr.append(st.format(pi))
        ha.append(st.format(pi))
    #hr += ha # Both FS and AS
    #hr1 += hr1a
    hr = [hr1,hr]
    m = [] # Contains the entries in the table.
    errs = {}
    for pi in parameters:
        errs.update({pi:{'Gsfs Error':[],'GAS Error':[]}})
    for r in experiments.keys():
        if orders:
            s=r'\multirow{{{}}}{{*}}{{{{'.format(len(experiments[r])+1)
        else:
            s=r'\multirow{{{}}}{{*}}{{{{'.format(len(experiments[r]))
        s = s + r + r' }}'
        hc.append(s)
        #hc.append(r)
        hs = []
        for pi in parameters:
            errs.update({pi:{'Gsfs Error':[],'GAS Error':[]}})
        for experiment in experiments[r]:
            hc += ['']
            rfs = ['{}'.format(experiment['N'])]
            ras = []
            hs += [experiment['h']]
            ss = experiment['sensitivities']
            for pi in parameters:
                sp = ss[pi]
                rfs.extend(['{:.2e}'.format(sp['Gsfs Error'])])
                ras.extend(['{:.2e}'.format(sp['GAS Error'])])
                errs[pi]['Gsfs Error'].append(sp['Gsfs Error'])
                errs[pi]['GAS Error'].append(sp['GAS Error'])
            #rfs += ras
            m.append(rfs)
        hc.pop()
        if orders:
            hc += ['']
            rfs = ['Order']
```

```
            for pi in parameters:
                if 'Gsfs' in orders:
                    Gsfso = np.polyfit([log(h) for h in hs],
                                       [log(e) for e in
                                        errs[pi]['Gsfs Error']],1)[0]
                    rfs.append('{:.2f}'.format(Gsfso))
                else:
                    rfs.append('-')
                '''
                if 'GAS' in orders:
                    GASo = np.polyfit([log(h) for h in hs],
                                       [log(e) for e in
                                        errs[pi]['GAS Error']],1)[0]
                    rfs.append('{:.2f}'.format(GASo))
                else:
                    rfs.append('-')
                '''
            m.append(rfs)
    forcol = ['%s'] + ['%s']*len(parameters)
    t = matrix2latex(m,filename,'tabular',headerRow=hr,
                     headerColumn=hc,caption=caption,formatColumn=forcol)
    return t

def generateTranASErrorTableWithRefinement(filename,caption,
                                                    experiments,parameters,
                                                    orders):
    hc = [] # Names of the experiments.
    hr,ha = ['$N$'],['$N$'] # hr will contain the names of the sensitivities.
    hr1,hr1a = [''],['']
    st = '$\\left\\{{ \\partial_{{{}}}G\\right\\}}$'
    for pi in parameters:
        hr1 += ['Forward Sensitivity Error']
        hr1a += ['Adjoint Sensitivity Error']
        hr.append(st.format(pi))
        ha.append(st.format(pi))
    hr = ha # Both FS and AS
    hr1 = hr1a
    hr = [hr1,hr]
    m = [] # Contains the entries in the table.
    errs = {}
    for pi in parameters:
        errs.update({pi:{'Gsfs Error':[],'GAS Error':[]}})
    for r in experiments.keys():
        if orders:
            s=r'\multirow{{{}}}{{*}}{{{{'.format(len(experiments[r])+1)
        else:
            s=r'\multirow{{{}}}{{*}}{{{{'.format(len(experiments[r]))
        s = s + r + r' }}'
        hc.append(s)
        #hc.append(r)
        hs = []
        for pi in parameters:
            errs.update({pi:{'Gsfs Error':[],'GAS Error':[]}})
        for experiment in experiments[r]:
            hc += ['']
            rfs = ['{}'.format(experiment['N'])]
            ras = ['{}'.format(experiment['N'])]
```

```
            hs += [experiment['h']]
            ss = experiment['sensitivities']
            for pi in parameters:
                sp = ss[pi]
                rfs.extend(['{:.2e}'.format(sp['Gsfs Error'])])
                ras.extend(['{:.2e}'.format(sp['GAS Error'])])
                errs[pi]['Gsfs Error'].append(sp['Gsfs Error'])
                errs[pi]['GAS Error'].append(sp['GAS Error'])
            rfs = ras
            m.append(rfs)
        hc.pop()
        if orders:
            hc += ['']
            rfs = ['Order']
            for pi in parameters:
                '''
                if 'Gsfs' in orders:
                    Gsfso = np.polyfit([log(h) for h in hs],
                                       [log(e) for e in
                                        errs[pi]['Gsfs Error']],1)[0]
                    rfs.append('{:.2f}'.format(Gsfso))
                else:
                    rfs.append('-')
                '''
                if 'GAS' in orders:
                    GASo = np.polyfit([log(h) for h in hs],
                                       [log(e) for e in
                                        errs[pi]['GAS Error']],1)[0]
                    rfs.append('{:.2f}'.format(GASo))
                else:
                    rfs.append('-')
            m.append(rfs)
    forcol = ['%s'] + ['%s']*len(parameters)
    t = matrix2latex(m,filename,'tabular',headerRow=hr,
                     headerColumn=hc,caption=caption,formatColumn=forcol)
    return t

def generateFlowErrorTable(filename,caption,experiments):
    hc = experiments.keys() # Names of the experiments.
    hr = ['$p$', '$u_1$', '$u_2$']
    hr1 = ['Error']*3
    hr = [hr1,hr]
    m = [] # Contains the entries in the table.
    for r in hc:
        rfs = []
        ex = experiments[r]
        rfs.extend([ex['p'][1],ex['ux'][1],ex['uy'][1]])
        m.append(rfs)
    forcol = ['%.2e']*3
    t = matrix2latex(m,filename,'tabular',headerRow=hr,
                     headerColumn=hc,caption=caption,formatColumn=forcol)
    return t

def generateFlowErrorTableWithRefinement(filename,caption,experiments,orders):
    #hc = experiments.keys() # Names of the experiments.
    hc = [] # Names of the experiments.
    hr = ['$N$','$p$', '$u_1$', '$u_2$']
```

```
    hr1 = [''] + ['Error']*3
    hr = [hr1,hr]
    m = [] # Contains the entries in the table.
    for r in experiments.keys():
        if orders:
            s=r'\multirow{{{}}}{{*}}{{{{{'.format(len(experiments[r])+1)
        else:
            s=r'\multirow{{{}}}{{*}}{{{{{'.format(len(experiments[r]))
        s = s + r + r' }}'
        hc.append(s)
        #hc.append(r)
        hs,perrs,uxerrs,uyerrs = [],[],[],[]
        for ex in experiments[r]:
            hc.append('')
            hs += [ex['h']]
            perrs += [ex['p'][1]]
            uxerrs += [ex['ux'][1]]
            uyerrs += [ex['uy'][1]]
            rfs = ['{:d}'.format(ex['N']),'{:.2e}'.format(ex['p'][1]),
                   '{:.2e}'.format(ex['ux'][1]),
                   '{:.2e}'.format(ex['uy'][1],[''])]
            m.append(rfs)
        rfs = ['Order']
        if 'p' in orders:
            pord = np.polyfit([log(h) for h in hs],[log(e) for e in
                                                    perrs],1)[0]
            rfs += ['{:.2f}'.format(pord)]
        else:
            rfs += ['-']
        if 'ux' in orders:
            uxord = np.polyfit([log(h) for h in hs],[log(e) for e in
                                                    uxerrs],1)[0]
            rfs += ['{:.2f}'.format(uxord)]
        else:
            rfs += ['-']
        if 'uy' in orders:
            uyord = np.polyfit([log(h) for h in hs],[log(e) for e in
                                                    uyerrs],1)[0]
            rfs += ['{:.2f}'.format(uyord)]
        else:
            rfs += ['-']
        m.append(rfs)
    forcol = ['%s']*5
    t = matrix2latex(m,filename,'tabular',headerRow=hr,
                     headerColumn=hc,caption=caption,formatColumn=forcol)
    return t

def plotTransportIntegral(experiments,titles,filenames):
    for experimentname in experiments.keys():
        ptitle = titles[experimentname]
        experiment = experiments[experimentname]
        c = experiment['c']
        x,y = experiment['x'],experiment['y']
        #k,b = experiment['k'],experiment['b']
        dx,dy = x[1:]-x[:-1],y[1:]-y[:-1]
        ts = np.array([t[0] for t in c])
        dxdy = np.outer(dx,dy)
```

```
        Gts = np.array([sum((dxdy*t[1]).flat) for t in c])
        plt.clf()
        plt.plot(ts,Gts,'s',color='black')
        plt.ylim(0.,1.1)
        plt.xlim(0.,2.)
        plt.ylabel('Integral of $c$')
        plt.xlabel('$t$')
        #plt.title(ptitle)
        plt.suptitle(ptitle,y=0.99)
        plt.savefig(filenames[experimentname])

def plotTransportIntegralWithRefinement(experiments,titles,filenames):
    markers = ['s','^','v','o','8','*','d','+','x']
    colors = ['black','grey','white']
    for experimentname in experiments.keys():
        ptitle = titles[experimentname]
        experimentlist = experiments[experimentname]
        plt.clf()
        for experiment,marker,colr in zip(experimentlist[::-1],markers,colors):
            c = experiment['c']
            x,y = experiment['x'],experiment['y']
            N = experiment['N']
            dx,dy = x[1:]-x[:-1],y[1:]-y[:-1]
            ts = np.array([t[0] for t in c])
            dxdy = np.outer(dx,dy)
            Gts = np.array([sum((dxdy*t[1]).flat) for t in c])
            #plt.plot(ts,Gts,'s',color='black')
            #plt.plot(ts,Gts,marker,color='black',label='N={}'.format(N))
            plt.plot(ts,Gts,marker,color=colr,label='N={}'.format(N))
        plt.ylim(0.,1.1)
        plt.xlim(0.,2.)
        plt.ylabel('Integral of $c$')
        plt.xlabel('$t$')
        #plt.suptitle(ptitle)
        plt.suptitle(ptitle,y=0.99)
        #plt.title(ptitle)
        plt.legend(loc=4)
        plt.savefig(filenames[experimentname])

def plotTransportSelectedTimes(experiments,titles,filenames):
    times = [0.25,0.75,1.25,1.75]
    for experimentname in experiments.keys():
        ptitle = titles[experimentname]
        experiment = experiments[experimentname]
        c = experiment['c']
        x,y = experiment['x'],experiment['y']
        X,Y = (x[1:]+x[:-1])/2.,(y[1:]+y[:-1])/2.
        ts = np.array([t[0] for t in c])
        plt.clf()
        f,axis = plt.subplots(2,2,sharex='row',sharey='col')
        for i in xrange(2):
            for j in xrange(2):
                tj = np.where(ts <= times[i*2+j])[0][-1]
                t = c[tj][0]
                cj = c[tj][1].T
                axis[i,j].pcolor(x,y,cj,cmap=plt.cm.get_cmap('Greys'),
                                 vmin=0.,vmax=1.)
```

```
                axis[i,j].set_title("$c(x,y,{:1.2f})$".format(t),
                                    fontdict={'fontsize':11})
                axis[i,j].set_aspect('equal','box-forced')
                axis[j,i].tick_params(axis='both',which='major',labelsize=8)
                axis[j,i].tick_params(axis='both',which='minor',labelsize=8)
        plt.setp([a.get_xticklabels() for a in axis[0,:]], visible=False)
        plt.setp([a.get_yticklabels() for a in axis[:,1]], visible=False)
        #plt.suptitle(ptitle)
        plt.suptitle(ptitle,y=0.99)
        plt.savefig(filenames[experimentname])

def plotTransportSelectedTimesWithRefinment(experiments,titles,filenames):
    times = [0.25,0.75,1.25,1.75]
    for experimentname in experiments.keys():
        ptitle = titles[experimentname]
        experiment = experiments[experimentname]
        ne = len(experiment)
        plt.clf()
        f,axis = plt.subplots(ne,4,sharex='row',sharey='col')
        for j in xrange(ne):
            N = experiment[j]['N']
            c = experiment[j]['c']
            ts = np.array([t[0] for t in c])
            x,y = experiment[j]['x'],experiment[j]['y']
            for i in xrange(4):
                tj = np.where(ts <= times[i])[0][-1]
                cj = c[tj][1].T
                t = c[tj][0]
                axis[j,i].pcolor(x,y,cj,cmap=plt.cm.get_cmap('Greys'),
                                 vmin=0.,vmax=1.)
                axis[j,i].set_title("$c(x,y,{:1.2f})$, $N={}$".format(t,N),
                                    fontdict={'fontsize':11})
                axis[j,i].set_aspect('equal','box-forced')
                if j != ne-1:
                    plt.setp(axis[j,i].get_xticklabels(),visible=False)
                if i != 0:
                    plt.setp(axis[j,i].get_yticklabels(),visible=False)
                axis[j,i].tick_params(axis='both',which='major',labelsize=8)
                axis[j,i].tick_params(axis='both',which='minor',labelsize=8)
        #plt.suptitle(ptitle)
        plt.suptitle(ptitle,y=0.99)
        plt.savefig(filenames[experimentname])

def plotPressureVelocity(experiments,titles,filenames):
    for experimentname in experiments.keys():
        ptitle = titles[experimentname]
        experiment = experiments[experimentname]
        p = experiment['p'][0]
        ux,uy = experiment['ux'][0],experiment['uy'][0]
        UX,UY = (ux[1:,:]+ux[:-1,:])/2.,(uy[:,1:]+uy[:,:-1])/2.
        speed = np.sqrt(UX*UX+UY*UY)
        x,y = experiment['x'],experiment['y']
        X,Y = (x[1:]+x[:-1])/2.,(y[1:]+y[:-1])/2.
        NX,NY = len(X),len(Y)
        dens = [NX/30.,NY/30.]
        plt.clf()
        #CS = plt.contour(X,Y,p.T,colors='k')
```

```
        #plt.clabel(CS,fontsize=8,inline=1)
        CS = plt.contourf(X,Y,p.T,cmap='viridis')
        plt.colorbar(CS)
        lw = 2.*speed/speed.max()
        plt.streamplot(X,Y,UX.T,UY.T,density=dens,linewidth=lw,color='k')
        plt.suptitle(ptitle,y=0.99)
        plt.ylim(0.,1.)
        plt.xlim(0.,1.)
        plt.ylabel('$y$')
        plt.xlabel('$x$')
        plt.axis('scaled')
        plt.savefig(filenames[experimentname])

def plotPressureVelocityWithRefinement(experiments,titles,filenames):
    for experimentname in experiments.keys():
        ptitle = titles[experimentname]
        experiment = experiments[experimentname][-1]
        p = experiment['p'][0]
        ux,uy = experiment['ux'][0],experiment['uy'][0]
        UX,UY = (ux[1:,:]+ux[:-1,:])/2.,(uy[:,1:]+uy[:,:-1])/2.
        speed = np.sqrt(UX*UX+UY*UY)
        x,y = experiment['x'],experiment['y']
        X,Y = (x[1:]+x[:-1])/2.,(y[1:]+y[:-1])/2.
        NX,NY = len(X),len(Y)
        dens = [NX/30.,NY/30.]
        plt.clf()
        #CS = plt.contour(X,Y,p.T,colors='k')
        #plt.clabel(CS,fontsize=8,inline=1)
        CS = plt.contourf(X,Y,p.T,cmap='viridis')
        plt.colorbar(CS)
        lw = 2.*speed/speed.max()
        plt.streamplot(X,Y,UX.T,UY.T,density=dens,linewidth=lw,color='k')
        plt.ylim(0.,1.)
        plt.xlim(0.,1.)
        plt.ylabel('$y$')
        plt.xlabel('$x$')
        plt.axis('scaled')
        plt.suptitle(ptitle,y=0.99)
        plt.savefig(filenames[experimentname])

class advectionUpwindC(object):
    def __init__(self,dx,dy,ux,uy,boundary):
        self.Nx,self.Ny = len(dx),len(dy)
        self.dx = dx
        self.dy = dy
        self.boundary = boundary
        self.ux = ux
        self.uy = uy
        self.uxp = ux.copy()
        self.uxp[ux <= 0.] = 0.
        self.uxp[ux > 0.] = 1.
        self.uxn = ux.copy()
        self.uxn[ux > 0.] = 0.
        self.uxn[ux <= 0.] = 1.
        self.uyp = uy.copy()
        self.uyp[uy <= 0.] = 0.
        self.uyp[uy > 0.] = 1.
```

```
        self.uyn = uy.copy()
        self.uyn[uy > 0.] = 0.
        self.uyn[uy <= 0.] = 1.
        self._fx = zeros_like(ux)
        self._fy = zeros_like(uy)

    def Flux(self,c):
        b = self.boundary
        uxp,uxn = self.uxp,self.uxn
        uyp,uyn = self.uyp,self.uyn
        self._fx[1:-1,:] = uxp[1:-1,:]*c[:-1,:] + uxn[1:-1,:]*c[1:,:]
        self._fx[0,b.W.I] = uxp[0,b.W.I]*b.W.Iv
        self._fx[0,b.W.O] = uxn[0,b.W.O]*c[0,b.W.O]
        self._fx[-1,b.E.I] = uxn[-1,b.E.I]*b.E.Iv
        self._fx[-1,b.E.O] = uxp[-1,b.E.O]*c[-1,b.E.O]
        self._fy[:,1:-1] = uyp[:,1:-1]*c[:,:-1] + uyn[:,1:-1]*c[:,1:]
        self._fy[b.S.I,0] = uyp[b.S.I,0]*b.S.Iv
        self._fy[b.S.O,0] = uyn[b.S.O,0]*c[b.S.O,0]
        self._fy[b.N.I,-1] = uyn[b.N.I,-1]*b.N.Iv
        self._fy[b.N.O,-1] = uyp[b.N.O,-1]*c[b.N.O,-1]
        return self._fx,self._fy
```

Listing 5: standardBoundary.py

```
""" Implements some standard boundary types. """

from numpy import arange,zeros,ones,linspace,array
import numpy as np

def WtoEPressure(Nx,Ny,difference):
    ea = np.array([],dtype=np.int)
    bnd = {'Neumann':
           {'West':ea,'East':ea,'North':arange(Nx),'South':arange(Nx)},
           'Dirichlet':
           {'West':arange(Ny),'East':arange(Ny),'North':ea,'South':ea},
           'Values':
           {'West':difference*ones(Ny),'East':zeros(Ny),'North':zeros(Nx),
            'South':zeros(Nx)}}
    return bnd

def SWtoNEpressure(Nx,Ny,nx,ny,difference):
    Wv = difference/2.*ones(Ny)
    Wv[ny:] = 0.
    Ev = zeros(Ny)
    Ev[Ny-ny:] = -difference/2.*ones(ny)
    Sv = difference/2.*ones(Nx)
    Sv[nx:] = 0.
    Nv = zeros(Nx)
    Nv[Nx-nx:] = -difference/2.*ones(nx)
    bnd = {'Neumann':
           {'West':arange(ny,Ny,dtype=np.int),
            'East':arange(Ny-ny,dtype=np.int),
            'North':arange(Nx-nx,dtype=np.int),
            'South':arange(nx,Nx,dtype=np.int)},
           'Dirichlet':
```

```
            {'West':arange(ny,dtype=np.int),
             'East':arange(Ny-ny,Ny,dtype=np.int),
             'North':arange(Nx-nx,Nx,dtype=np.int),
             'South':arange(nx,dtype=np.int)},
            'Values':
            {'West':Wv,'East':Ev,'North':Nv,'South':Sv}}
    return bnd

def SWtoNEflux(Nx,Ny,nx,ny,velocity):
    Wv = velocity*ones(Ny)
    Wv[ny:] = 0.
    Ev = velocity*ones(Ny)
    Ev[:Ny-ny] = 0.
    Sv = velocity*ones(Nx)
    Sv[nx:] = 0.
    Nv = velocity*ones(Nx)
    Nv[:Nx-nx] = 0.
    bnd = {'Neumann':
            {'West':arange(Ny-ny,dtype=np.int),
             'East':arange(ny,Ny,dtype=np.int),
             'North':arange(nx,Nx,dtype=np.int),
             'South':arange(Nx-nx,dtype=np.int)},
            'Dirichlet':
            {'West':arange(Ny-ny,Ny,dtype=np.int),
             'East':arange(ny,dtype=np.int),
             'North':arange(nx,dtype=np.int),
             'South':arange(Nx-nx,Nx,dtype=np.int)},
            'Values':
            {'West':Wv,'East':Ev,'North':Nv,'South':Sv}}
    return bnd

def SWtoNEfluxsmooth(Nx,Ny,nx,ny,velocity):
    linv = velocity*arange(ny+1)/ny
    linv = (linv[1:]+linv[:-1])/2.
    Wv = velocity*ones(Ny)
    Wv[ny:2*ny] = linv
    Wv[2*ny:] = 0.
    Ev = velocity*ones(Ny)
    Ev[Ny-2*ny:Ny-ny] = linv[::-1]
    Ev[:Ny-2*ny] = 0.
    Sv = velocity*ones(Nx)
    Sv[nx:2*nx] = linv
    Sv[2*nx:] = 0.
    Nv = velocity*ones(Nx)
    Nv[Nx-2*nx:Nx-nx] = linv[::-1]
    Nv[:Nx-2*nx] = 0.
    bnd = {'Neumann':
            {'West':arange(Ny-ny,dtype=np.int),
             'East':arange(ny,Ny,dtype=np.int),
             'North':arange(nx,Nx,dtype=np.int),
             'South':arange(Nx-nx,dtype=np.int)},
            'Dirichlet':
            {'West':arange(Ny-ny,Ny,dtype=np.int),
             'East':arange(ny,dtype=np.int),
             'North':arange(nx,dtype=np.int),
             'South':arange(Nx-nx,Nx,dtype=np.int)},
            'Values':
```

```
            {'West':Wv,'East':Ev,'North':Nv,'South':Sv}}
    return bnd
```

Listing 6: experimentWEConstant.py

```
""" W To E Constant Parameters Experiment for Ken Kennedy's Ph.D. Thesis. """

from numpy import linspace,tile,ones,zeros,where,sqrt,zeros_like,meshgrid
from numpy import array,append
from standardBoundary import *
from matrix2latex import matrix2latex
from numpy import log
from experimentDriver import runFlowExperiment,runTransportExperiment
from experimentDriver import generateFlowSensitivityErrorTable
from experimentDriver import generateFlowFSErrorTable
from experimentDriver import generateFlowASErrorTable
from experimentDriver import generateFlowErrorTable
from experimentDriver import generateTranSensitivityErrorTable
from experimentDriver import generateTranFSErrorTable
from experimentDriver import generateTranASErrorTable
from experimentDriver import plotTransportIntegral
from experimentDriver import plotTransportSelectedTimes
from experimentDriver import plotPressureVelocity

def WestToEastConstantParameters(k=1.,b=0.,Nx=10,Ny=10,Tfinal=2.,tol=1e-11,
                                 darcyInitial=True):
    """ Runs an experiment showing west to east flow with constant
    parameters. """
    x = linspace(0,1,Nx+1)
    y = linspace(0,1,Ny+1)
    dx = x[1:]-x[:-1]
    dy = y[1:]-y[:-1]
    ''' Set up the resistance for the non-Darcy system. '''
    def Kappa(aux=None,auy=None):
        if (aux is None) and (auy is None):
            K11 = ones((Nx+1,Ny+1))/k
            K12 = zeros((Nx+1,Ny+1))
            K22 = ones((Nx+1,Ny+1))/k
            K = {'11':K11,'12':K12,'22':K22}
        elif auy is None:
            if aux.shape != (Nx+1,Ny+1):
                raise TypeError("Magnitude of velocity has wrong shape.")
            K11 = ones((Nx+1,Ny+1))/k+b*aux
            K12 = zeros((Nx+1,Ny+1))
            K22 = ones((Nx+1,Ny+1))/k+b*aux
            K = {'11':K11,'12':K12,'22':K22}
        else:
            if aux.shape != (Nx+1,Ny+1) or auy.shape != (Nx+1,Ny+1):
                raise TypeError("Magnitude of velocity has wrong shape.")
            K11 = ones((Nx+1,Ny+1))/k+b*aux
            K12 = zeros((Nx+1,Ny+1))
            K22 = ones((Nx+1,Ny+1))/k+b*auy
            K = {'11':K11,'12':K12,'22':K22}
        return K
    ''' Set up the boundary conditions. '''
```

```
bnd = WtoEPressure(Nx,Ny,1.)
''' Set up the resistance for the sensitivity equation. '''
def sKappa(ux=None,uy=None):
    if (ux is None) and (uy is None):
        K11 = ones((Nx+1,Ny+1))/k
        K12 = zeros((Nx+1,Ny+1))
        K22 = ones((Nx+1,Ny+1))/k
        K = {'11':K11,'12':K12,'22':K22}
    elif uy is None:
        if ux.shape != (Nx+1,Ny+1):
            raise TypeError("Velocity has wrong shape.")
        K11 = ones((Nx+1,Ny+1))/k+2.*b*ux
        K12 = zeros((Nx+1,Ny+1))
        K22 = ones((Nx+1,Ny+1))/k+2.*b*ux
        K = {'11':K11,'12':K12,'22':K22}
    else:
        if ux.shape != (Nx+1,Ny+1) or uy.shape != (Nx+1,Ny+1):
            raise TypeError("Velocity has wrong shape.")
        K11 = ones((Nx+1,Ny+1))/k+2.*b*ux
        K12 = zeros((Nx+1,Ny+1))
        K22 = ones((Nx+1,Ny+1))/k+2.*b*uy
        K = {'11':K11,'12':K12,'22':K22}
    return K
''' Set up the boundary for the sensitivity equation. '''
sbnd = WtoEPressure(Nx,Ny,0.)
''' Set up the gravity-like term for the forward sensitivity to k. '''
def kgrav(ux,uy):
    skgx,skgy = ux/(k**2),uy/(k**2)
    skg = {'1x':skgx,'2y':skgy,'2x':zeros((Nx+1,Ny)),'1y':zeros((Nx,Ny+1))}
    return skg
def dKdk10x(ux,uy):
    return -ux/k/k
def dKdk10y(ux,uy):
    return -uy/k/k
def dKdk01x(ux,uy):
    return -ux/k/k
def dKdk01y(ux,uy):
    return -uy/k/k
K = {'name':'k','gravity':kgrav,
     'dKdp10x':dKdk10x,'dKdp10y':dKdk10y,
     'dKdp01x':dKdk01x,'dKdp01y':dKdk01y}
''' Set up the gravity-like term for the forward sensitivity to b. '''
def bgrav(ux,uy):
    sbgx,sbgy = -ux**2,-uy**2
    sbg = {'1x':sbgx,'2y':sbgy,'2x':zeros((Nx+1,Ny)),'1y':zeros((Nx,Ny+1))}
    return sbg
def dKdb10x(ux,uy):
    return ux*abs(ux)
def dKdb10y(ux,uy):
    return uy*abs(uy)
def dKdb01x(ux,uy):
    return ux*abs(ux)
def dKdb01y(ux,uy):
    return uy*abs(uy)
beta = {'name':'\\beta','gravity':bgrav,
        'dKdp10x':dKdb10x,'dKdp10y':dKdb10y,
        'dKdp01x':dKdb01x,'dKdp01y':dKdb01y}
```

```
    p,ux,uy,KS,sensitivities = runFlowExperiment(x,y,Kappa,sKappa,bnd,
                                                 sbnd,[K,beta],
                                                 darcyInitial)
    c,sensitivities = runTransportExperiment(x,y,ux,uy,sensitivities,
                                             [K,beta],Tfinal,
                                             KS,
                                             sbnd)
    ''' Compute analytic sensitivities. '''
    if b == 0.:
        aux = k
        asbx = -k**3
        askx = 1.
    else:
        aux = (sqrt(1.+4.*b*k**2)-1.)/(2.*b*k)
        asbx = (sqrt(1.+4.*b*k**2)-2.*b*k**2-1)/(2.*b**2*k*sqrt(1.+4.*b*k**2))
        askx = (sqrt(1.+4.*b*k**2)-1.)/(2.*b*k**2*sqrt(1.+4.*b*k**2))
    auy = 0.
    asby = 0.
    asky = 0.
    if Tfinal <= 1./aux:
        asGk = askx*Tfinal/2.
        asGb = asbx*Tfinal/2.
    else:
        asGk = askx/(2.*aux*aux*Tfinal)
        asGb = asbx/(2.*aux*aux*Tfinal)
    X = tile(x,(Ny,1)).T
    X = (X[1:,:] + X[:-1,:])/2.
    ''' Compute errors. '''
    perr = abs(p-(1.-X)).max()
    uxerr = abs(ux-aux).max()
    uyerr = abs(uy-auy).max()
    sensitivities['\\beta'].update(
        {'H10FS Error':abs(sensitivities['\\beta']['H10FS']-asbx),
         'H10AS Error':abs(sensitivities['\\beta']['H10AS']-asbx),
         'H01FS Error':abs(sensitivities['\\beta']['H01FS']-asby),
         'H01AS Error':abs(sensitivities['\\beta']['H01AS']-asby),
         'GAS Error':abs(sensitivities['\\beta']['GAS']-asGb),
         'Gsfs Error':abs(sensitivities['\\beta']['Gsfs']-asGb)})
    sensitivities['k'].update(
        {'H10FS Error':abs(sensitivities['k']['H10FS']-askx),
         'H10AS Error':abs(sensitivities['k']['H10AS']-askx),
         'H01FS Error':abs(sensitivities['k']['H01FS']-asky),
         'H01AS Error':abs(sensitivities['k']['H01AS']-asky),
         'GAS Error':abs(sensitivities['k']['GAS']-asGk),
         'Gsfs Error':abs(sensitivities['k']['Gsfs']-asGk)})
    return (p,perr),(ux,uxerr),(uy,uyerr),c,sensitivities,x,y

def WestToEastConstantParametersDriver(Nx=10,Ny=10,tol=1e-11,Tfinal=2.,
                                       darcyInitial=True):
    """ Runs an experiment showing west to east flow with constant parameters.
    The results are then shown in a table. """
    ks = [1.,1.,2.,1.]
    bs = [0.,1.,1.,2.]
    experiments = {}
    parameters = ['k','\\beta']
    st=r'\begin{{tabular}}{{@{{}}c@{{}}}}$k={}$ \\ $\beta={}$\end{{tabular}}'
    st = r'$\left\{{ \begin{{array}}{{c}} k={}\\'
```

```
    st = st + r' \beta={} \end{{array}}\right\}}$ '
    for k,b in zip(ks,bs):
        p,ux,uy,c,s,x,y = WestToEastConstantParameters(k,b,Nx,Ny,Tfinal,tol,
                                                       darcyInitial)
        expname = st.format(k,b)
        experiments.update({expname:{'p':p,'ux':ux,'uy':uy,'c':c,'x':x,'y':y,
                                     'k':k,'b':b,'sensitivities':s}})
    ''' Generate the table of errors in u and p. '''
    cap='West to East Flow With Contant Parameters: Error in $p$, $u_1$, $u_2$'
    t1 = generateFlowErrorTable('tables/WECpu',cap,experiments)
    ''' Generate the table of errors in the sensitivities. '''
    cap = 'West to East Flow With Contant Parameters: '
    cap = cap + 'Error in Flow Sensitivities'
    t2 = generateFlowSensitivityErrorTable('tables/WECsens',cap,experiments,
                                           parameters)
    cap = 'West to East Flow With Contant Parameters: '
    cap = cap + 'Error in FS Sensitivities'
    t2 = generateFlowFSErrorTable('tables/WECFS',cap,experiments,
                                           parameters)
    cap = 'West to East Flow With Contant Parameters: '
    cap = cap + 'Error in AS Sensitivities'
    t2 = generateFlowASErrorTable('tables/WECAS',cap,experiments,
                                           parameters)
    """ Generate the table of errors in the transport sensitivities. """
    cap = 'West to East Flow With Contant Parameters: '
    cap = cap + 'Error in Transport Sensitivities'
    t3 = generateTranSensitivityErrorTable('tables/WECsenstran',cap,
                                           experiments,parameters)
    cap = 'West to East Flow With Contant Parameters: '
    cap = cap + 'Error in Transport FS Sensitivities'
    t3 = generateTranFSErrorTable('tables/WECFStran',cap,
                                           experiments,parameters)
    cap = 'West to East Flow With Contant Parameters: '
    cap = cap + 'Error in Transport AS Sensitivities'
    t3 = generateTranASErrorTable('tables/WECAStran',cap,
                                           experiments,parameters)
    ptitle = r'West to East Flow With Constant Parameters: $k={},b={}$'
    titles = {}
    filenames = {}
    filenames2 = {}
    filenames3 = {}
    for e in experiments.keys():
        experiment = experiments[e]
        k,b = experiment['k'],experiment['b']
        titles.update({e:ptitle.format(k,b)})
        filenames.update({e:'figures/WECk{}b{}.pdf'.format(k,b)})
        filenames2.update({e:'figures/WECtk{}b{}.pdf'.format(k,b)})
        filenames3.update({e:'figures/WECPVk{}b{}.pdf'.format(k,b)})
    plotPressureVelocity(experiments,titles,filenames3)
    plotTransportIntegral(experiments,titles,filenames)
    plotTransportSelectedTimes(experiments,titles,filenames2)
    return t1,t2,t3

if __name__ == '__main__':
    WestToEastConstantParametersDriver()
```

Listing 7: experimentWESmooth.py

```
""" W To E Smooth Parameters Experiment for Ken Kennedy's Ph.D. Thesis. """

from numpy import linspace,tile,ones,zeros,where,sqrt,zeros_like
from numpy import array,append,log
from standardBoundary import *
from matrix2latex import matrix2latex
from experimentDriver import runFlowExperiment,runTransportExperiment
from experimentDriver import generateFlowSensitivityErrorTable
from experimentDriver import generateFlowSensitivityErrorTableWithRefinement
from experimentDriver import generateFlowFSErrorTableWithRefinement
from experimentDriver import generateFlowASErrorTableWithRefinement
from experimentDriver import generateFlowErrorTable
from experimentDriver import generateFlowErrorTableWithRefinement
from experimentDriver import generateTranSensitivityErrorTableWithRefinement
from experimentDriver import generateTranFSErrorTableWithRefinement
from experimentDriver import generateTranASErrorTableWithRefinement
from experimentDriver import plotTransportIntegralWithRefinement
from experimentDriver import plotTransportSelectedTimesWithRefinment
from experimentDriver import plotPressureVelocityWithRefinement

def WestToEastSmoothParameters(k=1.,b=0.,g=0.,Nx=10,Ny=10,Tfinal=10.,tol=1e-11,
                               darcyInitial=True):
    """ Runs an experiment showing west to east flow with smooth
    parameters. """
    x = linspace(0,1,Nx+1)
    y = linspace(0,1,Ny+1)
    dx = x[1:]-x[:-1]
    dy = y[1:]-y[:-1]
    X = tile(x,(Ny,1)).T
    X2 = tile(x,(Ny+1,1)).T
    XY = tile((x[1:]+x[:-1])/2.,(Ny+1,1)).T
    ''' Set up the resistance for the non-Darcy system. '''
    B = g*X2 + b
    Ks = ones((Nx+1,Ny+1))/k
    z = zeros((Nx+1,Ny+1))
    def Kappa(aux=None,auy=None):
        if (aux is None) and (auy is None):
            K11 = Ks
            K12 = z
            K22 = Ks
            K = {'11':K11,'12':K12,'22':K22}
        elif auy is None:
            K11 = Ks+B*aux
            K12 = z
            K22 = Ks+B*aux
            K = {'11':K11,'12':K12,'22':K22}
        else:
            K11 = Ks+B*aux
            K12 = z
            K22 = Ks+B*auy
            K = {'11':K11,'12':K12,'22':K22}
        return K
    ''' Set up the boundary conditions. '''
    bnd = WtoEPressure(Nx,Ny,1.)
    ''' Set up the resistance for the sensitivity equation. '''
```

```
def sKappa(ux=None,uy=None):
    if (ux is None) and (uy is None):
        K11 = Ks
        K12 = z
        K22 = Ks
        K = {'11':K11,'12':K12,'22':K22}
    elif uy is None:
        K11 = Ks+2.*B*ux
        K12 = z
        K22 = Ks+2.*B*ux
        K = {'11':K11,'12':K12,'22':K22}
    else:
        K11 = Ks+2.*B*ux
        K12 = z
        K22 = Ks+2.*B*uy
        K = {'11':K11,'12':K12,'22':K22}
    return K
''' Set up the boundary for the sensitivity equation. '''
sbnd = WtoEPressure(Nx,Ny,0.)
''' Set up the gravity-like term for the forward sensitivity to k. '''
def kgrav(ux,uy):
    skgx,skgy = ux/(k**2),uy/(k**2)
    skg = {'1x':skgx,'2y':skgy,'2x':zeros((Nx+1,Ny)),'1y':zeros((Nx,Ny+1))}
    return skg
def dKdk10x(ux,uy):
    return -ux/k/k
def dKdk10y(ux,uy):
    return -uy/k/k
def dKdk01x(ux,uy):
    return -ux/k/k
def dKdk01y(ux,uy):
    return -uy/k/k
K = {'name':'k','gravity':kgrav,
     'dKdp10x':dKdk10x,'dKdp10y':dKdk10y,
     'dKdp01x':dKdk01x,'dKdp01y':dKdk01y}
''' Set up the gravity-like term for the forward sensitivity to b. '''
def bgrav(ux,uy):
    sbgx,sbgy = -ux**2,-uy**2
    sbg = {'1x':sbgx,'2y':sbgy,'2x':zeros((Nx+1,Ny)),'1y':zeros((Nx,Ny+1))}
    return sbg
def dKdb10x(ux,uy):
    return ux*abs(ux)
def dKdb10y(ux,uy):
    return uy*abs(uy)
def dKdb01x(ux,uy):
    return ux*abs(ux)
def dKdb01y(ux,uy):
    return uy*abs(uy)
beta = {'name':'\\beta','gravity':bgrav,
        'dKdp10x':dKdb10x,'dKdp10y':dKdb10y,
        'dKdp01x':dKdb01x,'dKdp01y':dKdb01y}
''' Set up the gravity-like term for the forward sensitivity to g. '''
def ggrav(ux,uy):
    sggx,sggy = -ux**2*X,-uy**2*XY
    sgg = {'1x':sggx,'2y':sggy,'2x':zeros((Nx+1,Ny)),'1y':zeros((Nx,Ny+1))}
    return sgg
def dKdg10x(ux,uy):
```

```
    return X*ux*abs(ux)
def dKdg10y(ux,uy):
    return uy*abs(uy)
def dKdg01x(ux,uy):
    return X*ux*abs(ux)
def dKdg01y(ux,uy):
    return uy*abs(uy)
gamma = {'name':'\\gamma','gravity':ggrav,
        'dKdp10x':dKdg10x,'dKdp10y':dKdg10y,
        'dKdp01x':dKdg01x,'dKdp01y':dKdg01y}
p,ux,uy,KS,sensitivities = runFlowExperiment(x,y,Kappa,sKappa,bnd,
                                             sbnd,[K,beta,gamma],
                                             darcyInitial)
c,sensitivities = runTransportExperiment(x,y,ux,uy,sensitivities,
                                         [K,beta,gamma],Tfinal,
                                         KS,sbnd)
''' Compute analytic sensitivities. '''
aux = (sqrt(1.+2.*k**2*(2*b+g))-1.)/(k*(2.*b+g))
sqt = sqrt(4.*b*k*k+2.*g*k*k+1.)
asbx = (2.*sqt-2.*k*k*(2.*b+g)-2.)/(k*(2.*b+g)**2*sqt)
askx = (sqt - 1.)/(k*k*(2.*b+g)*sqt)
asgx = (sqt-k*k*(2.*b+g)-1.)/(k*(2.*b+g)**2*sqt)
auy = 0.
asby = 0.
asky = 0.
asgy = 0.
if Tfinal <= 1./aux:
    asGk = askx*Tfinal/2.
    asGb = asbx*Tfinal/2.
    asGg = asgx*Tfinal/2.
else:
    asGk = askx/(2.*aux*aux*Tfinal)
    asGb = asbx/(2.*aux*aux*Tfinal)
    asGg = asgx/(2.*aux*aux*Tfinal)
X = tile(x,(Ny,1)).T
X = (X[1:,:] + X[:-1,:])/2.
''' Compute errors. '''
ap = -(g/2.)*aux*aux*X**2-(b*aux**2 + aux/k)*X+1.
perr = abs(p-ap).max()
uxerr = abs(ux-aux).max()
uyerr = abs(uy-auy).max()
sensitivities['\\beta'].update(
    {'H10FS Error':abs(sensitivities['\\beta']['H10FS']-asbx),
     'H10AS Error':abs(sensitivities['\\beta']['H10AS']-asbx),
     'H01FS Error':abs(sensitivities['\\beta']['H01FS']-asby),
     'H01AS Error':abs(sensitivities['\\beta']['H01AS']-asby),
     'GAS Error':abs(sensitivities['\\beta']['GAS']-asGb),
     'Gsfs Error':abs(sensitivities['\\beta']['Gsfs']-asGb)})
sensitivities['\\gamma'].update(
    {'H10FS Error':abs(sensitivities['\\gamma']['H10FS']-asgx),
     'H10AS Error':abs(sensitivities['\\gamma']['H10AS']-asgx),
     'H01FS Error':abs(sensitivities['\\gamma']['H01FS']-asgy),
     'H01AS Error':abs(sensitivities['\\gamma']['H01AS']-asgy),
     'GAS Error':abs(sensitivities['\\gamma']['GAS']-asGg),
     'Gsfs Error':abs(sensitivities['\\gamma']['Gsfs']-asGg)})
sensitivities['k'].update(
    {'H10FS Error':abs(sensitivities['k']['H10FS']-askx),
```

```
            'H10AS Error':abs(sensitivities['k']['H10AS']-askx),
            'H01FS Error':abs(sensitivities['k']['H01FS']-asky),
            'H01AS Error':abs(sensitivities['k']['H01AS']-asky),
            'GAS Error':abs(sensitivities['k']['GAS']-asGk),
            'Gsfs Error':abs(sensitivities['k']['Gsfs']-asGk)})
    return (p,perr),(ux,uxerr),(uy,uyerr),c,sensitivities,x,y

def WestToEastSmoothParametersDriver(Nx=10,Ny=10,tol=1e-11,Tfinal=10.,
                                     darcyInitial=True):
    """ Runs an experiment showing west to east flow with constant parameters.
    The results are then shown in a table. """
    ks = [1.,1.,2.,1.]
    bs = [1.,2.,2.,2.]
    gs = [1.,-1.,-1.,5.]
    Ns = [8,16,32]
    experiments = {}
    parameters = ['k','\\beta','\\gamma']
    '''
    st = r'\multirow{{{{{}}}}}{{{{*}}}}{{{{'.format(len(Ns))
    st= st + r'\begin{{tabular}}{{@{{}}r@{{}}}}$k={}$ \\ $\beta={}$ '
    st = st + r'\\ $\gamma={}$ \end{{tabular}}}}'
    st = r'\multirow{{{{{}}}}}{{{{*}}}}{{{{'.format(len(Ns))
    st = st + r'$\left\{{ \begin{{array}}{{c}} k={}\\ \beta={}\\'
    st = st + r' \gamma={} \end{{array}}\right\}}$ }}'
    '''
    st = r'$\left\{{ \begin{{array}}{{c}} k={}\\ \beta={}\\'
    st = st + r' \gamma={} \end{{array}}\right\}}$ '
    sbnd = WtoEPressure(Nx,Ny,0.)
    for k,b,g in zip(ks,bs,gs):
        expname = st.format(k,b,g)
        exs = []
        for N in Ns:
            p,ux,uy,c,s,x,y = WestToEastSmoothParameters(k,b,g,N,N,Tfinal,tol,
                                                         darcyInitial)
            exs.append({'p':p,'ux':ux,'uy':uy,'c':c,'h':x[1]-x[0],
                        'sensitivities':s,'N':N,'k':k,'b':b,'g':g,
                        'x':x,'y':y})
        experiments.update({expname:exs})
    ''' Generate the table of errors in u and p. '''
    cap='West to East Flow With Smooth Parameters: Error in $p$, $u_1$, $u_2$'
    t1 = generateFlowErrorTableWithRefinement('tables/WESpu',cap,experiments,
                                              ['p'])
    ''' Generate the table of errors in the sensitivities. '''
    cap = 'West to East Flow With Smooth Parameters: Error in Sensitivities'
    t2 = generateFlowSensitivityErrorTableWithRefinement('tables/WESsens',cap,
                                                         experiments,
                                                         parameters,
                                                         [])
    cap = 'West to East Flow With Smooth Parameters: Error in FS Sensitivities'
    t2 = generateFlowFSErrorTableWithRefinement('tables/WESFSsens',cap,
                                                experiments,
                                                parameters,
                                                [])
    cap = 'West to East Flow With Smooth Parameters: Error in AS Sensitivities'
    t2 = generateFlowASErrorTableWithRefinement('tables/WESASsens',cap,
                                                experiments,
                                                parameters,
```

```
                                                        [])
    """ Generate the table of errors in the transport sensitivities. """
    cap = 'West to East Flow With Smooth Parameters: '
    cap = cap + 'Error in Transport Sensitivities'
    t3 = generateTranSensitivityErrorTableWithRefinement('tables/WESsenstran',
                                                        cap,
                                                        experiments,
                                                        parameters,
                                                        [])
    cap = 'West to East Flow With Smooth Parameters: '
    cap = cap + 'Error in Transport FS Sensitivities'
    t3 = generateTranFSErrorTableWithRefinement('tables/WESFStran',
                                                        cap,
                                                        experiments,
                                                        parameters,
                                                        ['Gsfs','GAS'])
    cap = 'West to East Flow With Smooth Parameters: '
    cap = cap + 'Error in Transport AS Sensitivities'
    t3 = generateTranASErrorTableWithRefinement('tables/WESAStran',
                                                        cap,
                                                        experiments,
                                                        parameters,
                                                        ['Gsfs','GAS'])
    ptitle = r'West to East Flow With Smooth Parameters: '
    ptitle = ptitle + '$k={},\\beta={},\\gamma={}$'
    "}}"
    titles = {}
    filenames = {}
    filenames2 = {}
    filenames3 = {}
    for e in experiments.keys():
        experiment = experiments[e]
        k,b,g = experiment[0]['k'],experiment[0]['b'],experiment[0]['g']
        titles.update({e:ptitle.format(k,b,g)})
        filenames.update({e:'figures/WESk{}b{}g{}.pdf'.format(k,b,g)})
        filenames2.update({e:'figures/WEStk{}b{}g{}.pdf'.format(k,b,g)})
        filenames3.update({e:'figures/WESPVk{}b{}g{}.pdf'.format(k,b,g)})
    plotTransportIntegralWithRefinement(experiments,titles,filenames)
    plotTransportSelectedTimesWithRefinment(experiments,titles,filenames2)
    plotPressureVelocityWithRefinement(experiments,titles,filenames3)
    return t1,t2,t3

if __name__ == '__main__':
    WestToEastSmoothParametersDriver()
```

Listing 8: experimentWEDiscontinuous.py

```
""" W To E Discont. Parameters Experiment for Ken Kennedy's Ph.D. Thesis. """

from numpy import linspace,tile,ones,zeros,where,sqrt,zeros_like,meshgrid
from numpy import array,append,log
from scipy.interpolate import interp1d
from standardBoundary import *
from myDirac import dirac2D
from matrix2latex import matrix2latex
```

```
from experimentDriver import runFlowExperiment,runTransportExperiment
from experimentDriver import generateFlowSensitivityErrorTable
from experimentDriver import generateFlowSensitivityErrorTableWithRefinement
from experimentDriver import generateFlowFSErrorTableWithRefinement
from experimentDriver import generateFlowASErrorTableWithRefinement
from experimentDriver import generateFlowErrorTable
from experimentDriver import generateFlowErrorTableWithRefinement
from experimentDriver import generateTranSensitivityErrorTableWithRefinement
from experimentDriver import generateTranFSErrorTableWithRefinement
from experimentDriver import generateTranASErrorTableWithRefinement
from experimentDriver import plotTransportIntegralWithRefinement
from experimentDriver import plotTransportSelectedTimesWithRefinment
from experimentDriver import plotPressureVelocityWithRefinement

def WestToEastDiscontinuousParameters(k1=1.,k2=2.,b1=3.,b2=4.,x0=0.5,
                                      Nx=11,Ny=11,Tfinal=10.,
                                      tol=1e-11,darcyInitial=True):
    """ Runs an experiment showing west to east flow with discontinuous
    parameters. """
    x = linspace(0,1,Nx+1)
    y = linspace(0,1,Ny+1)
    dx = x[1:]-x[:-1]
    dy = y[1:]-y[:-1]
    X = tile(x,(Ny,1)).T
    X2 = tile(x,(Ny+1,1)).T
    XY = tile((x[1:]+x[:-1])/2.,(Ny+1,1)).T
    left = where(X2<x0)
    right = where(X2>=x0)
    leftX = where(X<x0)
    rightX = where(X>=x0)
    leftXY = where(XY<x0)
    rightXY = where(XY>=x0)
    ''' Set up the resistance for the non-Darcy system. '''
    Bs = b1*ones((Nx+1,Ny+1))
    Bs[right] = b2
    Ks = ones((Nx+1,Ny+1))/k1
    Ks[right] = 1./k2
    z = zeros((Nx+1,Ny+1))
    oneleft = ones((Nx+1,Ny))
    oneleft[rightX] = 0.
    oneleftY = ones((Nx,Ny+1))
    oneleftY[rightXY] = 0.
    oneright = ones((Nx+1,Ny))
    oneright[leftX] = 0.
    onerightY = ones((Nx,Ny+1))
    onerightY[leftXY] = 0.
    def Kappa(aux=None,auy=None):
        if (aux is None) and (auy is None):
            K11 = Ks
            K12 = z
            K22 = Ks
            K = {'11':K11,'12':K12,'22':K22}
        elif auy is None:
            K11 = Ks+Bs*aux
            K12 = z
            K22 = Ks+Bs*aux
            K = {'11':K11,'12':K12,'22':K22}
```

```
    else:
        K11 = Ks+Bs*aux
        K12 = z
        K22 = Ks+Bs*auy
        K = {'11':K11,'12':K12,'22':K22}
    return K
''' Set up the boundary conditions. '''
bnd = WtoEPressure(Nx,Ny,1.)
''' Set up the resistance for the sensitivity equation. '''
def sKappa(ux=None,uy=None):
    if (ux is None) and (uy is None):
        K11 = Ks
        K12 = z
        K22 = Ks
        K = {'11':K11,'12':K12,'22':K22}
    elif uy is None:
        K11 = Ks+2.*Bs*ux
        K12 = z
        K22 = Ks+2.*Bs*ux
        K = {'11':K11,'12':K12,'22':K22}
    else:
        K11 = Ks+2.*Bs*ux
        K12 = z
        K22 = Ks+2.*Bs*uy
        K = {'11':K11,'12':K12,'22':K22}
    return K
''' Set up the boundary for the sensitivity equation. '''
sbnd = WtoEPressure(Nx,Ny,0.)
''' Set up the gravity-like term for the forward sensitivity to k1. '''
def k1grav(ux,uy):
    sk1gx,sk1gy = ux/(k1**2),uy/(k1**2)
    sk1gx[rightX] = 0.
    sk1gy[rightXY] = 0.
    sk1g = {'1x':sk1gx,'2y':sk1gy,
            '2x':zeros((Nx+1,Ny)),'1y':zeros((Nx,Ny+1))}
    return sk1g
def dKdk110x(ux,uy):
    return -oneleft*ux/k1/k1
def dKdk110y(ux,uy):
    return -oneleftY*uy/k1/k1
def dKdk101x(ux,uy):
    return -oneleft*ux/k1/k1
def dKdk101y(ux,uy):
    return -oneleftY*uy/k1/k1
K1 = {'name':'k_1','gravity':k1grav,
     'dKdp10x':dKdk110x,'dKdp10y':dKdk110y,
     'dKdp01x':dKdk101x,'dKdp01y':dKdk101y}
''' Set up the gravity-like term for the forward sensitivity to k2. '''
def k2grav(ux,uy):
    sk2gx,sk2gy = ux/(k2**2),uy/(k2**2)
    sk2gx[leftX] = 0.
    sk2gy[leftXY] = 0.
    sk2g = {'1x':sk2gx,'2y':sk2gy,
            '2x':zeros((Nx+1,Ny)),'1y':zeros((Nx,Ny+1))}
    return sk2g
def dKdk210x(ux,uy):
    return -oneright*ux/k2/k2
```

```
def dKdk210y(ux,uy):
    return -onerightY*uy/k2/k2
def dKdk201x(ux,uy):
    return -oneright*ux/k2/k2
def dKdk201y(ux,uy):
    return -onerightY*uy/k2/k2
K2 = {'name':'k_2','gravity':k2grav,
     'dKdp10x':dKdk210x,'dKdp10y':dKdk210y,
     'dKdp01x':dKdk201x,'dKdp01y':dKdk201y}
''' Set up the gravity-like term for the forward sensitivity to b1. '''
def b1grav(ux,uy):
    sb1gx,sb1gy = -ux**2,-uy**2
    sb1gx[rightX] = 0.
    sb1gy[rightXY] = 0.
    sb1g = {'1x':sb1gx,'2y':sb1gy,
            '2x':zeros((Nx+1,Ny)),'1y':zeros((Nx,Ny+1))}
    return sb1g
def dKdb110x(ux,uy):
    return oneleft*ux*abs(ux)
def dKdb110y(ux,uy):
    return oneleftY*uy*abs(uy)
def dKdb101x(ux,uy):
    return oneleft*ux*abs(ux)
def dKdb101y(ux,uy):
    return oneleftY*uy*abs(uy)
beta1 = {'name':'\\beta_1','gravity':b1grav,
        'dKdp10x':dKdb110x,'dKdp10y':dKdb110y,
        'dKdp01x':dKdb101x,'dKdp01y':dKdb101y}
''' Set up the gravity-like term for the forward sensitivity to b2. '''
def b2grav(ux,uy):
    sb2gx,sb2gy = -ux**2,-uy**2
    sb2gx[leftX] = 0.
    sb2gy[leftXY] = 0.
    sb2g = {'1x':sb2gx,'2y':sb2gy,
            '2x':zeros((Nx+1,Ny)),'1y':zeros((Nx,Ny+1))}
    return sb2g
def dKdb210x(ux,uy):
    return oneright*ux*abs(ux)
def dKdb210y(ux,uy):
    return onerightY*uy*abs(uy)
def dKdb201x(ux,uy):
    return oneright*ux*abs(ux)
def dKdb201y(ux,uy):
    return onerightY*uy*abs(uy)
beta2 = {'name':'\\beta_2','gravity':b2grav,
        'dKdp10x':dKdb210x,'dKdp10y':dKdb210y,
        'dKdp01x':dKdb201x,'dKdp01y':dKdb201y}
''' Set up the gravity-like term for the forward sensitivity to x0. '''
xc,yc = (x[1:]+x[:-1])/2.,(y[1:]+y[:-1])/2.
hx = x[1]-x[0]
hy = y[1] - y[0]
Xx,Yx = meshgrid(x,yc)
Xy,Yy = meshgrid(xc,y)
Xdy = x0*ones(Ny+1)
Ydy = linspace(0.,1.,Ny+1)
Xd = x0*ones(Ny)
Yd = yc
```

```
x0i = where(x<=x0)[0][-1]
def x0grav(ux,uy):
    uxfun = interp1d(x,ux,axis=0)
    uxx0 = uxfun(x0)
    if x[x0i] == x0 and x0i > 0:
        uyx0 = (uy[x0i,:] + uy[x0i-1,:])/2.
    else:
        uyx0 = uy[x0i,:]
    sx0gx = dirac2D(Xx.T,Yx.T,y[1]-y[0],Xd,Yd,
                    -((uxx0/k1-uxx0/k2) +
                      (b1-b2)*uxx0*abs(uxx0)),hx,hy)
    sx0gy = dirac2D(Xy.T,Yy.T,x[1]-x[0],Xdy,Ydy,
                    -((uyx0/k1-uyx0/k2) +
                      (b1-b2)*uyx0*abs(uyx0)),hx,hy)
    sx0g = {'1x':sx0gx,'2y':sx0gy,
            '2x':zeros((Nx+1,Ny)),'1y':zeros((Nx,Ny+1))}
    return sx0g
def dKdx010x(ux,uy):
    xfunvals = ux/k1 - ux/k2 + (b1-b2)*abs(ux)*ux
    xfun = interp1d(x,xfunvals,axis=0)
    xvals = dirac2D(Xx,Yx,y[1]-y[0],Xd,Yd,xfun(x0),hx,hy)
    return xvals.T
def dKdx010y(ux,uy):
    yfunvals = uy/k1 - uy/k2 + (b1-b2)*abs(uy)*uy
    yfun = interp1d(xc,yfunvals,axis=0)
    yvals = dirac2D(Xy,Yy,x[1]-x[0],Xdy,Ydy,yfun(x0),hx,hy)
    return yvals.T
def dKdx001x(ux,uy):
    xfunvals = ux/k1 - ux/k2 + (b1-b2)*abs(ux)*ux
    xfun = interp1d(x,xfunvals,axis=0)
    xvals = dirac2D(Xx,Yx,y[1]-y[0],Xd,Yd,xfun(x0),hx,hy)
    return xvals.T
def dKdx001y(ux,uy):
    yfunvals = uy/k1 - uy/k2 + (b1-b2)*abs(uy)*uy
    yfun = interp1d(xc,yfunvals,axis=0)
    yvals = dirac2D(Xy,Yy,x[1]-x[0],Xdy,Ydy,yfun(x0),hx,hy)
    return yvals.T
X0 = {'name':'x_0','gravity':x0grav,
        'dKdp10x':dKdx010x,'dKdp10y':dKdx010y,
        'dKdp01x':dKdx001x,'dKdp01y':dKdx001y}
p,ux,uy,KS,sensitivities = runFlowExperiment(x,y,Kappa,sKappa,bnd,
                                             sbnd,[K1,K2,beta1,beta2,X0],
                                             darcyInitial,tol)
c,sensitivities = runTransportExperiment(x,y,ux,uy,sensitivities,
                                         [K1,K2,beta1,beta2,X0],
                                         Tfinal,KS,sbnd)
''' Compute analytic sensitivities. '''
Xerr = (X[1:,:] + X[:-1,:])/2.
sqt = sqrt(((1.-x0)*k1+x0*k2)**2+4.*k1*k1*k2*k2*(b2*(1.-x0)+b1*x0))
aux = (sqt-((1.-x0)*k1+x0*k2))/(2.*k1*k2*(b2*(1.-x0)+b1*x0))
u1 = aux
ap = 1.-(aux/k1+b1*aux*aux)*Xerr
rcoef = u1/k2 + b2*u1*u1
ap[where(Xerr>x0)] = rcoef-rcoef*Xerr[where(Xerr>x0)]
sqt = sqrt(4*k1**2*k2**2*(b1*x0 + b2*(-x0 + 1))+(k1*(-x0 + 1)+k2*x0)**2)
ask1x = ((x0 + (4*k1*k2**2*(b1*x0 + b2*(-x0 + 1)) + (-2*x0 + 2)*(k1*(-x0+1)
             + k2*x0)/2)/sqt - 1)/(2*k1*k2*(b1*x0 + b2*(-x0 + 1)))
```

```
            - (-k1*(-x0 + 1) - k2*x0 + sqt)/(2*k1**2*k2*(b1*x0
            + b2*(-x0 + 1))))
ask2x = ((-x0 + (4*k1**2*k2*(b1*x0 + b2*(-x0 + 1)) + x0*(k1*(-x0 + 1) +
          k2*x0))/sqt)/(2*k1*k2*(b1*x0+b2*(-x0 + 1)))
         - (-k1*(-x0 + 1) - k2*x0 + sqt)/(2*k1*k2**2*(b1*x0+b2*(-x0+1))))
asb1x = (k1*k2*x0/((b1*x0 + b2*(-x0 + 1))*sqt) - x0*(-k1*(-x0 + 1) - k2*x0
         + sqt)/(2*k1*k2*(b1*x0 + b2*(-x0 + 1))**2))
asb2x = (k1*k2*(-x0 + 1)/((b1*x0 + b2*(-x0 + 1))*sqt) + (x0 - 1)*(
         -k1*(-x0 + 1) - k2*x0 + sqt)/(2*k1*k2*(b1*x0 + b2*(-x0+1))**2))
asx0x = ((b2-b1)*(-k1*(1.-x0)-k2*x0 + sqt)/(2*k1*k2*(b1*x0+b2*(1.-x0))**2)+
         (k1 - k2 + (2*k1**2*k2**2*(b1 - b2) + (2.*k2-2.*k1)*(k1*(1.-x0) +
          k2*x0)/2)/sqt)/(2*k1*k2*(b1*x0 + b2*(1.-x0))))
auy = 0.
asb1y = 0.
asb2y = 0.
ask1y = 0.
ask2y = 0.
asx0y = 0.
if Tfinal <= 1./aux:
    asGk1 = ask1x*Tfinal/2.
    asGk2 = ask2x*Tfinal/2.
    asGb1 = asb1x*Tfinal/2.
    asGb2 = asb2x*Tfinal/2.
    asGx0 = asx0x*Tfinal/2.
else:
    asGk1 = ask1x/(2.*aux*aux*Tfinal)
    asGk2 = ask2x/(2.*aux*aux*Tfinal)
    asGb1 = asb1x/(2.*aux*aux*Tfinal)
    asGb2 = asb2x/(2.*aux*aux*Tfinal)
    asGx0 = asx0x/(2.*aux*aux*Tfinal)
X = tile(x,(Ny,1)).T
X = (X[1:,:] + X[:-1,:])/2.
''' Compute errors. '''
perr = abs(p-ap).max()
uxerr = abs(ux-aux).max()
uyerr = abs(uy-auy).max()
sensitivities['\\beta_1'].update(
    {'H10FS Error':abs(sensitivities['\\beta_1']['H10FS']-asb1x),
     'H10AS Error':abs(sensitivities['\\beta_1']['H10AS']-asb1x),
     'H01FS Error':abs(sensitivities['\\beta_1']['H01FS']-asb1y),
     'H01AS Error':abs(sensitivities['\\beta_1']['H01AS']-asb1y),
     'GAS Error':abs(sensitivities['\\beta_1']['GAS']-asGb1),
     'Gsfs Error':abs(sensitivities['\\beta_1']['Gsfs']-asGb1)})
sensitivities['\\beta_2'].update(
    {'H10FS Error':abs(sensitivities['\\beta_2']['H10FS']-asb2x),
     'H10AS Error':abs(sensitivities['\\beta_2']['H10AS']-asb2x),
     'H01FS Error':abs(sensitivities['\\beta_2']['H01FS']-asb2y),
     'H01AS Error':abs(sensitivities['\\beta_2']['H01AS']-asb2y),
     'GAS Error':abs(sensitivities['\\beta_2']['GAS']-asGb2),
     'Gsfs Error':abs(sensitivities['\\beta_2']['Gsfs']-asGb2)})
sensitivities['k_1'].update(
    {'H10FS Error':abs(sensitivities['k_1']['H10FS']-ask1x),
     'H10AS Error':abs(sensitivities['k_1']['H10AS']-ask1x),
     'H01FS Error':abs(sensitivities['k_1']['H01FS']-ask1y),
     'H01AS Error':abs(sensitivities['k_1']['H01AS']-ask1y),
     'GAS Error':abs(sensitivities['k_1']['GAS']-asGk1),
     'Gsfs Error':abs(sensitivities['k_1']['Gsfs']-asGk1)})
```

```
    sensitivities['k_2'].update(
        {'H10FS Error':abs(sensitivities['k_2']['H10FS']-ask2x),
         'H10AS Error':abs(sensitivities['k_2']['H10AS']-ask2x),
         'H01FS Error':abs(sensitivities['k_2']['H01FS']-ask2y),
         'H01AS Error':abs(sensitivities['k_2']['H01AS']-ask2y),
         'GAS Error':abs(sensitivities['k_2']['GAS']-asGk2),
         'Gsfs Error':abs(sensitivities['k_2']['Gsfs']-asGk2)})
    sensitivities['x_0'].update(
        {'H10FS Error':abs(sensitivities['x_0']['H10FS']-asx0x),
         'H10AS Error':abs(sensitivities['x_0']['H10AS']-asx0x),
         'H01FS Error':abs(sensitivities['x_0']['H01FS']-asx0y),
         'H01AS Error':abs(sensitivities['x_0']['H01AS']-asx0y),
         'GAS Error':abs(sensitivities['x_0']['GAS']-asGx0),
         'Gsfs Error':abs(sensitivities['x_0']['Gsfs']-asGx0)})
    return (p,perr),(ux,uxerr),(uy,uyerr),c,sensitivities,x,y

def WestToEastDiscontinuousParametersDriver(Nx=10,Ny=10,tol=1e-11,Tfinal=10.,
                                            darcyInitial=True):
    """ Runs an experiment showing west to east flow with constant parameters.
    The results are then shown in a table. """
    p10,ux10,uy10,c10,s10,x10,y10 = WestToEastDiscontinuousParameters(
                                                                1.,2.,3.,4.,
                                                                0.5,10,10,
                                                                Tfinal,tol,
                                                                darcyInitial)
    p11,ux11,uy11,c11,s11,x11,y11 = WestToEastDiscontinuousParameters(
                                                                1.,2.,3.,4.,
                                                                0.5,11,11,
                                                                Tfinal,tol,
                                                                darcyInitial)
    experiments = {'N=10':{'p':p10,'ux':ux10,'uy':uy10,'c':c10,
                           'h':x10[1]-x10[0],'sensitivities':s10,'N':10},
                   'N=11':{'p':p11,'ux':ux11,'uy':uy11,'c':c11,
                           'h':x11[1]-x11[0],'sensitivities':s11,'N':11}}
    # Make a table of errors.
    cap = "Error with grid aligned and not aligned to the discontinuity."
    generateFlowErrorTable('tables/WEDpu1011',cap,experiments)
    # Run the experiments with grid refinements.
    k1s = [1.,2.]
    k2s = [2.,1.]
    b1s = [1.,.5]
    b2s = [.5,1.]
    x0 = 0.5
    Ns = [5,15,45]
    #Ns = [3,9,27]
    experiments = {}
    parameters = ['k_1','k_2','\\beta_1','\\beta_2']
    st = r'\multirow{{{{{}}}}}{{{{*}}}}{{{{'.format(len(Ns)+1)
    st= st + r'\begin{{tabular}}{{@{{}}c@{{}}}}$k_1={}$ \\ $k_2={}$ '
    st = st + r'\\ $\beta_1={}$ \\ $\beta_2={}$\end{{tabular}}}}}'
    st = r'$\left\{{ \begin{{array}}{{c}} k_1={}, k_2={}\\'
    st = st + r' \beta_1={}, \beta_2={} \end{{array}}\right\}}$ '
    for k1,k2,b1,b2 in zip(k1s,k2s,b1s,b2s):
        expname = st.format(k1,k2,b1,b2)
        exs = []
        for N in Ns:
            sbnd = WtoEPressure(N,N,0.)
```

```
        p,ux,uy,c,s,x,y = WestToEastDiscontinuousParameters(k1,k2,b1,b2,x0,
                                                            N,N,Tfinal,tol,
                                                            darcyInitial)
        exs.append({'p':p,'ux':ux,'uy':uy,'c':c,'h':x[1]-x[0],
                    'k1':k1,'k2':k2,'b1':b1,'b2':b2,'x0':x0,'x':x,'y':y,
                    'sensitivities':s,'N':N})
    experiments.update({expname:exs})
''' Generate the table of errors in u and p. '''
cap='West to East Flow With Smooth Parameters: Error in $p$, $u_1$, $u_2$'
t1 = generateFlowErrorTableWithRefinement('tables/WEDpu',cap,experiments,
                                          [])
''' Generate the table of errors in the sensitivities. '''
cap = 'West to East Flow With Discontinuous Parameters: '
cap = cap + 'Error in Sensitivities'
parameters = ['k_1','k_2','\\beta_1','\\beta_2','x_0']
parameters1 = ['k_1','k_2']
parameters2 = ['\\beta_1','\\beta_2','x_0']
t2 = generateFlowSensitivityErrorTableWithRefinement('tables/WEDsens',cap,
                                                     experiments,
                                                     ['k_1','k_2'],
                                                     [])
t3 = generateFlowSensitivityErrorTableWithRefinement('tables/WEDsens2',cap,
                                                     experiments,
                                                   ['\\beta_1','\\beta_2'],
                                                     [])
t4 = generateFlowSensitivityErrorTableWithRefinement('tables/WEDsens3',cap,
                                                     experiments,
                                                     ['x_0'],
                                                     [])
cap = 'West to East Flow With Discontinuous Parameters: '
cap = cap + 'Error in FS Sensitivities'
t2 = generateFlowFSErrorTableWithRefinement('tables/WEDFSsens',cap,
                                                     experiments,
                                                     parameters,
                                                     [])
cap = 'West to East Flow With Discontinuous Parameters: '
cap = cap + 'Error in FS Sensitivities'
t2 = generateFlowFSErrorTableWithRefinement('tables/WEDFSsens1',cap,
                                                     experiments,
                                                     parameters1,
                                                     [])
cap = 'West to East Flow With Discontinuous Parameters: '
cap = cap + 'Error in FS Sensitivities'
t2 = generateFlowFSErrorTableWithRefinement('tables/WEDFSsens2',cap,
                                                     experiments,
                                                     parameters2,
                                                     [])
cap = 'West to East Flow With Discontinuous Parameters: '
cap = cap + 'Error in AS Sensitivities'
t2 = generateFlowASErrorTableWithRefinement('tables/WEDASsens',cap,
                                                     experiments,
                                                     parameters,
                                                     [])
cap = 'West to East Flow With Discontinuous Parameters: '
cap = cap + 'Error in AS Sensitivities'
t2 = generateFlowASErrorTableWithRefinement('tables/WEDASsens1',cap,
                                                     experiments,
```

```
                                                    parameters1,
                                                    [])
cap = 'West to East Flow With Discontinuous Parameters: '
cap = cap + 'Error in AS Sensitivities'
t2 = generateFlowASErrorTableWithRefinement('tables/WEDASsens2',cap,
                                                    experiments,
                                                    parameters2,
                                                    [])
""" Generate the table of errors in the transport sensitivities. """
cap = 'West to East Flow With Discontinuous Parameters: '
cap = cap + 'Error in Transport Sensitivities'
t5 = generateTranSensitivityErrorTableWithRefinement('tables/WEDtran',
                                                    cap,
                                                    experiments,
                                                    parameters,
                                                    [])
cap = 'West to East Flow With Discontinuous Parameters: '
cap = cap + 'Error in Transport FS Sensitivities'
t5 = generateTranFSErrorTableWithRefinement('tables/WEDFStran',
                                                    cap,
                                                    experiments,
                                                    parameters,
                                                    ['Gsfs','GAS'])
cap = 'West to East Flow With Discontinuous Parameters: '
cap = cap + 'Error in Transport AS Sensitivities'
t5 = generateTranASErrorTableWithRefinement('tables/WEDAStran',
                                                    cap,
                                                    experiments,
                                                    parameters,
                                                    ['Gsfs','GAS'])
ptitle = r'West to East Flow With Discontinuous Parameters: '
ptitle = ptitle + '$k_1={},k_2={},\\beta_1={},\\beta_2={},x_0={}$'
"}}"
titles = {}
filenames = {}
filenames2 = {}
filenames3 = {}
for e in experiments.keys():
    experiment = experiments[e]
    k1,b1,x0 = experiment[0]['k1'],experiment[0]['b1'],experiment[0]['x0']
    k2,b2 = experiment[0]['k2'],experiment[0]['b2']
    titles.update({e:ptitle.format(k1,k2,b1,b2,x0)})
    filenames.update({e:'figures/WEDk1{}k2{}b1{}b2{}x0{}.pdf'.format(k1,k2,
                                                                b1,b2,
                                                                x0)})
    filenames2.update({e:'figures/WEDtk1{}k2{}b1{}b2{}x0{}.pdf'.format(
                                                                k1,k2,
                                                                b1,b2,
                                                                x0)})
    filenames3.update({e:'figures/WEDPVk1{}k2{}b1{}b2{}x0{}.pdf'.format(
                                                                k1,k2,
                                                                b1,b2,
                                                                x0)})
plotTransportIntegralWithRefinement(experiments,titles,filenames)
plotTransportSelectedTimesWithRefinment(experiments,titles,filenames2)
plotPressureVelocityWithRefinement(experiments,titles,filenames3)
return t1,t2,t3,t4,t5
```

```
if __name__ == '__main__':
    WestToEastDiscontinuousParametersDriver()
```

Listing 9: utility.py

```
""" Some utility functions. """

from numpy import maximum,minimum,outer,sum
import numpy as np
from scipy.interpolate import interp1d

def harmonicMean(a,b):
    M = maximum(a,b)
    m = minimum(a,b)
    return 2.*M*(m/(1.+m/M))

def integrateVelocity(x,y,u,axis):
    dx = x[1:] - x[:-1]
    dy = y[1:] - y[:-1]
    if axis == 'x':
        U = (u[1:,:] + u[:-1,:])/2.
    elif axis == 'y':
        U = (u[:,1:] + u[:,:-1])/2.
    U *= outer(dx,dy)
    return sum(U)

def projectPressure(xf,yf,xc,yc,p):
    nx,ny = len(xc)-1,len(yc)-1
    Nx,Ny = len(xf)-1,len(yf)-1
    pf = np.zeros((Nx,Ny))
    for i in xrange(Nx):
        for j in xrange(Ny):
            x,y = (xf[i] + xf[i+1])/2.,(yf[j]+yf[j+1])/2.
            xi = np.where(x>=xc)[0][-1]
            yi = np.where(y>=yc)[0][-1]
            pf[i,j] = p[xi,yi]
    return pf

def projectVelocity(xf,yf,xc,yc,ux,uy):
    nx,ny = len(xc),len(yc)
    Nx,Ny = len(xf),len(yf)
    uxfun = interp1d(xc,ux,axis=0)
    uxf = uxfun(xf)
    uyfun = interp1d(yc,uy)
    uyf = uyfun(yf)
    UX,UY = np.zeros((Nx,Ny-1)),np.zeros((Nx-1,Ny))
    for j in xrange(Ny-1):
        yj = (yf[j]+yf[j+1])/2.
        yi = np.where(yj>=yc)[0][-1]
        UX[:,j] = uxf[:,yi]
    for j in xrange(Nx-1):
        xj = (xf[j]+xf[j+1])/2.
        xi = np.where(xj>=xc)[0][-1]
        UY[j,:] = uyf[xi,:]
```

```
    return UX,UY

def L2DifferenceVelocity(x,y,ux,uy,UX,UY):
    dux = (ux-UX)*(ux-UX)
    eux = np.sqrt(integrateVelocity(x,y,dux,'x'))
    duy = (uy-UY)*(uy-UY)
    euy = np.sqrt(integrateVelocity(x,y,duy,'y'))
    return eux,euy

def L2DifferencePressure(x,y,p,P):
    dp = (p-P)*(p-P)
    dx = x[1:] - x[:-1]
    dy = y[1:] - y[:-1]
    dp *= outer(dx,dy)
    return sum(dp)
```

Listing 10: uzawa.py

```
"""
This module provides two variants of the Uzawa algorithm.

First is the Uzawa algorithm expressed as a gradient based method, and second
is an Uzawa algorithm with conjugate directions (as in conjugate gradient).

The Uzawa algorithm is a method for solving saddle point problems of the form

  Au + B'p = f
  Bu       = g

where B' is the transpose of B. Such problems arise naturally in finding the
minimum of

  J(u) = u'Au/2 - f'u

subject to

  Bu   = g

The algorithm is known to converge for step size a < 2/||BA`Bt|| where A` is
the inverse of A.

"""

from numpy import dot,zeros,abs
import numpy as np
import scipy.sparse.linalg as la

def gradientUzawa(A,B,Bt,f,g,p0=None,tol=1e-8,
                  slvrname='gmres',slvrops=None,verbose=False):
    """
    Implementation of the gradient based Uzawa algorithm.

    Required inputs: A,B,Bt,f,g where we are solving
      Au + Btp = f
      Bu       = g
```

```
    Optional inputs: p0, slvrname, slvrops
      Where p0 is an initial guess for p and the default is 0.0.
      Where tol is the tolerance for the Uzawa algorithm and the default is
      1e-8.
      Where slvrname is the name of the solver from scipy.sparse.linalg. The
      default if gmres.
      Where slvrops is a dictionary of options to pass to the solver. The
      default is to have tol=1e-8.
    Outputs: u,p which solve the equation to the desired tolerance.

    """
    """ Select the solver. """
    slvr = {'gmres':la.gmres,
            'bicg':la.bicg,
            'biststab':la.bicgstab,
            'cg':la.cg,
            'cgs':la.cgs,
            'lgmres':la.lgmres,
            'minres':la.minres,
            'qmr':la.qmr}.get(slvrname,la.gmres)
    """ Default solver options. """
    if slvrops is None:
        slvrops = {'tol':1e-8}
    m,n = B.shape
    if p0 is None:
        """ No initial guess given. """
        p = zeros((m,1))
    else:
        p = p0.copy()
    """ Initial u from p. """
    u,success = slvr(A,f-Bt.dot(p),**slvrops)
    err = tol + 1.0
    """ Iterate until convergence. """
    q = g - B.dot(u.reshape((-1,1)))
    if verbose:
        itcnt = 0
    while (err >= tol):
        l = Bt.dot(q)
        h,success = slvr(A,l,**slvrops)
        if dot(l.reshape(-1),h) != 0.:
            a = dot(q.reshape(-1),q.reshape(-1))/dot(l.reshape(-1),h)
        else:
            print "FAILURE!"
            break
        p = p - a*q
        u = u + a*h
        q = g - B.dot(u.reshape((-1,1)))
        err = abs(q).max()
        if verbose:
            print err
            itcnt += 1
    if verbose:
        print "Required {} iterations.".format(itcnt)
        return u,p,itcnt
    return u,p

def conjugateUzawa(A,B,Bt,f,g,p0=None,tol=1e-8,
```

```
                slvrname='gmres',slvrops=None,verbose=False):
"""
Implementation of the Uzawa algorithm with conjugate directions.

Required inputs: A,B,Bt,f,g where we are solving
  Au + Btp = f
  Bu       = g
Optional inputs: p0, tol
  Where p0 is an initial guess for p and the default is 0.0.
  Where tol is the tolerance for the Uzawa algorithm and the default
  is 1.0e-8.
  Where slvrname is the name of the solver from scipy.sparse.linalg. The
  default if gmres.
  Where slvrops is a dictionary of options to pass to the solver. The
  default is to have tol=1e-8.
Outputs: u,p which solve the equation to the desired tolerance.

"""
""" Select the solver. """
slvr = {'gmres':la.gmres,
        'bicg':la.bicg,
        'biststab':la.bicgstab,
        'cg':la.cg,
        'cgs':la.cgs,
        'lgmres':la.lgmres,
        'minres':la.minres,
        'qmr':la.qmr}.get(slvrname,la.gmres)
""" Default solver options. """
if slvrops is None:
    slvrops = {'tol':1e-8}
m,n = B.shape
if p0 is None:
    """ No initial guess given. """
    p = zeros(m)
else:
    p = p0.copy()
""" Initial u from p. """
u,success = slvr(A,f.reshape(-1)-Bt.dot(p),**slvrops)
q = g.reshape(-1) - B.dot(u)
d = -q.copy()
err = tol + 1.0
""" Iterate until convergence. """
if verbose:
    itcnt = 0
while (err >= tol):
    #l = Bt*d
    l = Bt.dot(d)
    h,success = slvr(A,l,**slvrops)
    a = dot(q,q)/dot(l,h)
    p = p + a*d
    u = u - a*h
    den = dot(q,q)
    #q = g - B*u
    q = g.reshape(-1) - B.dot(u)
    b = dot(q,q)/den
    d = -q + b*d
    err = abs(q).max()
```

```
        if verbose:
            print err
            itcnt += 1
    if verbose:
        print "Required {} iterations.".format(itcnt)
        return u,p,itcnt
    return u,p

def makeABfg(N,k1=1.,k2=1.,h=None):
    if h is None:
        h = 1./(N+1)
    A = np.diag(np.append(k1*np.ones(N*(N+1)),k2*np.ones(N*(N+1))))
    B = np.zeros((N*N,2*(N+1)*N))
    for i in xrange(N):
        for j in xrange(N):
            B[i+j*N,i+j*(N+1)] = -1.
            B[i+j*N,N*(N+1)+i+j*N] = -1.
            B[i+j*N,i+1+j*(N+1)] = 1.
            B[i+j*N,N*(N+1)+i+(j+1)*N] = 1.
    B /= h
    f = -np.ones((N*N,1))
    g = 0.*np.ones((2*N*(N+1),1))
    return A,B,f,g

def run_test():
    N = 10
    A,B,f,g = makeABfg(N)
    print "Running Gradient Uzawa"
    u,p,itcnt = gradientUzawa(A,B,B.T,g,f,verbose=True)
    print "Running Conjugate Uzawa"
    u,p,itcnt = conjugateUzawa(A,B,B.T,g,f,verbose=True)

if __name__ == '__main__':
    run_test()
```

Listing 11: myDirac.py

```
"""
Some things to deal with singular source terms.
Derived following [Engquist].

"""

from numpy import minimum,sum,abs,zeros_like,where,ndarray

def phi1L(xi):
    return minimum(xi+1.,1.-xi)

def dirac1D(x,h):
    if isinstance(x,ndarray):
        ret = zeros_like(x)
        ret[where(abs(x)<=h)] = phi1L(x[where(abs(x)<=h)]/h)/h
        return ret
    else:
        if abs(x) <= h:
```

```
            return phi1L(x/h)/h
        else:
            return 0.

def dirac2D(x,y,dS,X,Y,g,hx,hy):
    """ A 2D Dirac function defined on the curve Gamma.

    x,y is the point to evaluate the Dirac at
    dS is the step size for the points on Gamma
    X,Y are the points on Gamma used in the computation of L
    gS are the strengh of the Dirac at the points in X,Y
    hx,hy are the grid spacings in the x and y directions

    """
    assert len(X) == len(Y)
    assert x.shape == y.shape
    assert len(X) == len(g)
    ret = zeros_like(x)
    for j in xrange(len(X)):
        ret += dirac1D(x-X[j],hx)*dirac1D(y-Y[j],hy)*g[j]
    return dS*ret
```

Listing 12: homotopyExperimentAlgebraic.py

```
"""
Demonstrates continuation on the algebraic example problem.

"""

import numpy as np

def solveForwardEuler(beta,kp,N,saveSteps=False):
    u = -kp
    derivative = getDerivativeFunction(beta)
    ls = np.linspace(0.,1.,N+1)
    if saveSteps:
        steps = []
    for j in xrange(N):
        l = ls[j+1]
        dl = ls[j+1] - ls[j]
        du = derivative(l,u)
        u += dl*du
        if saveSteps:
            steps.append([l,u])
    if saveSteps:
        return steps
    else:
        return u

def solveODE(beta,kp,saveSteps=False,integrator='dopri5'):
    u0 = -kp
    derivative = getDerivativeFunction(beta)
    from scipy.integrate import ode
    r = ode(derivative).set_integrator(integrator)
    if saveSteps:
```

```
        steps = []
        def SaveSteps(t,U):
            steps.append([t,U.copy()])
        r.set_solout(SaveSteps)
    r.set_initial_value(u0,0.)
    u = r.integrate(1.)
    if saveSteps:
        return steps
    else:
        return u

def solveQuasiNewton(beta,kp,N,saveSteps=False):
    u = -kp
    derivative = getDerivativeFunction(beta)
    ls = np.linspace(0.,1.,N+1)
    if saveSteps:
        steps = []
    for j in xrange(N):
        l = ls[j+1]
        dl = ls[j+1] - ls[j]
        du = derivative(l,u)
        u += dl*du
        u = -kp - l*beta*u*np.abs(u)
        if saveSteps:
            steps.append([l,u])
    if saveSteps:
        return steps
    else:
        return u

def getDerivativeFunction(beta):
    def derivativeFunction(l,u):
        return -beta*u*np.abs(u)/(2.*l*beta*np.abs(u)+1.)
    return derivativeFunction

def testSolvers():
    exactu = 0.5
    # Use 10 iterations since that is what the Scipy ODE solver uses.
    stepsFEN = []
    stepsQNN = []
    for j in xrange(10):
        stepsFE = solveForwardEuler(2.,-1.,j+1,True)
        stepsQN = solveQuasiNewton(2.,-1.,j+1,True)
        stepsFEN.append(stepsFE)
        stepsQNN.append(stepsQN)
    stepsODE = solveODE(2.,-1.,True)
    # Plot the Error as a function of N for FE
    errors = [0.5] + [step[-1][1]-0.5 for step in stepsFEN]
    filename = 'figures/homotopy/AlgebraicErrorFE.pdf'
    plt.clf()
    plt.plot(np.arange(0,11),errors,color='black',marker='x')
    plt.suptitle(r'Forward Euler Error in $u$ for $\Delta\lambda =\frac{1}{N}$',
                 y=0.99)
    plt.ylabel('Error')
    plt.xlabel(r'$N$')
    plt.savefig(filename)
    # Plot the Error as a function of N for FE and QN
```

```
    errors = [0.5] + [np.abs(step[-1][1]-0.5) for step in stepsFEN]
    errorsQN = [0.5] + [np.abs(step[-1][1]-0.5) for step in stepsQNN]
    filename = 'figures/homotopy/AlgebraicErrorFEQN.pdf'
    plt.clf()
    plt.plot(np.arange(0,11),errors,color='black',marker='x',label='FE')
    plt.plot(np.arange(0,11),errorsQN,color='black',marker='*',label='QN')
    ti = r'Forward Euler and Quasi Newton Error in $u$ for '
    ti = ti + r'$\Delta\lambda = \frac{1}{N}$'
    plt.suptitle(ti,y=0.99)
    plt.ylabel('Error')
    plt.xlabel(r'$N$')
    plt.legend(loc='upper right')
    plt.savefig(filename)
    # Plot the solution path
    filename = 'figures/homotopy/AlgebraicSolutions.pdf'
    plt.clf()
    step = stepsFEN[-1]
    lsFE = [0] + [s[0] for s in step]
    usFE = [1.] + [s[1] for s in step]
    plt.plot(lsFE,usFE,color='black',marker='x',label='FE')
    stepQN = stepsQNN[-1]
    lsQN = [0] + [s[0] for s in stepQN]
    usQN = [1.] + [s[1] for s in stepQN]
    plt.plot(lsQN,usQN,color='black',marker='*',label='QN')
    lsODE = [s[0] for s in stepsODE]
    usODE = [s[1][0] for s in stepsODE]
    plt.plot(lsODE,usODE,color='black',marker='^',label='ODE')
    plt.axhline(y=0.5,color='k',linestyle='-',label="Exact")
    ti = r'Solution Path for Forward Euler, Quasi Newton, and Scipy ODE Suite'
    plt.suptitle(ti,y=0.99)
    plt.ylabel(r'$u$')
    plt.xlabel(r'$\lambda$')
    plt.legend(loc='upper right')
    plt.savefig(filename)

if __name__ == '__main__':
    import matplotlib
    matplotlib.use('PDF')
    import matplotlib.pyplot as plt
    testSolvers()
```

Listing 13: homotopyExperimentFlow.py

```
"""
Demonstrates using continuation on the flow model.

"""

import matplotlib
matplotlib.use('PDF')
import matplotlib.pyplot as plt
import numpy as np
from flowSolver import DarcySolver
from utility import L2DifferenceVelocity,L2DifferencePressure
```

```
def solveSensitivityForwardEuler(Nx,Ny,N):
    x,y,xc,yc,bnd,F,g,kappa,Kl,Klinv,Knl,dKnl = setupProblem(Nx,Ny)
    slvr = DarcySolver(x,y,bnd,Klinv(),F,g)
    p0,ux0,uy0 = slvr.solve()
    p,ux,uy = p0.copy(),ux0.copy(),uy0.copy()
    pu = np.append(p,np.append(ux,uy))
    derivative = getDerivativeFunction(x,y,bnd)
    ls = np.linspace(0.,1.,N+1)
    for j in xrange(N):
        l = ls[j+1]
        dl = ls[j+1] - ls[j]
        dpu = derivative(l,pu)
        pu += dl*dpu
    p = pu[:Nx*Ny].reshape((Nx,Ny))
    u = pu[Nx*Ny:]
    ux = u[:(Nx+1)*Ny].reshape((Nx+1,Ny))
    uy = u[(Nx+1)*Ny:].reshape((Nx,Ny+1))
    return p,ux,uy,x,y,xc,yc

def solveSensitivityODE(Nx,Ny,integrator='dopri5',
        savesteps=False,verbose=False):
    x,y,xc,yc,bnd,F,g,kappa,Kl,Klinv,Knl,dKnl = setupProblem(Nx,Ny)
    slvr = DarcySolver(x,y,bnd,Klinv(),F,g)
    p0,ux0,uy0 = slvr.solve()
    p,ux,uy = p0.copy(),ux0.copy(),uy0.copy()
    pu = np.append(p,np.append(ux,uy))
    derivative = getDerivativeFunction(x,y,bnd)
    from scipy.integrate import ode
    r = ode(derivative).set_integrator(integrator)
    if verbose and savesteps:
        tpu = []
        def saveSteps(t,pu):
            print t
            p = pu[:Nx*Ny].reshape((Nx,Ny))
            u = pu[Nx*Ny:]
            ux = u[:(Nx+1)*Ny].reshape((Nx+1,Ny))
            uy = u[(Nx+1)*Ny:].reshape((Nx,Ny+1))
            tpu.append([t,p,ux,uy])
        r.set_solout(saveSteps)
    elif verbose:
        def saveSteps(t,pu):
            print t
        r.set_solout(saveSteps)
    elif savesteps:
        tpu = []
        def saveSteps(t,pu):
            p = pu[:Nx*Ny].reshape((Nx,Ny))
            u = pu[Nx*Ny:]
            ux = u[:(Nx+1)*Ny].reshape((Nx+1,Ny))
            uy = u[(Nx+1)*Ny:].reshape((Nx,Ny+1))
            tpu.append([t,p,ux,uy])
        r.set_solout(saveSteps)
    r.set_initial_value(pu,0.)
    pu = r.integrate(1.)
    p = pu[:Nx*Ny].reshape((Nx,Ny))
    u = pu[Nx*Ny:]
    ux = u[:(Nx+1)*Ny].reshape((Nx+1,Ny))
```

```
    uy = u[(Nx+1)*Ny:].reshape((Nx,Ny+1))
    if savesteps:
        return p,ux,uy,tpu,x,y,xc,yc
    return p,ux,uy,x,y,xc,yc

count = 0
def solveSensitivityNewton(Nx,Ny,N,countIterations=False):
    x,y,xc,yc,bnd,F,g,kappa,Kl,Klinv,Knl,dKnl = setupProblem(Nx,Ny)
    ftol = min([1./(Nx*Nx),1./(Ny*Ny),1./N])/2.
    if countIterations:
        global count
        count = 0
        def my_callback(x,f):
            global count
            count += 1
        slvr = DarcySolver(x,y,bnd,Klinv(),F,g,NDSolverOptions={'verbose':True,
                                                                'f_tol':ftol,
                                                          'callback':my_callback})
    else:
        slvr = DarcySolver(x,y,bnd,Klinv(),F,g,NDSolverOptions={'verbose':True,
                                                                'f_tol':ftol})
    p0,ux0,uy0 = slvr.solve()
    p,ux,uy = p0.copy(),ux0.copy(),uy0.copy()
    pu = np.append(p,np.append(ux,uy))
    derivative = getDerivativeFunction(x,y,bnd)
    ls = np.linspace(0.,1.,N+1)
    for j in xrange(N):
        l = ls[j+1]
        dl = ls[j+1] - ls[j]
        dpu = derivative(l,pu)
        pu += dl*dpu
        p = pu[:Nx*Ny].reshape((Nx,Ny))
        u = pu[Nx*Ny:]
        kappal = getkappal(l,x,y)
        p,ux,uy = slvr.solveNonDarcy(kappal,p,ux0=u,DarcyInitial=False)
        pu = np.append(p,np.append(ux,uy))
    if countIterations:
        return p,ux,uy,x,y,xc,yc,count
    else:
        return p,ux,uy,x,y,xc,yc

def solveSensitivityQuasiNewton(Nx,Ny,N,lambdas=None,saveSteps=False):
    x,y,xc,yc,bnd,F,g,kappa,Kl,Klinv,Knl,dKnl = setupProblem(Nx,Ny)
    slvr = DarcySolver(x,y,bnd,Klinv(),F,g)
    p0,ux0,uy0 = slvr.solve()
    if saveSteps:
        ps = [p0.copy()]
        uxs = [ux0.copy()]
        uys = [uy0.copy()]
    p,ux,uy = p0.copy(),ux0.copy(),uy0.copy()
    pu = np.append(p,np.append(ux,uy))
    derivative = getDerivativeFunction(x,y,bnd)
    if lambdas is not None:
        ls = lambdas
    else:
        ls = np.linspace(0.,1.,N+1)
    for j in xrange(N):
```

```
            l = ls[j+1]
            dl = ls[j+1] - ls[j]
            dpu = derivative(l,pu)
            pu += dl*dpu
            p = pu[:Nx*Ny].reshape((Nx,Ny))
            u = pu[Nx*Ny:]
            ux = u[:(Nx+1)*Ny].reshape((Nx+1,Ny))
            uy = u[(Nx+1)*Ny:].reshape((Nx,Ny+1))
            UX,UY = slvr.velocityToCorners(ux,uy)
            kappal = getkappal(l,x,y)
            slvr.kappa = kappal(UX,UY)
            p,ux,uy = slvr.solve(p)
            pu = np.append(p,np.append(ux,uy))
            if saveSteps:
                ps.append(p.copy())
                uxs.append(ux.copy())
                uys.append(uy.copy())
        if saveSteps:
            return ps,uxs,uys,x,y,xc,yc
        else:
            return p,ux,uy,x,y,xc,yc

    def setupProblem(Nx,Ny):
        linspace,meshgrid,arange = np.linspace,np.meshgrid,np.arange
        cos,sin,pi = np.cos,np.sin,np.pi
        zeros = np.zeros
        x,y = linspace(0.,1.,Nx+1),linspace(0.,1.,Ny+1)
        xc,yc = (x[1:]+x[:-1])/2.,(y[1:]+y[:-1])/2.
        xx,yy = np.meshgrid(xc,yc)
        xx,yy = xx.T,yy.T
        ea = np.array([],dtype=np.int)
        '''
        bnd = {'Neumann':
                {'West':arange(Ny),'East':ea,'South':arange(Nx),'North':ea},
                'Dirichlet':
                {'West':ea,'East':arange(Ny),'South':ea,'North':arange(Nx)},
                'Values':
                {'West':zeros(Ny),'East':zeros(Ny),
                    'South':zeros(Nx),'North':zeros(Nx)}}
        '''
        bnd = {'Neumann':
                {'West':ea,'East':ea,'South':ea,'North':ea},
                'Dirichlet':
                {'West':arange(Ny),'East':arange(Ny),
                    'South':arange(Nx),'North':arange(Nx)},
                'Values':
                {'West':zeros(Ny),'East':zeros(Ny),
                    'South':zeros(Nx),'North':zeros(Nx)}}
        F = 4.*pi*yy*cos(4.*pi*xx*yy) + 3.*pi*xx*cos(3.*pi*xx*yy)
        xx,yx = meshgrid(xc,y)
        xx,yx = xx.T,yx.T
        xy,yy = meshgrid(x,yc)
        xy,yy = xy.T,yy.T
        g1y = ((2.+cos(xx*yx)+10.*np.abs(sin(4.*pi*xx*yx)))*sin(4.*pi*xx*yx)
                + 2.*pi*cos(2.*pi*xx)*sin(2.*pi*yx))
        g1x = ((2.+cos(xy*yy)+10.*np.abs(sin(4.*pi*xy*yy)))*sin(4.*pi*xy*yy)
                + 2.*pi*cos(2.*pi*xy)*sin(2.*pi*yy))
```

```
    g2y = ((2.+sin(xx*yx)+15.*np.abs(sin(3.*pi*xx*yx)))*sin(3.*pi*xx*yx)
            + 2.*pi*sin(2.*pi*xx)*cos(2.*pi*yx))
    g2x = ((2.+sin(xy*yy)+15.*np.abs(sin(3.*pi*xy*yy)))*sin(3.*pi*xy*yy)
            + 2.*pi*sin(2.*pi*xy)*cos(2.*pi*yy))
    g = {'1x':g1x,'2y':g2y,'2x':g2x,'1y':g1y}
    kappa,Kl,Klinv,Knl,dKnl = getKappa(x,y)
    return x,y,xc,yc,bnd,F,g,kappa,Kl,Klinv,Knl,dKnl

def getDerivativeFunction(x,y,bnd):
    Nx,Ny = len(x)-1,len(y)-1
    kappa,Kl,Klinv,Knl,dKnl = getKappa(x,y)
    bnd2 = {'Neumann':
            {'West':bnd['Neumann']['West'],
             'East':bnd['Neumann']['East'],
             'South':bnd['Neumann']['South'],
             'North':bnd['Neumann']['North']},
            'Dirichlet':
            {'West':bnd['Dirichlet']['West'],
             'East':bnd['Dirichlet']['East'],
             'South':bnd['Dirichlet']['South'],
             'North':bnd['Dirichlet']['North']},
            'Values':
            {'West':0.*bnd['Values']['West'],
             'East':0.*bnd['Values']['East'],
             'South':0.*bnd['Values']['South'],
             'North':0.*bnd['Values']['North']}}
    slvr = DarcySolver(x,y,bnd2)
    def derivativeFunction(l,pu):
        # pu contains p and ux and uy flattened and appended
        p = pu[:Nx*Ny].reshape((Nx,Ny))
        u = pu[Nx*Ny:]
        ux = u[:(Nx+1)*Ny].reshape((Nx+1,Ny))
        uy = u[(Nx+1)*Ny:].reshape((Nx,Ny+1))
        UX,UY = slvr.velocityToCorners(ux,uy)
        r = getResistance(Kl,Knl,dKnl,l,UX,UY)
        g = getHomotopySource(Knl,UX,UY)
        slvr.kappa = r
        slvr.g = g
        dp,dux,duy = slvr.solve()
        return np.append(dp,np.append(dux,duy))
    return derivativeFunction

def getTrueSolution(x,y,xc,yc):
    xx,yy = np.meshgrid(xc,yc)
    xx,yy = xx.T,yy.T
    p = np.sin(2.*np.pi*xx)*np.sin(2.*np.pi*yy)
    xx,yy = np.meshgrid(x,yc)
    xx,yy = xx.T,yy.T
    ux = np.sin(4.*np.pi*xx*yy)
    xx,yy = np.meshgrid(xc,y)
    xx,yy = xx.T,yy.T
    uy = np.sin(3.*np.pi*xx*yy)
    return p,ux,uy

def getKappa(x,y):
    xx,yy = np.meshgrid(x,y)
    xx,yy = xx.T,yy.T
```

```
z = 0.*xx
def Kl(aux=None,auy=None):
    Kl11 = 2. + np.cos(xx*yy)
    Kl12 = z
    Kl22 = 2. + np.sin(xx*yy)
    K = {'11':Kl11,'12':Kl12,'22':Kl22}
    return K
def Klinv(aux=None,auy=None):
    Kl11 = 1./(2. + np.cos(xx*yy))
    Kl12 = z
    Kl22 = 1./(2. + np.sin(xx*yy))
    K = {'11':Kl11,'12':Kl12,'22':Kl22}
    return K
def Knl(aux=None,auy=None):
    if (aux is None) and (auy is None):
        Knl11 = z
        Knl12 = z
        Knl22 = z
        K = {'11':Knl11,'12':Knl12,'22':Knl22}
    elif (auy is None):
        Knl11 = 10.*np.abs(aux)
        Knl12 = z
        Knl22 = 15.*np.abs(aux)
        K = {'11':Knl11,'12':Knl12,'22':Knl22}
    else:
        Knl11 = 10.*np.abs(aux)
        Knl12 = z
        Knl22 = 15.*np.abs(auy)
        K = {'11':Knl11,'12':Knl12,'22':Knl22}
    return K
def dKnl(aux=None,auy=None):
    if (aux is None) and (auy is None):
        Knl11 = z
        Knl12 = z
        Knl22 = z
        K = {'11':Knl11,'12':Knl12,'22':Knl22}
    elif (auy is None):
        Knl11 = 10.*np.abs(aux)
        Knl12 = z
        Knl22 = 15.*np.abs(aux)
        K = {'11':Knl11,'12':Knl12,'22':Knl22}
    else:
        Knl11 = 10.*np.abs(aux)
        Knl12 = z
        Knl22 = 15.*np.abs(auy)
        K = {'11':Knl11,'12':Knl12,'22':Knl22}
    return K
def kappa(aux=None,auy=None):
    if (aux is None) and (auy is None):
        Knl11 = 2. + np.cos(xx*yy)
        Knl12 = z
        Knl22 = 2. + np.sin(xx*yy)
        K = {'11':Knl11,'12':Knl12,'22':Knl22}
    elif (auy is None):
        Knl11 = 2. + np.cos(xx*yy) + 10.*np.abs(aux)
        Knl12 = z
        Knl22 = 2. + np.sin(xx*yy) + 15.*np.abs(aux)
```

```
            K = {'11':Knl11,'12':Knl12,'22':Knl22}
        else:
            Knl11 = 2. + np.cos(xx*yy) + 10.*np.abs(aux)
            Knl12 = z
            Knl22 = 2. + np.sin(xx*yy) + 15.*np.abs(auy)
            K = {'11':Knl11,'12':Knl12,'22':Knl22}
        return K
    return kappa,Kl,Klinv,Knl,dKnl

def getkappal(l,x,y):
    xx,yy = np.meshgrid(x,y)
    xx,yy = xx.T,yy.T
    z = 0.*xx
    def kappal(aux=None,auy=None):
        if (aux is None) and (auy is None):
            Knl11 = 2. + np.cos(xx*yy)
            Knl12 = z
            Knl22 = 2. + np.sin(xx*yy)
            K = {'11':Knl11,'12':Knl12,'22':Knl22}
        elif (auy is None):
            Knl11 = 2. + np.cos(xx*yy) + l*10.*np.abs(aux)
            Knl12 = z
            Knl22 = 2. + np.sin(xx*yy) + l*15.*np.abs(aux)
            K = {'11':Knl11,'12':Knl12,'22':Knl22}
        else:
            Knl11 = 2. + np.cos(xx*yy) + l*10.*np.abs(aux)
            Knl12 = z
            Knl22 = 2. + np.sin(xx*yy) + l*15.*np.abs(auy)
            K = {'11':Knl11,'12':Knl12,'22':Knl22}
        return K
    return kappal

def getResistance(Kl,Knl,dKnl,l,ux,uy):
    kl,knl,dknl = Kl(),Knl(np.abs(ux),np.abs(uy)),dKnl(np.abs(ux),np.abs(uy))
    K = {'11':kl['11']+l*knl['11']+l*dknl['11'],
         '12':kl['12']+l*knl['12']+l*dknl['12'],
         '22':kl['22']+l*knl['22']+l*dknl['22']}
    return K

def getHomotopySource(knl,ux,uy):
    k = knl(np.abs(ux),np.abs(uy))
    g1 = -(k['11']*ux + k['12']*uy)
    g2 = -(k['12']*ux + k['22']*uy)
    g = {'1x':(g1[:,1:]+g1[:,:-1])/2.,'2x':(g2[:,1:]+g2[:,:-1])/2.,
         '1y':(g1[1:,:]+g1[:-1,:])/2.,'2y':(g2[1:,:]+g2[:-1,:])/2.}
    return g

def testSolvers(Nx,Ny):
    # Generate the true solution and make a plot.
    filename = 'figures/homotopy/solution.pdf'
    x,y = np.linspace(0.,1.,Nx+1),np.linspace(0.,1.,Ny+1)
    xc,yc = (x[1:]+x[:-1])/2.,(y[1:]+y[:-1])/2.
    p,ux,uy = getTrueSolution(x,y,xc,yc)
    UX,UY = (ux[1:,:]+ux[:-1,:])/2.,(uy[:,1:]+uy[:,:-1])/2.
    speed = np.sqrt(UX*UX+UY*UY)
    dens = [Nx/30.,Ny/30.]
    plt.clf()
```

```
#CS = plt.contour(xc,yc,p.T,colors='k')
#plt.clabel(CS,fontsize=8,inline=1)
# CS = plt.contourf(xc,yc,p.T,cmap='viridis')
plt.pcolormesh(xc,yc,p.T,cmap='viridis')
plt.colorbar()
lw = 2.*speed/speed.max()
plt.streamplot(xc,yc,UX.T,UY.T,density=dens,linewidth=lw,color='k')
plt.suptitle('Solution to the Continuation Flow Example',y=0.99)
plt.ylim(0.,1.)
plt.xlim(0.,1.)
plt.ylabel('$y$')
plt.xlabel('$x$')
plt.axis('scaled')
plt.savefig(filename)
# Test each solver for number of iterations to solution.
solvers = ['solveSensitivityForwardEuler','solveSensitivityNewton',
           'solveSensitivityQuasiNewton','solveSensitivityODE']
markers = ['x','o','s','^']
labels = ['Forward Euler','Newton','Quasi-Newton','Scipy ODE']
errors = {}
for solver,marker,label in zip(solvers,markers,labels):
    e = testSolver(Nx,Ny,solver)
    errors.update({solver:{'Error':e,'label':label,'marker':marker}})
# Plot error as a function of number of divisions in lambda. Exclude ODE.
filename = 'figures/homotopy/linear.pdf'
plt.clf()
plt.suptitle(r'Error in $p$ for $\Delta\lambda = \frac{1}{N}$',y=0.99)
for solver in solvers[:-1]:
    err = errors[solver]['Error']
    lab = errors[solver]['label']
    mark = errors[solver]['marker']
    plt.plot([e[0] for e in err],[e[1] for e in err],
             color='black',marker=mark,label=lab)
plt.legend(loc='upper center')
plt.ylabel('Error')
plt.xlabel(r'$N$')
plt.savefig(filename)
filename = 'figures/homotopy/linearu.pdf'
plt.clf()
plt.suptitle(r'Error in $u$ for $\Delta\lambda = \frac{1}{N}$',y=0.99)
for solver in solvers[:-1]:
    err = errors[solver]['Error']
    lab = errors[solver]['label']
    mark = errors[solver]['marker']
    plt.plot([e[0] for e in err],[e[2] for e in err],
             color='black',marker=mark,label=lab)
plt.legend(loc='upper center')
plt.ylabel('Error')
plt.xlabel(r'$N$')
plt.savefig(filename)
# Plot semi-log error. Exclude ODE.
filename = 'figures/homotopy/loglog.pdf'
plt.clf()
plt.suptitle(r'Error in $p$ for $\Delta\lambda = \frac{1}{N}$',y=0.99)
for solver in solvers[:-1]:
    err = errors[solver]['Error']
    lab = errors[solver]['label']
```

```
    mark = errors[solver]['marker']
    '''
    plt.loglog([e[0] for e in err],[e[1] for e in err],
               color='black',marker=mark,label=lab)
    '''
    plt.semilogy([e[0] for e in err],[e[1] for e in err],
               color='black',marker=mark,label=lab)
plt.legend(loc='upper right')
plt.ylabel('Error')
plt.xlabel(r'$N$')
plt.savefig(filename)
filename = 'figures/homotopy/loglogu.pdf'
plt.clf()
plt.suptitle(r'Error in $u$ for $\Delta\lambda = \frac{1}{N}$',y=0.99)
for solver in solvers[:-1]:
    err = errors[solver]['Error']
    lab = errors[solver]['label']
    mark = errors[solver]['marker']
    plt.semilogy([e[0] for e in err],[e[2] for e in err],
               color='black',marker=mark,label=lab)
    '''
    plt.loglog([e[0] for e in err],[e[2] for e in err],
               color='black',marker=mark,label=lab)
    '''
plt.legend(loc='upper right')
plt.ylabel('Error')
plt.xlabel(r'$N$')
plt.savefig(filename)
# Plot number of solves.
filename = 'figures/homotopy/nonlin.pdf'
plt.clf()
plt.suptitle(r'System Solutions for $\Delta\lambda = \frac{1}{N}$',y=0.99)
solver = 'solveSensitivityNewton'
err = errors[solver]['Error']
lab = errors[solver]['label']
mark = errors[solver]['marker']
# Newton takes one solve for the Darcy + one for the predictor +
# the number of iterations for the Newton.
plt.plot([e[0] for e in err],[1+e[-1]+e[0] for e in err],
         color='black',marker=mark,label=lab)
solver = 'solveSensitivityQuasiNewton'
err = errors[solver]['Error']
lab = errors[solver]['label']
mark = errors[solver]['marker']
# Quasi-Newton takes one for the Darcy + one for the predictor +
# one for the corrector.
plt.plot([e[0] for e in err],[1+2*e[0] for e in err],
         color='black',marker=mark,label=lab)
solver = 'solveSensitivityForwardEuler'
err = errors[solver]['Error']
lab = errors[solver]['label']
mark = errors[solver]['marker']
# Forward Euler takes one for the Darcy + one for the predictor
plt.plot([e[0] for e in err],[1+e[0] for e in err],
         color='black',marker=mark,label=lab)
plt.ylabel('System Solutions')
plt.xlabel(r'$N$')
```

```
    plt.ylim(0.,plt.ylim()[1])
    plt.legend(loc='upper left')
    plt.savefig(filename)
    # Generate table to compare RK method.
    solver = 'solveSensitivityODE'
    err = errors[solver]['Error']
    nrk = err[0]
    print solver
    print err
    for solver in solvers[:-1]:
        err = errors[solver]['Error']
        found = False
        for e in reversed(err):
            if e[0] <= nrk:
                print solver
                print e
                found = True
                break
        if not found:
            print solver
            print err[-1]

def testSolver(Nx,Ny,solverName):
    EulerSolvers={'solveSensitivityForwardEuler':solveSensitivityForwardEuler,
           'solveSensitivityQuasiNewton':solveSensitivityQuasiNewton}
    if solverName in EulerSolvers.keys():
        solver = EulerSolvers[solverName]
        maxN = max([Nx*Nx,Ny*Ny])
        nstep = max([Nx,Ny])
        p,ux,uy,x,y,xc,yc = solver(Nx,Ny,1)
        P,UX,UY = getTrueSolution(x,y,xc,yc)
        errors = []
        for j in xrange(1,nstep):
            print "Solving {}: {}/{}".format(solverName,j,maxN)
            # Solve the system with j continuation steps.
            p,ux,uy,x,y,xc,yc = solver(Nx,Ny,j)
            # Compute and store the error in p,ux,uy
            eux,euy = L2DifferenceVelocity(x,y,ux,uy,UX,UY)
            eu = np.sqrt(eux*eux+euy*euy)
            ep = L2DifferencePressure(x,y,p,P)
            errors.append([j,ep,eu])
        '''
        for j in xrange(1,maxN+1,nstep):
            print "Solving {}: {}/{}".format(solverName,j,maxN)
            # Solve the system with j continuation steps.
            p,ux,uy,x,y,xc,yc = solver(Nx,Ny,j)
            # Compute and store the error in p,ux,uy
            eux,euy = L2DifferenceVelocity(x,y,ux,uy,UX,UY)
            eu = np.sqrt(eux*eux+euy*euy)
            ep = L2DifferencePressure(x,y,p,P)
            errors.append([j,ep,eu])
        '''
    elif solverName == 'solveSensitivityNewton':
        solver = solveSensitivityNewton
        maxN = max([Nx*Nx,Ny*Ny])
        nstep = max([Nx,Ny])
        p,ux,uy,x,y,xc,yc = solver(Nx,Ny,1)
```

```
        P,UX,UY = getTrueSolution(x,y,xc,yc)
        errors = []
        for j in xrange(1,nstep,10):
            # Solve the system with j continuation steps.
            print "Solving {}: {}/{}".format(solverName,j,maxN)
            p,ux,uy,x,y,xc,yc,count = solver(Nx,Ny,j,True)
            # Compute and store the error in p,ux,uy
            eux,euy = L2DifferenceVelocity(x,y,ux,uy,UX,UY)
            eu = np.sqrt(eux*eux+euy*euy)
            ep = L2DifferencePressure(x,y,p,P)
            errors.append([j,ep,eu,count])
        '''
        for j in xrange(1,maxN+1,nstep):
            # Solve the system with j continuation steps.
            p,ux,uy,x,y,xc,yc,count = solver(Nx,Ny,j,True)
            # Compute and store the error in p,ux,uy
            eux,euy = L2DifferenceVelocity(x,y,ux,uy,UX,UY)
            eu = np.sqrt(eux*eux+euy*euy)
            ep = L2DifferencePressure(x,y,p,P)
            errors.append([j,ep,eu,count])
        '''
    elif solverName == 'solveSensitivityODE':
        solver = solveSensitivityODE
        p,ux,uy,tpu,x,y,xc,yc = solver(Nx,Ny,savesteps=True)
        P,UX,UY = getTrueSolution(x,y,xc,yc)
        eux,euy = L2DifferenceVelocity(x,y,ux,uy,UX,UY)
        eu = np.sqrt(eux*eux+euy*euy)
        ep = L2DifferencePressure(x,y,p,P)
        errors = [len(tpu),ep,eu]
    else:
        raise NotImplementedError
    return errors

if __name__ == '__main__':
    testSolvers(50,50)
```

Listing 14: adaptivityExperimentFlow.py

```
""" Compute the error in the quantity of interest. """

import numpy as np
import scipy
from homotopyExperimentFlow import solveSensitivityQuasiNewton,setupProblem
from homotopyExperimentFlow import getTrueSolution,getHomotopySource
from homotopyExperimentFlow import getDerivativeFunction,getResistance
from flowSolver import DarcySolver
from utility import integrateVelocity

def uniformLambda(Nx,Ny,N):
    # Solve the flow system for uniformly spaced lambda.
    sens,ps,uxs,uys,x,y,xc,yc = solveSensitivityQuasiNewton(Nx,Ny,N,
                                        saveSteps=True,saveSensitivity=True)
    # Solve the adjoint flow system for each labmda.
    # The boundary is the same for each lambda.
    x,y,xc,yc,bnd,F,g,kappa,Kl,Klinv,Knl,dKnl = setupProblem(Nx,Ny)
```

```
    sbnd,g10,g01 = getSensitivityBoundarySource(Nx,Ny,bnd)
    ux,uy = uxs[0],uys[0]
    # Get adjoint sensitivity.
    H10AS,H01AS = getAdjointSensitivity(0.,x,y,sbnd,g10,g01,ux,uy)
    H10s,H01s = [H10AS],[H01AS]
    # Get forward sensitivity.
    sp = sens[0][:Nx*Ny].reshape((Nx,Ny))
    su = sens[0][Nx*Ny:]
    sux = su[:(Nx+1)*Ny].reshape((Nx+1,Ny))
    suy = su[(Nx+1)*Ny:].reshape((Nx,Ny+1))
    H10FS = integrateVelocity(x,y,sux,'x')
    H01FS = integrateVelocity(x,y,suy,'y')
    H10FSs,H01FSs = [H10FS],[H01FS]
    ls = np.linspace(0.,1.,N+1)
    for j in xrange(N):
        l = ls[j+1]
        ux,uy = uxs[j+1],uys[j+1]
        # Get adjoint sensitivity.
        H10AS,H01AS = getAdjointSensitivity(l,x,y,sbnd,g10,g01,ux,uy)
        H10s.append(H10AS)
        H01s.append(H01AS)
        # Get forward sensitivity.
        sp = sens[j][:Nx*Ny].reshape((Nx,Ny))
        su = sens[j][Nx*Ny:]
        sux = su[:(Nx+1)*Ny].reshape((Nx+1,Ny))
        suy = su[(Nx+1)*Ny:].reshape((Nx,Ny+1))
        H10FS = integrateVelocity(x,y,sux,'x')
        H01FS = integrateVelocity(x,y,suy,'y')
        H10FSs.append(H10FS)
        H01FSs.append(H01FS)
    # Compute the integral in lambda.
    e10 = scipy.integrate.trapz(H10s,ls)
    e01 = scipy.integrate.trapz(H01s,ls)
    e10FS = scipy.integrate.trapz(H10FSs,ls)
    e01FS = scipy.integrate.trapz(H01FSs,ls)
    return e10,e01,e10FS,e01FS

def gaussianLambda(Nx,Ny,N):
    # Solve the flow system for uniformly spaced lambda.
    ls,ws = np.polynomial.legendre.leggauss(N)
    ls = (ls + 1.)/2.
    ls2 = np.append([0.],ls)
    sens,ps,uxs,uys,x,y,xc,yc = solveSensitivityQuasiNewton(Nx,Ny,N,lambdas=ls2,
                                        saveSteps=True,saveSensitivity=True)
    # Solve the adjoint flow system for each labmda.
    x,y,xc,yc,bnd,F,g,kappa,Kl,Klinv,Knl,dKnl = setupProblem(Nx,Ny)
    sbnd,g10,g01 = getSensitivityBoundarySource(Nx,Ny,bnd)
    ux,uy = uxs[0],uys[0]
    H10AS,H01AS = getAdjointSensitivity(0.,x,y,sbnd,g10,g01,ux,uy)
    H10s,H01s = [],[]
    # Get forward sensitivity.
    sp = sens[0][:Nx*Ny].reshape((Nx,Ny))
    su = sens[0][Nx*Ny:]
    sux = su[:(Nx+1)*Ny].reshape((Nx+1,Ny))
    suy = su[(Nx+1)*Ny:].reshape((Nx,Ny+1))
    H10FS = integrateVelocity(x,y,sux,'x')
    H01FS = integrateVelocity(x,y,suy,'y')
```

```
    H10FSs,H01FSs = [],[]
    for j in xrange(N):
        l = ls2[j+1]
        ux,uy = uxs[j+1],uys[j+1]
        # Get adjoint sensitivity.
        H10AS,H01AS = getAdjointSensitivity(l,x,y,sbnd,g10,g01,ux,uy)
        H10s.append(H10AS)
        H01s.append(H01AS)
        H10s.append(H10AS)
        H01s.append(H01AS)
        # Get forward sensitivity.
        sp = sens[j][:Nx*Ny].reshape((Nx,Ny))
        su = sens[j][Nx*Ny:]
        sux = su[:(Nx+1)*Ny].reshape((Nx+1,Ny))
        suy = su[(Nx+1)*Ny:].reshape((Nx,Ny+1))
        H10FS = integrateVelocity(x,y,sux,'x')
        H01FS = integrateVelocity(x,y,suy,'y')
        H10FSs.append(H10FS)
        H01FSs.append(H01FS)
    # Compute the integral in lambda.
    e10 = sum([w*h for w,h in zip(ws,H10s)])/2.
    e01 = sum([w*h for w,h in zip(ws,H01s)])/2.
    e10FS = sum([w*h for w,h in zip(ws,H10FSs)])/2.
    e01FS = sum([w*h for w,h in zip(ws,H01FSs)])/2.
    return e10,e01,e10FS,e01FS

def getSensitivityBoundarySource(Nx,Ny,bnd):
    ones,zeros = np.ones,np.zeros
    # The boundary is the same for each lambda.
    sbnd = {'Neumann':bnd['Neumann'],'Dirichlet':bnd['Dirichlet'],
            'Values':
            {'West':0.*bnd['Values']['West'],
             'East':0.*bnd['Values']['East'],
             'South':0.*bnd['Values']['South'],
             'North':0.*bnd['Values']['North']}
            }
    sbnd['Values']['West'][sbnd['Dirichlet']['West']] = 1.
    sbnd['Values']['East'][sbnd['Dirichlet']['East']] = 1.
    sbnd['Values']['South'][sbnd['Dirichlet']['South']] = 1.
    sbnd['Values']['North'][sbnd['Dirichlet']['North']] = 1.
    # The gravity-like term is the same for each lambda.
    g10 = {'1x':ones((Nx+1,Ny)),'2y':zeros((Nx,Ny+1)),'2x':zeros((Nx+1,Ny)),
           '1y':ones((Nx,Ny+1))}
    g01 = {'1x':zeros((Nx+1,Ny)),'2y':ones((Nx,Ny+1)),'2x':ones((Nx+1,Ny)),
           '1y':zeros((Nx,Ny+1))}
    return sbnd,g10,g01

def getAdjointSensitivity(l,x,y,sbnd,g10,g01,ux,uy):
    # Get the resistance term.
    kappaS = getKappaSensitivity(l,x,y)
    # Solve the adjoint systems.
    ASsolver = DarcySolver(x,y,sbnd,kappaS(),g=g10)
    UX,UY = ASsolver.velocityToCorners(ux,uy)
    ASsolver.kappa = kappaS(UX,UY)
    mu10,vx10,vy10 = ASsolver.solve()
    ASsolver = DarcySolver(x,y,sbnd,kappaS(),g=g01)
    ASsolver.kappa = kappaS(UX,UY)
```

```
    mu01,vx01,vy01 = ASsolver.solve()
    # Compute the sensitivity of H at each lambda.
    dKdl = getKappaSensitivity2(l,x,y)
    H10AS = -(integrateVelocity(x,y,dKdl(ux,uy)['11']*ux*vx10,'x') +
              integrateVelocity(x,y,dKdl(ux,uy)['22']*uy*vy10,'y'))
    H01AS = -(integrateVelocity(x,y,dKdl(ux,uy)['11']*ux*vx01,'x') +
              integrateVelocity(x,y,dKdl(ux,uy)['22']*uy*vy01,'y'))
    return H10AS,H01AS

def getKappaSensitivity(l,x,y):
    xx,yy = np.meshgrid(x,y)
    xx,yy = xx.T,yy.T
    z = 0.*xx
    def kappal(aux=None,auy=None):
        if (aux is None) and (auy is None):
            Knl11 = 2. + np.cos(xx*yy)
            Knl12 = z
            Knl22 = 2. + np.sin(xx*yy)
            K = {'11':Knl11,'12':Knl12,'22':Knl22}
        elif (auy is None):
            Knl11 = 2. + np.cos(xx*yy) + 2.*l*10.*np.abs(aux)
            Knl12 = z
            Knl22 = 2. + np.sin(xx*yy) + 2.*l*15.*np.abs(aux)
            K = {'11':Knl11,'12':Knl12,'22':Knl22}
        else:
            Knl11 = 2. + np.cos(xx*yy) + 2.*l*10.*np.abs(aux)
            Knl12 = z
            Knl22 = 2. + np.sin(xx*yy) + 2.*l*15.*np.abs(auy)
            K = {'11':Knl11,'12':Knl12,'22':Knl22}
        return K
    return kappal

def getKappaSensitivity2(l,x,y):
    xx,yy = np.meshgrid(x,y)
    xx,yy = xx.T,yy.T
    z = 0.*xx
    def kappal(aux=None,auy=None):
        if (aux is None) and (auy is None):
            Knl11 = z
            Knl12 = z
            Knl22 = z
            K = {'11':Knl11,'12':Knl12,'22':Knl22}
        elif (auy is None):
            Knl11 = 10.*np.abs(aux)
            Knl12 = z
            Knl22 = 15.*np.abs(aux)
            K = {'11':Knl11,'12':Knl12,'22':Knl22}
        else:
            Knl11 = 10.*np.abs(aux)
            Knl12 = z
            Knl22 = 15.*np.abs(auy)
            K = {'11':Knl11,'12':Knl12,'22':Knl22}
        return K
    return kappal

def plotErrors(Nx,Ny,N):
    # Get true solution.
```

```
    x,y = np.linspace(0.,1.,Nx+1),np.linspace(0.,1.,Ny+1)
    xc,yc = (x[1:]+x[:-1])/2.,(y[1:]+y[:-1])/2.
    p,ux,uy = getTrueSolution(x,y,xc,yc)
    # Get Darcy solution.
    x,y,xc,yc,bnd,F,g,kappa,Kl,Klinv,Knl,dKnl = setupProblem(Nx,Ny)
    slvr = DarcySolver(x,y,bnd,Klinv(),F,g)
    p0,ux0,uy0 = slvr.solve()
    # Get the difference.
    uxd,uyd = ux-ux0,uy-uy0
    # Compute the "exact" value of the difference in the QOI.
    H10 = integrateVelocity(x,y,uxd,'x')
    H01 = integrateVelocity(x,y,uyd,'y')
    # Up to N, get the approximate value of the QOI using both methods.
    Ns,H10us,H01us,H10gs,H01gs = [],[],[],[],[]
    H10usfs,H01usfs,H10gsfs,H01gsfs = [],[],[],[]
    for j in xrange(1,N):
        exu,eyu,exufs,eyufs = uniformLambda(Nx,Ny,j)
        exg,eyg,exgfs,eygfs = gaussianLambda(Nx,Ny,j)
        Ns.append(j)
        H10us.append(exu)
        H01us.append(eyu)
        H10usfs.append(exufs)
        H01usfs.append(eyufs)
        H10gs.append(exg)
        H01gs.append(eyg)
        H10gsfs.append(exgfs)
        H01gsfs.append(eygfs)
    # Make the plot.
    plt.clf()
    titl = r'Error in Average $x$ Velocity'
    plt.suptitle(titl,y=0.99)
    plt.plot(Ns,H10us,color='k',linestyle='--',label='Uniform (AS)')
    plt.plot(Ns,H10gs,color='k',linestyle='-.',label='Gauss-Legendre (AS)')
    plt.plot(Ns,H10usfs,color='k',linestyle=':',label='Uniform (FS)')
    plt.plot(Ns,H10gsfs,color='k',linestyle='-',label='Gauss-Legendre (FS)')
    plt.axhline(y=H10,color='k',linestyle='-',label="Exact",linewidth=2.0)
    plt.legend(loc='lower right')
    plt.xlabel('Number of Integration Points')
    plt.ylabel(r'Estimated Error')
    plt.savefig('figures/adaptivity/xVelocity.pdf')
    plt.clf()
    titl = r'Error in Average $y$ Velocity'
    plt.suptitle(titl,y=0.99)
    plt.plot(Ns,H01us,color='k',linestyle='--',label='Uniform (AS)')
    plt.plot(Ns,H01gs,color='k',linestyle='-.',label='Gauss-Legendre (AS)')
    plt.plot(Ns,H01usfs,color='k',linestyle=':',label='Uniform (FS)')
    plt.plot(Ns,H01gsfs,color='k',linestyle='-',label='Gauss-Legendre (FS)')
    plt.axhline(y=H01,color='k',linestyle='-',label="Exact",linewidth=2.0)
    plt.legend(loc='lower right')
    plt.xlabel('Number of Integration Points')
    plt.ylabel(r'Estimated Error')
    plt.savefig('figures/adaptivity/yVelocity.pdf')

if __name__ == '__main__':
    Nx,Ny = 50,50
    N = 10
    e10,e01,e10FS,e01FS = uniformLambda(Nx,Ny,N)
```

```
import matplotlib
matplotlib.use('PDF')
import matplotlib.pyplot as plt
plotErrors(Nx,Ny,N)
```