University of Montana

ScholarWorks at University of Montana

Graduate Student Theses, Dissertations, & Professional Papers

Graduate School

2013

An Investigation of Viscosity Using Measured Velocities on Helheim Glacier

Glen David Granzow
The University of Montana

Follow this and additional works at: https://scholarworks.umt.edu/etd

Let us know how access to this document benefits you.

Recommended Citation

Granzow, Glen David, "An Investigation of Viscosity Using Measured Velocities on Helheim Glacier" (2013). *Graduate Student Theses, Dissertations, & Professional Papers*. 4184.
https://scholarworks.umt.edu/etd/4184

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.

AN INVESTIGATION OF VISCOSITY USING MEASURED VELOCITIES ON HELHEIM GLACIER

By

Glen David Granzow

Bachelor of Science; Mechanical Engineering, University of New Mexico,
Albuquerque, New Mexico, 1980
Master of Science; Mechanical Engineering, University of New Mexico,
Albuquerque, New Mexico, 1982
Doctor of Philosophy; Applied Mathematics, Northwestern University,
Evanston, Illinois, 1997

Thesis

presented in partial fulfillment of the requirements
for the degree of

Master of Science
in Computer Science

The University of Montana
Missoula, Montana

December 2013

Approved by:

Sandy Ross, Dean of the Graduate School
Graduate School

Jesse V. Johnson, Chair
Computer Science

Douglas W. Raiford
Computer Science

Joel Harper
Geosciences

Granzow, Glen, M.S., Fall 2013 Computer Science

An Investigation of Viscosity Using Measured Velocities on Helheim Glacier

Chairperson: Jesse V. Johnson

Using recent measurements of the velocity of ice on the surface of Helheim Glacier, located on the south-east coast of Greenland, the variation of viscosity, a material property of the ice, is investigated. The investigation focuses on determining the value of a parameter in Glen's flow law, a constitutive equation widely used in numerical models of ice sheets. Three different approaches are used: (1) Finite differences are used in a direct approach, (2) A finite element model is used in an approach involving an inverse problem, and (3) Nonlinear least squares fits of the velocity are used. The parameter focused on, the exponent in Glen's flow law, is assigned a value of three in most ice sheet models but the results presented in this thesis suggest that the behavior of ice (at least that in the fast moving portion of Helheim Glacier) might be more accurately modeled using a significantly smaller value, 1.6.

Table of Contents

Abstract **ii**

Table of Contents **iii**

List of Figures **v**

Chapter 1: Introduction **1**

1.1 Background 1
1.2 An Overview of the Thesis 2

Chapter 2: Evaluating the Exponent in Glen's Flow Law **5**

2.1 Introduction 5
2.2 Velocity Data 6
2.3 A Mathematical Model 9
2.4 Glen's Flow Law 12
2.5 Concluding Remarks 15

Chapter 3: A Forward Problem **19**

3.1 Introduction 19
3.2 A Boundary Value Problem 19
3.3 The Finite Element Method 21
3.4 The Solution of the Forward Problem 23

Chapter 4: An Inverse Problem **25**

4.1 Introduction 25
4.2 The Inverse Problem 25
4.3 A Solution of the Inverse Problem 27
4.4 Tikhonov Regularization 31

4.5 Evaluating the Exponent in Glen's Flow Law 36

Chapter 5: Using Least-Squares Fits to Evaluate the Exponent in Glen's Flow Law **39**
5.1 Introduction 39
5.2 Solution of the Differential Equation 40
5.3 Least Squares Fits 42
5.4 Results 43

Chapter 6: Conclusion **49**
6.1 Summary 49
6.2 Other Glaciers 50

Appendix A: Adjoint Methods **53**
A.1 Introduction 53
A.2 A Typical Situation 55
A.3 A Numerical Example 61
A.4 A Data Assimilation Example From Glaciology 66

Appendix B: Source Code **71**
B.1 helheim.py 71
B.2 forward1D.py 78
B.3 inverse1D.py 81
B.4 curvefit.py 87
B.5 adjoint.py 92
B.6 ismiphomD.py 100

Bibliography **109**

List of Figures

1.1 A map of Greenland showing the location of Helheim Glacier 3

2.1 Surface velocity data for Helheim Glacier 7
2.2 Surface velocity along the transect of Helheim Glacier 8
2.3 Viscosity ratio for the region between the vertical dashed lines in figure 2.2 11
2.4 The viscosity ratio $\frac{\mu}{\mu_m}$ plotted versus $\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|$ on a log-log scale 14

3.1 Approximate solution of the forward problem 24

4.1 Velocity obtained when solving the inverse problem 29
4.2 Viscosity ratio obtained by solving the inverse problem 30
4.3 Poor approximation of measured velocity obtained when solving the regularized inverse problem with a large value for γ 32
4.4 Velocity obtained when solving the regularized inverse problem 34
4.5 Viscosity ratio obtained by solving the regularized inverse problem . . 35
4.6 The viscosity ratio $\frac{\mu}{\mu_m}$ plotted versus $\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|$ on a log-log scale 37
4.7 A least squares fit of a second order polynomial to the 21 data points around the location of maximum velocity 38

5.1 Least squares fits for Glen's flow exponents $n = 0.5$, $n = 1$, $n = 1.6$, and $n = 3$ 45
5.2 Error in the least squares fits, $\overline{E}$, plotted versus n 46

A.1 Curves for the problem presented in Example 2 60
A.2 Solution to the boundary value problem $u'' - 2u' + u = 1 + x - 5x^2$ for $0 < x < 1$, and $u(0) = 0$, $u(1) = 0$ 63
A.3 Solution of an inverse problem 68

Chapter 1

Introduction

Background information and an overview of the thesis are presented.

1.1 Background

Glacial ice currently covers about 10% of the earth's land. (Cuffey and Patterson [1] are the source for this and other information in this introductory chapter.) Most of this ice (more than 99% [2]) is in the ice sheets of Greenland and Antarctica, the rest in alpine environments. Geologic evidence suggests that during the earth's "ice ages" much larger ice sheets existed.

Scientists have long been interested in the evolution of glaciers and ice sheets. Recent evidence of global climate change has increased this interest by creating concerns about the contribution that diminishing glacial ice volumes may make to rising sea levels [3]. The growth and attenuation of glaciers and ice sheets involves more than just freezing and melting; deformation of the ice, basal sliding, and calving (the breaking away of ice from a glacier's terminus) can be dominant processes.

Use of mathematical models is a key part of scientists' current efforts to understand the deformation of glacial ice. Field observations and laboratory experiments indicate that in many respects ice acts like a (very viscous) liquid. Thus the equations of fluid mechanics are used to model its behavior. (See, for example, Greve and Blatter [4].) A key parameter in these equations is viscosity, a property of the fluid of interest. Viscosity essentially describes the relationship between the rate at which the fluid deforms (the technical term for this is the *strain rate*) in response to applied forces (or more accurately, *stresses*). Some fluids' behavior is accurately described using a single

number (*i.e.* a constant) for the viscosity but the behavior of ice is generally thought to be more accurately modeled using a viscosity reflecting a nonlinear relationship between strain rate and stress.

The most commonly used constituency equation describing glacial ice's nonlinear relationship between strain rate and stress is *Glen's Flow Law* (or, more precisely, "Nye's generalization of Glen's flow law" [5]) which can be expressed using an appropriate viscosity. But Glen's flow law itself involves parameters whose values are not well established.

Historically, data about glaciers and ice sheets was relatively scarce because collecting information was difficult due to their remote locations and formidable environments. In recent years data has become more plentiful as remote sensing techniques using air-borne and satellite mounted devices have been developed and implemented.

1.2 An Overview of the Thesis

In this thesis, surface velocities extracted from measurements made using Synthetic Aperture Radar (SAR) [6] on board the TerraSAR-X satellite are used to investigate the viscosity of ice in Helheim Glacier, located in south-eastern Greenland (see figure 1.1). The thesis contains chapters devoted to the following topics:

- **Chapter 2:** Using measured velocities and a mathematical model approximated using finite differences, the variation in viscosity along a transect of Helheim Glacier is investigated and one of the parameters in Glen's flow law is evaluated.

- **Chapter 3:** The finite element method is used to solve for velocities along the transect of Helheim Glacier using the viscosity variations found in Chapter 2. This reproduces the measured velocities, verifying the finite element model and that the viscosity variations can account for the velocities seen, at least if the mathematical model is applicable.

- **Chapter 4:** An inverse problem is solved (using the finite element model from chapter 3) to investigate the variation in viscosity along the transect of Helheim Glacier. This might be expected to produce the same viscosity variations found in Chapter 2 but the actual results differ in significant ways.

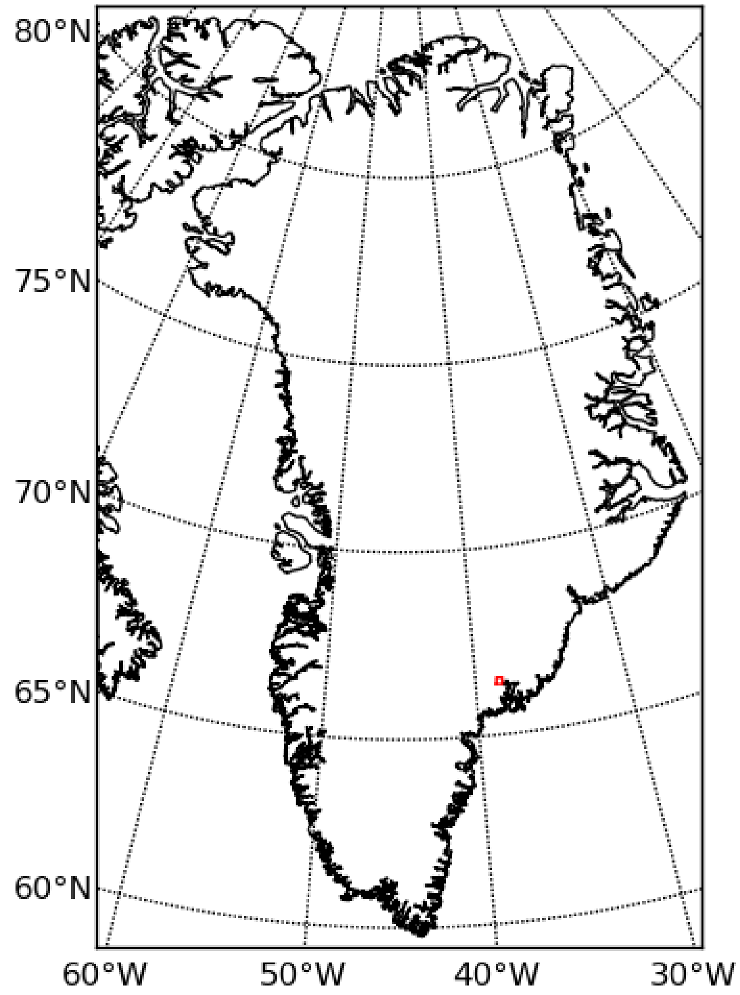

Figure 1.1: A map of Greenland. The small red rectangle indicates the location of Helheim Glacier.

- **Chapter 5:** Nonlinear least squares fits of the velocity along the transect of Helheim Glacier are used to estimate the parameter of interest in Glen's flow law. The value obtained is very close to that obtained in Chapter 2.

- **Chapter 6:** Results are summarized in the final chapter. A brief description of the results of applying the methods described in the thesis to Greenland outlet glaciers other than Helheim is included.

Solution of the inverse problem in Chapter 4 is accomplished by writing it as an optimization problem. The method used to solve the optimization problem requires calculating derivatives of an objective function with respect to parameters in the model. An *adjoint method* was used to find these derivatives. Appendix A, explaining adjoint methods, is included at the end of the thesis.

Computations necessary for the analysis presented in this thesis were performed by scripts written by its author in the *Python* programming language. All figures were created using the *matplotlib.pyplot* package. The map in figure 1.1 was created with the *mpl_toolkits.basemap* package. Finite element modeling used the *FEniCS dolfin* package [7, 8] and solution of the inverse problem utilized the *scipy.optimize* and *dolfin-adjoint* [9] packages. The *scipy.optimize* module was also used while finding the nonlinear least squares fits. Source code for scripts that perform most of the calculations described in the thesis are provided in Appendix B.

Chapter 2

Evaluating the Exponent in Glen's Flow Law Using Surface Velocities on Helheim Glacier

A method is presented for evaluating the exponent in Glen's flow law using measured surface velocities on a transect of a fast moving ice stream. Applying the method to Helheim Glacier (in southeastern Greenland) gives a value of 1.6, lower than the commonly used value 3.

2.1 Introduction

Glen's flow law is a constitutive relation for ice, widely used in numerical models of ice sheets, relating the components of the strain rate tensor, $\dot{\varepsilon}_{ij}$, to the components of the deviatoric stress tensor, τ_{ij} [1, 4, 5].

$$\dot{\varepsilon}_{ij} = A\tau^{n-1}\tau_{ij} \tag{2.1}$$

Here A is a temperature dependent parameter and τ is the second invariant of the deviatoric stress tensor. The exponent, n, in Glen's flow law has been measured in laboratories; the value most widely used in ice sheet models is three although measured values have deviated significantly from this, ranging from 1.5 to 4.2 [10]. In this paper, measurements of surface velocity are used to evaluate the exponent in Glen's flow law. Thus the exponent is obtained for actual glacial ice whose physical properties may differ from the ice used in laboratory experiments due to crystal orientations and impurities.

The relationship between the strain rate and the stress in a fluid are commonly described using viscosity, a material property of the fluid. In a simple Newtonian fluid, where the viscosity is constant, strain rate is proportional to stress. This corresponds to a constant A and $n = 1$ in equation 2.1. The nonlinear relationship between strain rate and stress described by equation 2.1 with $n \neq 1$ can be described using a viscosity that varies with the effective strain rate (*i.e.* the second invariant of the strain rate tensor). In this paper, the variation in the viscosity of ice along a transect of Helmeim Glacier is deduced from measured surface velocities. From these same surface velocities the effective strain rate is obtained, so the relationship between viscosity and effective strain rate can be examined. This examination indicates that Glen's flow law is applicable between the shear margins of Helheim Glacier and allows the exponent in Glen's flow law to be evaluated.

The mathematical model used is based on the Stokes equations. Flow in the model varies in only one direction, transverse to the flow, and temperature variations are neglected.

2.2 Velocity Data

Figure 2.1 shows surface velocity data for Helheim Glacier obtained from the National Snow and Ice Data Center [11]. A transect, indicated by the dashed red line in figure 2.1, was selected. This transect was judged to approximately satisfy two of the assumptions made in the analysis presented in this paper; the glacier is flowing in roughly a straight line with little variation of velocity in the direction of flow. The magnitude of the surface velocity along the transect is shown in figure 2.2.

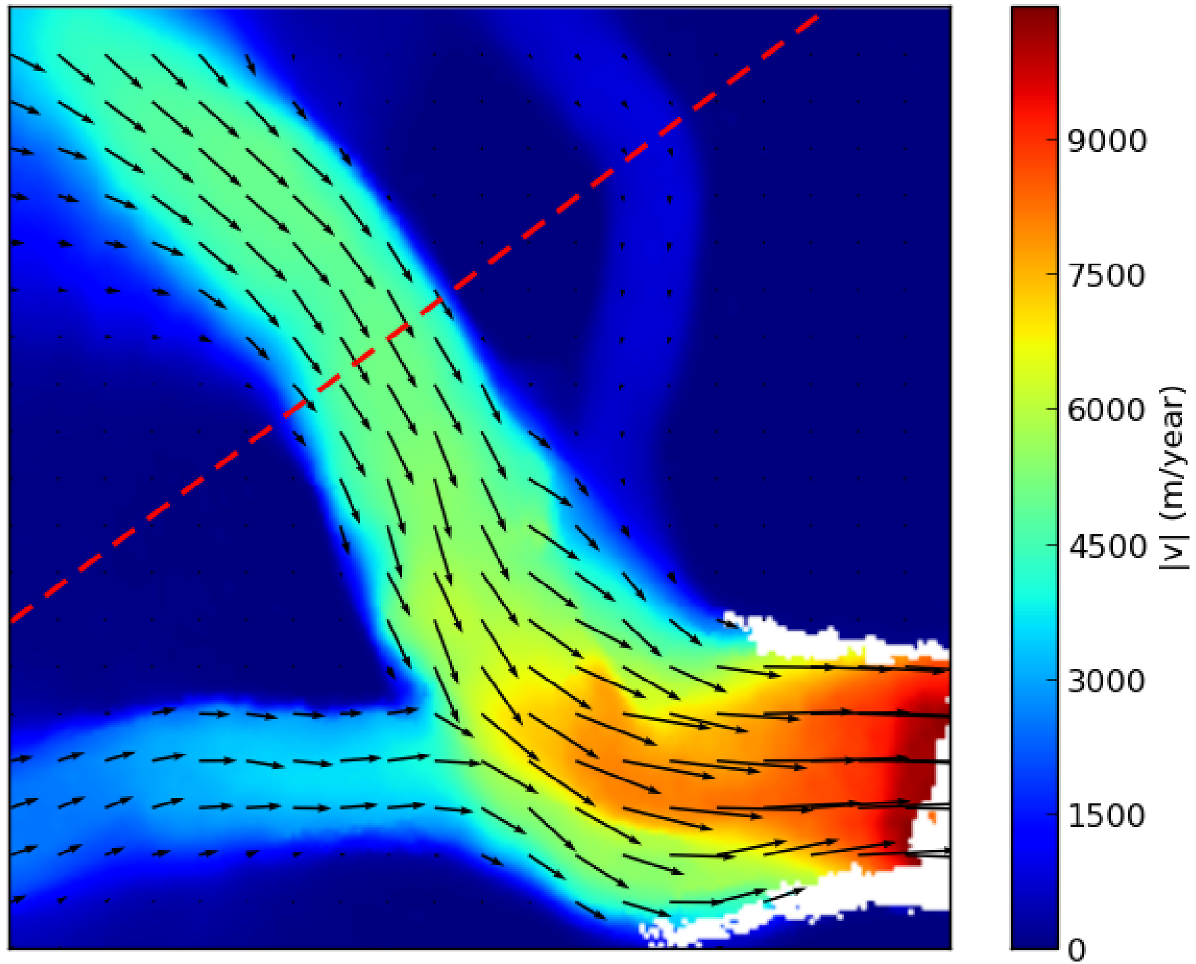

Figure 2.1: Surface velocity data for Helheim Glacier for the period from 2 September 2010 to 13 September 2010. The color scale gives the magnitude of the velocity in meters/annum. Arrows indicate the direction of the velocity as well as its magnitude. The dashed red line shows the transect used for this paper. The figure represents a 20 km by 20 km area. Grid spacing of the data is 100 meters.

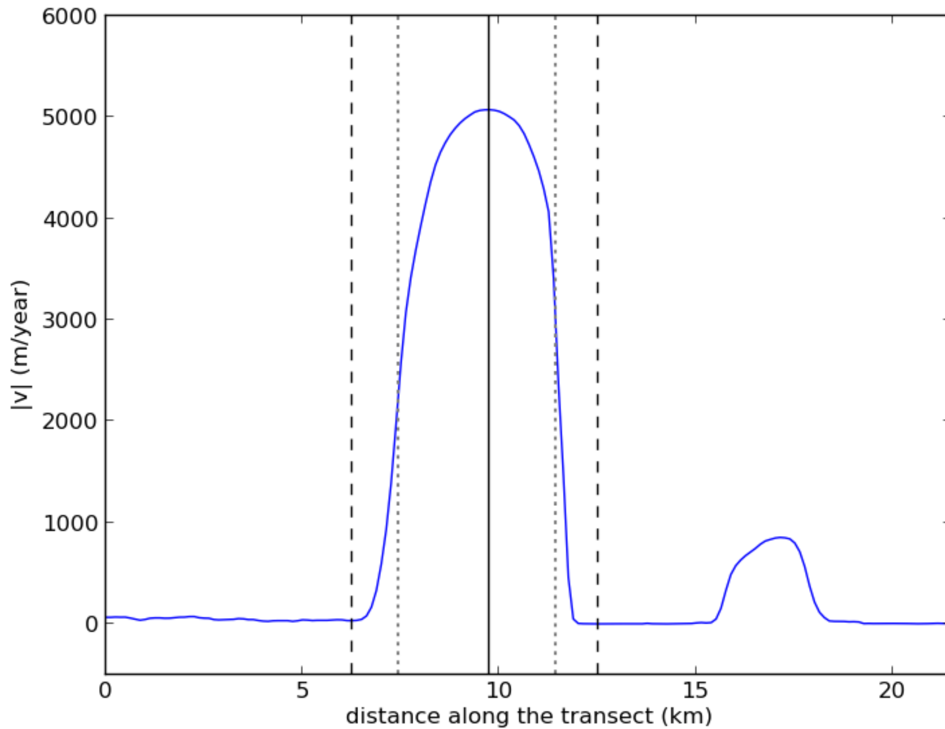

Figure 2.2: Surface velocity (in meters/annum) along the transect of Helheim Glacier shown as a dashed red line in figure 2.1. Only the region between the vertical dashed lines was used in the analysis presented in this paper. The vertical dotted lines are at the "shear margins", defined here as the locations of the minima of viscosity (as plotted in figure 2.3). These minima occur near the inflection points of the velocity curve. The solid vertical line is at x_m, the location of the maximum velocity.

2.3 A Mathematical Model

Stokes equations represent Newton's second law ($\Sigma F = ma$) for a fluid when the inertial terms (ma) are negligible. There are three equations, one for each of the coordinate directions x, y, and z. The equation representing the sum of forces in the y direction is

$$\frac{\partial}{\partial x}\left[\mu\left(\frac{\partial u}{\partial y}+\frac{\partial v}{\partial x}\right)\right]+\frac{\partial}{\partial y}\left[-p+2\mu\frac{\partial v}{\partial y}\right]+\frac{\partial}{\partial z}\left[\mu\left(\frac{\partial w}{\partial y}+\frac{\partial v}{\partial z}\right)\right]=-\rho g_y \tag{2.2}$$

where μ is the viscosity, u, v, and w are the components of fluid velocity in the x, y, and z directions respectively, p is pressure, ρ is the density of the fluid, and g_y is the component of gravity in the y direction.

In a three dimensional "full Stokes" ice model, the two additional Stokes equations (for the x and z directions) would also be considered, and an equation representing conservation of mass would be added to make four equations in the four unknowns u, v, w, and p. Such a model would typically align the z-axis vertically so $g_y = 0$.

In the present treatment, we consider a fast moving stream of ice flowing in a straight line, in the direction opposite to the surface gradient. The z-axis is perpendicular to the upper surface of the ice, with the y-axis pointing in the direction of flow. The x direction is horizontal, transverse to the flow. Here $g_y = g\alpha$ where $\alpha = \sin\left(\tan^{-1}|s|\right)$, s being the surface slope. We assume that there is no variation in the direction of flow so all derivatives with respect to y in equation 2.2 are zero. We also assume that derivatives in the z direction are zero, an assumption believed to be reasonable for regions where basal friction is low. Under these assumptions equation 2.2 simplifies to

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[\mu\frac{\mathrm{d}v}{\mathrm{d}x}\right]=-\rho g\alpha \tag{2.3}$$

where the derivatives have now been written as ordinary derivatives since variation is only in the x direction.

We will apply equation 2.3 to the portion of the transect between the dashed vertical lines in figure 2.2. Assuming the surface slope (and thus α) is constant along this portion of the transect, both sides of equation 2.3 can be integrated, with the value of the constant of integration selected such that the resulting equation is

satisfied at x_m the location of maximum velocity (where $\frac{\mathrm{d}v}{\mathrm{d}x} = 0$). This gives

$$\mu \frac{\mathrm{d}v}{\mathrm{d}x} = \rho g \alpha (x_m - x) \tag{2.4}$$

The region to which equation 2.4 is to be applied has been chosen such that the location of the maximum velocity, x_m, is the *only* place where $\frac{\mathrm{d}v}{\mathrm{d}x} = 0$. At every point other than x_m equation 2.4 can be solved for the viscosity

$$\mu = \frac{\rho g \alpha (x_m - x)}{\frac{\mathrm{d}v}{\mathrm{d}x}} \tag{2.5}$$

Let μ_m denote the viscosity of the ice at x_m. While equation 2.5 cannot be applied at x_m (since that would involve dividing by zero) the viscosity *is* continuous at x_m so, applying L'Hôpital's rule

$$\mu_m = \lim_{x \to x_m} \mu = \lim_{x \to x_m} \frac{\rho g \alpha (x_m - x)}{\frac{\mathrm{d}v}{\mathrm{d}x}} = \lim_{x \to x_m} \frac{-\rho g \alpha}{\left(\frac{\mathrm{d}^2 v}{\mathrm{d}x^2}\right)} = -\frac{\rho g \alpha}{\left.\frac{\mathrm{d}^2 v}{\mathrm{d}x^2}\right|_{x_m}} \tag{2.6}$$

Dividing the corresponding sides of equations 2.5 and 2.6 gives an expression for the ratio of the viscosity at a point x to the viscosity at x_m

$$\frac{\mu}{\mu_m} = \frac{\left.\frac{\mathrm{d}^2 v}{\mathrm{d}x^2}\right|_{x_m} (x - x_m)}{\frac{\mathrm{d}v}{\mathrm{d}x}} \tag{2.7}$$

Using the velocity data plotted in figure 2.2, finite differences were used to approximate the derivatives in equation 2.7 and the resulting calculated viscosity ratios along the transect are shown in figure 2.3.

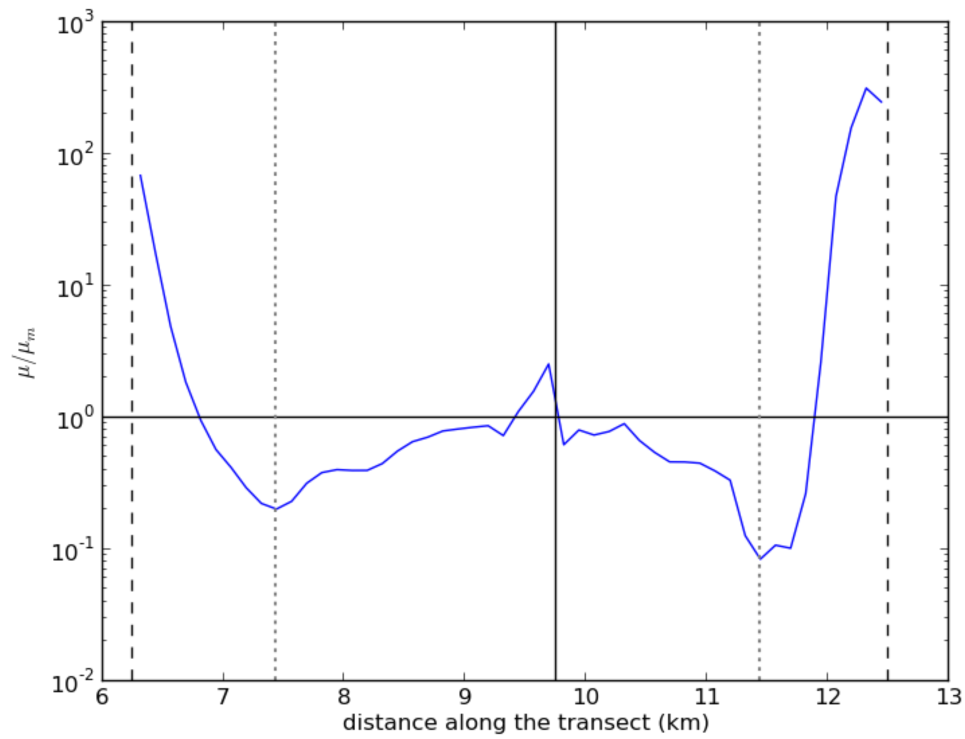

Figure 2.3: Viscosity ratio for the region between the vertical dashed lines in figure 2.2 calculated using finite difference approximations to the derivatives in equation 2.7. The vertical dotted lines are at the minima of the viscosity ratio, considered here to be the "shear margins." The solid vertical line is at x_m, the location of the maximum velocity.

2.4 Glen's Flow Law

Glen's flow law can be realized using the viscosity

$$\mu = \frac{1}{2} A^{-\frac{1}{n}} \dot{\varepsilon}^{-\frac{n-1}{n}} \tag{2.8}$$

where A is the temperature dependent parameter in equation 2.1 and $\dot{\varepsilon}$ is the second invariant of the strain rate tensor which can be defined as

$$\begin{aligned} \dot{\varepsilon}^2 &= \frac{1}{2}\left[\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial z}\right)^2\right] \\ &+ \frac{1}{4}\left[\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right)^2\right] \end{aligned} \tag{2.9}$$

The n in equation 2.8 is the same "exponent in Glen's flow law" seen in equation 2.1. Under the assumptions that derivatives with respect to y and z are negligible and there is no component of velocity except in the y direction ($u = 0$ and $w = 0$) equation 2.9 simplifies to

$$\dot{\varepsilon} = \frac{1}{2}\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right| \tag{2.10}$$

Substituting equation 2.10 into equation 2.8 then dividing both sides by μ_m shows that Glen's flow law predicts

$$\frac{\mu}{\mu_m} = \frac{\frac{1}{2}A^{-\frac{1}{n}}\left(\frac{1}{2}\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|\right)^{-\frac{n-1}{n}}}{\mu_m} = \gamma\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|^{-\frac{n-1}{n}} \tag{2.11}$$

where, if we neglect the dependence of A on temperature, γ is a constant. Taking the logarithm of both sides of equation 2.11 and using the rules of logarithms

$$\log\frac{\mu}{\mu_m} = \log\gamma - \left(\frac{n-1}{n}\right)\log\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right| \tag{2.12}$$

According to equation 2.12, if $\frac{\mu}{\mu_m}$ is plotted versus $\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|$ on a log-log scale the curve should be a line with slope

$$m = -\left(\frac{n-1}{n}\right) \tag{2.13}$$

For a Newtonian fluid, having $n = 1$, the slope will be zero. As n approaches infinity, the slope approaches negative one. Equation 2.13 can be solved for the exponent in Glen's flow law

$$n = \frac{1}{1+m} \tag{2.14}$$

The viscosity ratios $\frac{\mu}{\mu_m}$ for the transect through Helheim Glacier plotted versus x in figure 2.3 can also be plotted versus $\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|$ (again using finite differences to approximate $\frac{\mathrm{d}v}{\mathrm{d}x}$). This is done in figure 2.4.

A striking feature of figure 2.4 is that the data points corresponding to spacial points between the shear margins (blue triangles) and the data points corresponding to spacial points beyond the shear margins (red squares) are both aligned almost linearly but with different slopes. For the region beyond the shear margin the slope of the points is close to negative one which corresponds to the exponent in Glen's flow law becoming infinite. For the region between the shear margins the slope of the points appears to correspond to an exponent in Glen's flow law between one and two. A linear least squares fit of these points has slope $m = -0.359$ which (according to equation 2.14) corresponds to an exponent $n = 1.56$.

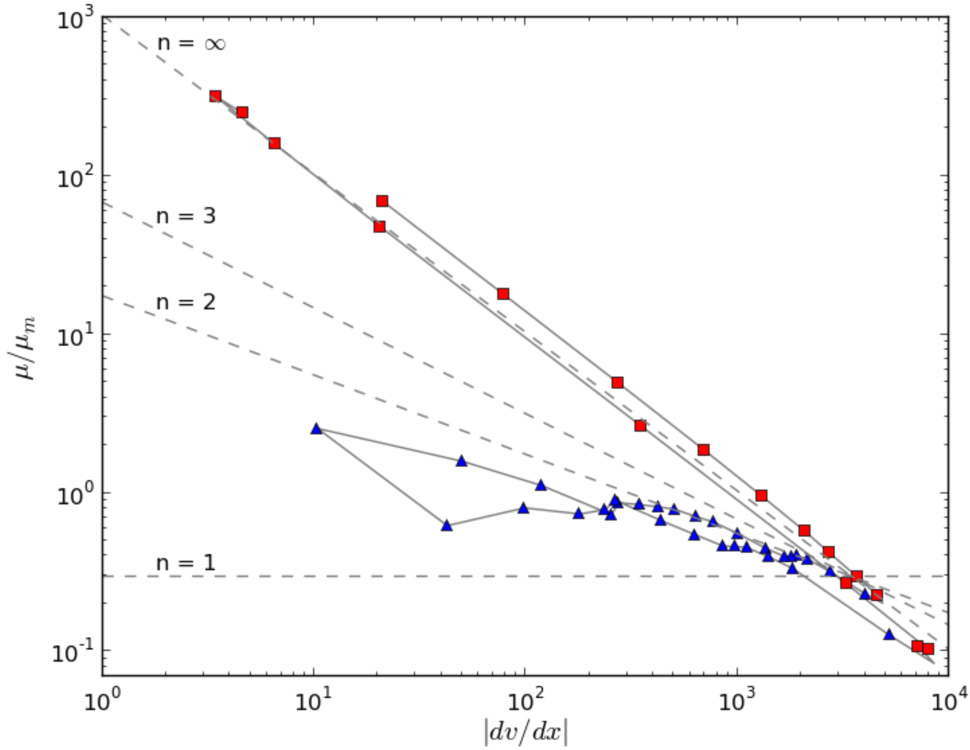

Figure 2.4: The viscosity ratio $\frac{\mu}{\mu_m}$ plotted versus $\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|$ on a log-log scale for a transect of Helheim Glacier. The blue triangles correspond to points between the shear margins (shown as dotted vertical lines in figures 2.2 and 2.3). The red squares correspond to points outside of the shear margins. Data points adjacent in space are connected by line segments. As shown in equation 2.12, for fluids behaving according to Glen's flow law the points should fall on a line. The dashed lines in this figure have slopes corresponding to Glen's flow law exponents of $n = 1$, $n = 2$, $n = 3$, and $n = \infty$.

2.5 Concluding Remarks

The analysis presented provides a methodology for assessing the exponent in Glen's flow law using measured velocities. The linear arrangement of the blue triangles in figure 2.4 with slope between zero and negative one suggests that Glen's flow law is applicable to the portion of Helheim Glacier between the shear margins. However, the model appears to break down at the shear margins beyond which the exponent in the flow law becomes infinite. A likely cause of the failure of the model beyond the shear margins is that basal traction becomes important near the edges of the glacier so that the $\frac{\partial}{\partial z}$ term in equation 2.2 is not negligible in these outlying regions. But note that basal traction is not *necessary* to produce the velocities seen in figure 2.2 including the concave upward portions outside the shear margins; they *could* (at least from a mathematical viewpoint) be due to the viscosity variations shown in figure 2.3.

To investigate the effects of basal traction, a two-dimensional transverse model could be used; perhaps solving an inverse problem for the basal traction and the parameters in Glen's flow law. This approach loses the simplicity of the current one-dimensional model. The basal topography of Helheim Glacier includes a vee-shaped trough so the ice is much thinner near the shear margins than near the center of the glacier. Although not very precise, interpolation of a 1 km grid digital elevation map [12] indicates that the thickness of the ice at the shear margins identified in figures 2.2 and 2.3 is less than 500 meters and these shear margins are near the edges of the trough while, near the center of the trough, ice thickness is around 1600 meters.

Neglecting the dependence on temperature of the parameter A in equation 2.11 may be a significant error in the current analysis as strain heating may introduce substantial temperature variations in the ice. Of course, other assumptions used in the mathematical model may also introduce inaccuracies. These assumptions are:

1. Deformation of the ice is governed by the Stokes equations.

2. Flow is in a straight line in the direction opposite to the surface gradient.

3. Velocity and pressure do not vary in the direction of flow.

4. Velocity does not vary in the direction perpendicular to the surface.

5. Density of the ice and the surface slope are both constant along the transect.

6. Viscosity of the ice is finite and continuous.

7. Finite differences provide a good approximation to the derivative of velocity with respect to the transverse coordinate (error in the approximation of the second derivative is inconsequential).

8. The components of velocity in the horizontal direction transverse to the flow and in the direction perpendicular to the surface are zero.

Using the velocity data shown in figure 2.1 the flow law exponent for the fast moving ice was found to be 1.6, smaller than the value 3 usually used in ice sheet models. The coefficient of determination for the linear least squares fit of the blue triangles in figure 2.4 used to determine this exponent was $R^2 = 0.81$.

The data shown in figure 2.1 represents the period from 2 September 2010 to 13 September 2010. The United States' National Aeronautics and Space Administration (NASA) Making Earth System Data Records for Use in Research Environments (MEaSUREs) program also provides similar velocity data for Helhiem Glacier for other time periods [11]. The analysis described in this paper was applied, using the same transect, for each of these other time periods. The average of the Glen's flow law exponents obtained was 1.6, agreeing with the exponent found using the 2 September 2010 to 13 September 2010 data set thus making it (the data set analyzed in detail in this paper) somewhat representative of all of the data. For some of the data sets the coefficient of determination was as low as $R^2 = 0.32$ largely due to points near the location of maximum velocity not falling linearly on a plot of $\frac{\mu}{\mu_m}$ versus $|\frac{\mathrm{d}v}{\mathrm{d}x}|$ (like that in figure 2.4). Even in figure 2.4 there is a loop in the leftmost part of the curve through the blue triangles which corresponds to the eight data points (four on each side) closest to x_m, the location of the maximum velocity. (These eight data points represent a distance of about 1 km). The derivative of the velocity at these points is relatively small making the finite difference approximations to this derivative more susceptible to error in the data. This error in the approximation of the derivative may be partly responsible for the somewhat jagged appearance of the curve in figure 2.3 near x_m. With these eight points removed, the coefficient of determination of the linear least squares fit of the ratio $\frac{\mu}{\mu_m}$ versus $|\frac{\mathrm{d}v}{\mathrm{d}x}|$ (on a log-log scale) increases from $R^2 = 0.81$ to $R^2 = 0.90$ and the slope is $m = -0.541$ corresponding to an exponent in Glen's flow law of $n = 2.18$. For many of the data sets, deleting the eight data points closest to the location of maximum velocity dramatically increased the coefficient of

determination. Results of the analysis for all of the time periods for which data was available are given in table 2.1.

This study has considered viscosity ratios, which are sufficient for examining the exponent in Glen's flow law and require only the velocity data. Using surface elevation measurments the surface slope along the transect could be obtained and the value of the viscosity of the ice could then be evaluated using equations 2.5 and 2.6. Using these viscosities the value of the parameter A in equations 2.1 and 2.8 (*i.e.* Glen's flow law) could also be estimated.

Table 2.1: Glen's flow law exponent, n, calculated for different data sets. Also given is the coefficient of determination, R^2, for the linear least squares fit of the data used to calculate n. The columns labeled "Culled data" are results of analysis with the eight data points closest to the the location of maximum velocity deleted. Bold font identifies the data set presented in detail in this paper.

Data set	All data		Culled data	
Start date - End date	n	R^2	n	R^2
27 Jan 09 - 07 Feb 09	1.52	0.80	2.17	0.89
25 Apr 09 - 06 May 09	1.37	0.32	2.21	0.77
19 Jun 09 - 30 Jun 09	1.52	0.34	2.45	0.93
24 Aug 09 - 04 Sep 09	1.77	0.88	2.17	0.89
20 Nov 09 - 01 Dec 09	1.83	0.70	2.72	0.94
23 Apr 10 - 04 May 10	1.71	0.41	2.58	0.92
09 Jul 10 - 20 Jul 10	1.93	0.60	2.27	0.76
02 Sep 10 - 13 Sep 10	**1.56**	**0.81**	**2.18**	**0.90**
18 Nov 10 - 29 Nov 10	1.48	0.57	2.44	0.96
08 Mar 11 - 19 Mar 11	1.45	0.69	2.06	0.93
04 Jun 11 - 15 Jun 11	1.57	0.38	2.44	0.97
07 Jul 11 - 18 Jul 11	1.64	0.43	2.49	0.94
09 Aug 11 - 20 Aug 11	1.41	0.47	2.17	0.94
16 Nov 11 - 27 Nov 11	1.64	0.60	2.03	0.80
Mean	1.60	0.57	2.31	0.90
Standard deviation	0.16	0.18	0.20	0.07

Chapter 3

A Forward Problem: Solving for Velocity Given Viscosity

A finite element model is described and results generated using this model are presented.

3.1 Introduction

In the preceding chapter, measured velocities and a mathematical model were used to investigate the variation in viscosity along a transect of Helheim Glacier. If that analysis is correct, one should be able to use the viscosity variations found to reproduce the measured velocities. That is the goal of the current chapter. A boundary value problem is described and the finite element method is used to approximate solutions to this boundary value problem. The approximate solution obtained agrees with the measured velocity along the transect of Helheim Glacier. There are two reasons for performing this analysis:

1. It provides a check that the analysis in the preceding chapter is correct.
2. The finite element model developed will be used in a subsequent chapter as a component in the solution of an "inverse problem".

3.2 A Boundary Value Problem

A "boundary value problem" is a type of problem consisting of a differential equation that must be satisfied within a specified domain together with "boundary conditions"

that must be satisfied on the boundary of this domain. Boundary value problems are often used as a model of a physical system (as is the case here.)

In Chapter 2 a mathematical model was obtained from the Stokes equations by making assumptions that derivatives with respect to y and z were negligible. The resulting equation (equation 2.3) is the differential equation in the boundary value problem considered in this chapter

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[\mu\frac{\mathrm{d}v}{\mathrm{d}x}\right] = -\rho g\alpha \qquad \text{for} \qquad a < x < b \tag{3.1}$$

Here a and b are the coordinates of the left and right edges of the portion of the transect analysed in Chapter 2. (These locations are represented by the dashed vertical lines in figures 2.2 and 2.3.) As in Chapter 2, the viscosity μ is a function of x and $\rho g\alpha$ is a constant. Unlike in Chapter 2, where the analysis was based on a known velocity v, here it is assumed that μ and $\rho g\alpha$ are known and the goal is to solve for v. To complete the boundary value problem, boundary conditions assert that the velocity at the lateral edges of the glacier are zero

$$v(a) = 0 \tag{3.2}$$

$$v(b) = 0 \tag{3.3}$$

In Chapter 2 the viscosity was determined to within a constant (that is, we found $\frac{\mu}{\mu_m}$ but we don't know μ_m) and no information was gained about $\rho g\alpha$. Dividing both sides of equation 3.1 by μ_m gives

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[\left(\frac{\mu}{\mu_m}\right)\frac{\mathrm{d}v}{\mathrm{d}x}\right] = -\frac{\rho g\alpha}{\mu_m} \tag{3.4}$$

so in essence there is only one unknown constant, $\dfrac{\rho g\alpha}{\mu_m}$. From a physical viewpoint the differential equation of interest in this chapter is actually 3.4, but to avoid the more complicated appearance of equation 3.4, it is mathematically equivalent to set $\mu_m = 1$ and use equation 3.1 instead.

The solution, v, to the boundary value problem (equations 3.1, 3.2, and 3.3) varies linearly with $\rho g\alpha$. For example, if a solution, $v(x)$, is known for a given value of $\rho g\alpha$, then the solution with $\rho g\alpha$ doubled is simply two times the original solution. Using the viscosity ratio found in Chapter 2 for μ, one should be able to choose an

appropriate value for $\rho g \alpha$ so that the solution to the boundary value problem agrees with the measured velocities along the transect of Helheim Glacier (*i.e.* the velocities between the dashed vertical lines in figure 2.2.)

The boundary value problem (equations 3.1, 3.2, and 3.3) has the exact solution

$$v(x) = \rho g \alpha \int_a^x \frac{C - \tilde{x}}{\mu(\tilde{x})} d\tilde{x} \tag{3.5}$$

where

$$C = \frac{\int_a^b \frac{x}{\mu(x)} dx}{\int_a^b \frac{1}{\mu(x)} dx} \tag{3.6}$$

but without a formula for $\mu(x)$ this solution has limited value. One could obtain an approximation to the integrals using numerical quadrature but here a different approach is used; an approximate solution to the boundary value problem is found using the finite element method.

3.3 The Finite Element Method

The finite element method for approximating the solution to the boundary value problem given in the preceding section is based on the assertion that if a function $v(x)$ satisfies the differential equation 3.1 then for every "test function" $\psi(x)$ the following integral equation is also satisfied

$$\int_a^b \left(\frac{\mathrm{d}}{\mathrm{d}x} \left[\mu \frac{\mathrm{d}v}{\mathrm{d}x} \right] \right) \psi(x) dx = - \int_a^b \rho g \alpha \psi(x) dx \tag{3.7}$$

Performing integration by parts on the left hand side of equation 3.7 gives

$$\left[\mu \frac{\mathrm{d}v}{\mathrm{d}x} \right] \psi \Big|_a^b - \int_a^b \left[\mu \frac{\mathrm{d}v}{\mathrm{d}x} \right] \frac{\mathrm{d}\psi}{\mathrm{d}x} dx = - \int_a^b \rho g \alpha \psi dx \tag{3.8}$$

so for every test function $\psi(x)$ with $\psi(a) = 0$ and $\psi(b) = 0$

$$- \int_a^b \left[\mu \frac{\mathrm{d}v}{\mathrm{d}x} \right] \frac{\mathrm{d}\psi}{\mathrm{d}x} dx = - \int_a^b \rho g \alpha \psi dx \tag{3.9}$$

The next step in the finite element method is to approximate the solution $v(x)$ using a prescribed set of basis functions ϕ_k

$$v(x) = \sum_{k=0}^{n} v_k \phi_k(x) \tag{3.10}$$

where the coefficients v_k are constants that are to be found. Piecewise linear basis functions are often used and were chosen for the current study

$$\phi_k(x) = \begin{cases} 0 & \text{if } \; x \leq x_{k-1} \\ \frac{x - x_{k-1}}{x_k - x_{k-1}} & \text{if } \; x_{k-1} < x \leq x_k \\ \frac{x_{k+1} - x}{x_{k+1} - x_k} & \text{if } \; x_k < x \leq x_{k+1} \\ 0 & \text{if } \; x > x_{k+1} \end{cases} \tag{3.11}$$

where, for a uniform mesh such as that used here,

$$x_k = a + \left(\frac{b-a}{n} \right) k \tag{3.12}$$

The locations x_k are refered to as "nodes" in the parlance of finite element methods. The prescribed set of basis functions can also be used to approximate μ

$$\mu(x) = \sum_{k=0}^{n} \mu_k \phi_k(x) \tag{3.13}$$

but unlike v (which is the unknown in the boundary value problem) appropriate values for the μ_k's can be chosen immediately based on the known values of μ at discrete points found in Chapter 2.

A piecewise linear approximation to the solution of the boundary value problem is obtained by finding values for the $n+1$ coefficients v_k for $k = 0, 1, 2, 3, \ldots, n$ in equation 3.10. The boundary conditions given by equations 3.2 and 3.3 are satisfied by choosing $v_0 = 0$ and $v_n = 0$. To find values for the remaining $n-1$ unknown coefficients (v_k for $k = 1, 2, 3, \ldots, n-1$) a system of $n-1$ equations is solved. These equations correspond to equation 3.9 with $n-1$ different test functions ψ. The test functions chosen are simply the basis functions, $\psi(x) = \phi_j(x)$ for $j = 1, 2, 3, \ldots, n-1$ (all of which satisfy the conditions $\psi(a) = 0$ and $\psi(b) = 0$.) With the chosen approximations for v and μ (equations 3.10 and 3.13), and test function

$\psi(x) = \phi_j(x)$, the derivatives and integrals in equation 3.9 can be evaluated in closed form so a system of algebraic equations is obtained which can be solved for the desired coefficients v_k.

The FEniCS software package [7, 8] was used to execute the finite element method. Using this package, a relatively simple *Python* script prescribing the "weak form" of the differential equation (equation 3.9), the boundary conditions (equations 3.2 and 3.3), the number of finite elements n, and specifying that piecewise linear basis functions are to be used, is sufficient to obtain an approximate solution to the boundary value problem. The most challenging task in using FEniCS for this boundary value problem is incorporating the data for μ.

3.4 The Solution of the Forward Problem

Velocities, v, found using the finite element method to approximate the solution to the boundary problem given by equations 3.1, 3.2, and 3.3 when using the viscosity ratio shown in figure 2.3 as μ are plotted in figure 3.1 together with the measured velocities along the transect of Helheim Glacier for comparison. The value (approximately 443) of the right hand side of the differential equation (equation 3.1 or more accurately from a physical standpoint equation 3.4) was chosen such that the maximum calculated velocity matches the maximum measured velocity. The calculated velocities throughout the domain are in good agreement with the measured velocities although they fall slightly below the measured values in some locations. One of these locations is at the extreme left edge of the plot in figure 3.1 where the discrepancy is due to the fact that the measured velocity at this location is 32 meters/annum but the boundary condition (equation 3.2) in the model sets the velocity there to zero.

The curve representing the calculated velocities is somewhat smoother than the curve representing the measured velocities because the calculated curve includes twice as many points. This was done because the viscosity ratios found in Chapter 2 were calculated at the midpoints of the velocity data. The finite element model includes nodes at the points where viscosities were calculated as well as the at the points where velocities had been obtained along the transect. A total of 101 nodes were used in the finite element model ($n = 100$ in equation 3.10) corresponding to the 51 velocity data points and 50 midpoints where viscosity was calculated in the analysis in Chapter 2.

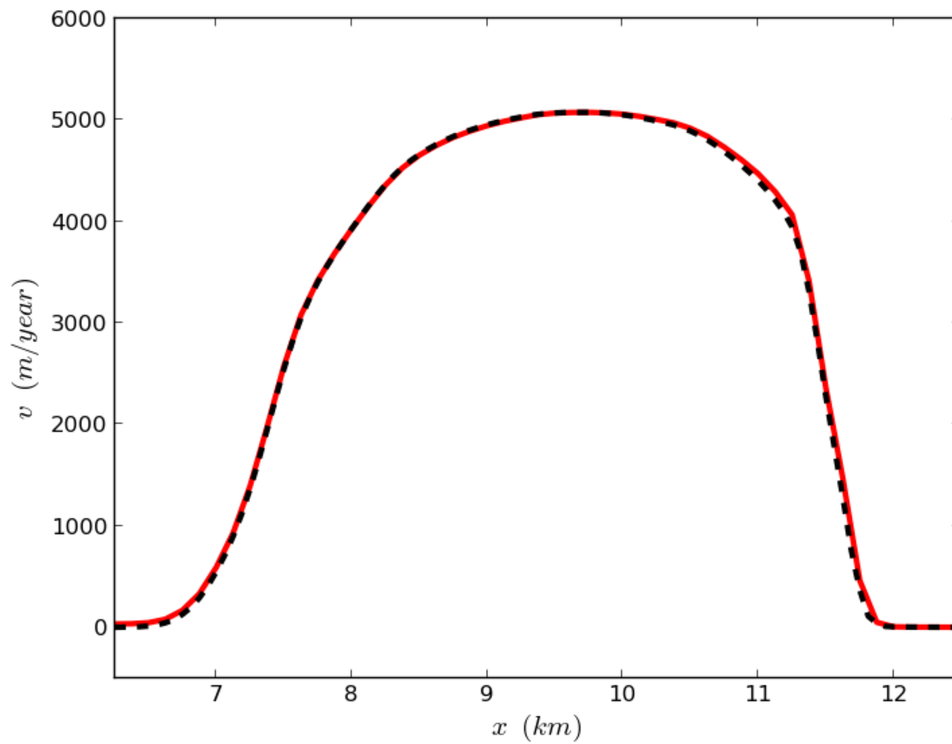

Figure 3.1: Velocities, v, found using the finite element method to approximate the solution to the boundary problem given by equations 3.1, 3.2, and 3.3 are plotted as a black dashed line. The solid red line shows the measured velocities along the transect of Helheim Glacier for comparison.

Chapter 4

An Inverse Problem: Solving for Viscosity Given Velocity

An alternative method to that presented in Chapter 2 is used to investigate ice viscosity variations using measured surface velocities on Helheim Glacier.

4.1 Introduction

In Chapter 2, measured surface velocities and a mathematical model were used to investigate the variation in viscosity along a transect of Helheim Glacier. In the current chapter the same measured surface velocities and a mathematical model very similar to that used in Chapter 2 are used to, once again, investigate the variation in viscosity along the same transect of Helheim Glacier. The primary difference is in the approach taken and the numerical approximations used. In Chapter 2 the analysis followed a direct approach and used finite differences. Here an inverse problem is posed and solved. Solution of the inverse problem involves use of the finite element method described in Chapter 3.

4.2 The Inverse Problem

Equations 3.1, 3.2, and 3.3 in Chapter 3 describe a boundary value problem. These equations, an essential part of the inverse problem studied in the current chapter, are

repeated here

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[\mu\frac{\mathrm{d}v}{\mathrm{d}x}\right] = -\rho g\alpha \qquad \text{for} \quad a < x < b \tag{4.1}$$

$$v(a) = 0 \tag{4.2}$$

$$v(b) = 0 \tag{4.3}$$

The differential equation, 4.1, was derived (in Chapter 2, equation 2.3) from the Stokes equation and is a mathematical model of the ice in Helheim Glacier: μ is the ice viscosity and v is the velocity of the ice, both considered to vary in the direction x which is transverse to the flow; ρ is the ice density, g is the acceleration of gravity, and α is (essentially) the slope of the surface of the ice, all considered to be constants. The (x) coordinates of the lateral boundaries of the glacier are given by a and b. Equations 4.2 and 4.3 are boundary conditions that assert that the velocity of the glacier is zero on its (lateral) edges.

In a typical problem in fluid mechanics, the viscosity is known and one solves for the fluid velocity. This was the problem considered in Chapter 3 and was referred to in the title of that chapter as a "forward problem". Here we are interested in the corresponding "inverse problem"; the fluid velocity is known and the viscosity is unknown.

An iterative technique can be used to find approximate solutions of inverse problems. Conceptually, for the problem at hand, one would

1. "Guess" $\mu(x)$.

2. Solve the forward problem to find $v(x)$.

3. If $v(x)$ is in close agreement with the measured values, quit; the guessed μ is an approximate solution to the inverse problem. Otherwise return to step 1.

Of course, one hopes that subsequent guesses are improvements over their predecessors. Creating these guesses is an important part of the iterative process. A related consideration is how to measure the "agreement with the measured values". A common approach is to use

$$J = \int_a^b \left[v(x) - \tilde{v}(x)\right]^2 dx \tag{4.4}$$

where $\tilde{v}(x)$ is the desired velocity (derived from the measured values; note that $\tilde{v}(x)$ must be defined over the entire interval (a, b) but the measured values are only at discrete points).

It can seen from equation 4.4 that $J \geq 0$ and $J = 0$ if and only if $v(x) = \tilde{v}(x)$ for all $x \in (a, b)$. Thus the inverse problem can be posed as an optimization problem:

> *Find $\mu(x)$ such that the solution $v(x)$ of the boundary value problem given by equations 4.1, 4.2, and 4.3 minimizes the objective function J given by equation 4.4.*

Using the finite element method to approximate the boundary value problem (as was done in Chapter 3), the optimization problem is discretized:

> *Find the coefficients μ_k in equation 3.13 such that the approximate solution $v(x)$ of the boundary value problem minimizes the objective function J given by equation 4.4.*

Using the the basis functions (equation 3.11) to specify $\tilde{v}(x)$

$$\tilde{v}(x) = \sum_{k=0}^{n} \tilde{v}_k \phi_k(x) \tag{4.5}$$

(*cf.* equations 3.10 and 3.13) facilitates evaluation of the integral in equation 4.4. The coefficients $\tilde{v}_k$ are chosen using the measured velocities.

Algorithms for solving optimization problems have been developed and continue to be developed [13]. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm was chosen for use here. The primary purpose of these algorithms is effectively choosing the sequence of "guesses" in an iterative process. Because the goal is to minimize the objective function J, most efficient optimization algorithms (including the BFGS algorithm) require information about how the objective function changes (locally) as the problem parameters (in our case the μ_k's) change. This information, which is a collection of partial derivatives, $\frac{\partial J}{\partial \mu_k}$ for $k = 0, 1, 2, 3, \ldots, n$, can be obtained using an adjoint method. Adjoint methods are described in an appendix to this thesis.

4.3 A Solution of the Inverse Problem

A *Python* script was written to obtain an approximate solution to the discrete optimization problem described in the preceding section. An important part of solving the

inverse problem is solving the forward problem with various $\mu(x)$'s; this was accomplished using the finite element method described in Chapter 3, implemented using the *FEniCS dolfin* module. The *fmin_bfgs* function in the *scipy.optimize* module was used to carry out the BFGS algorithm. Partial derivatives required by the BFGS algorithm were obtained using the *dolfin_adjoint* module.

A small technical detail: The viscosity of a fluid is a positive quantity. This is not stipulated in equation 4.1. There are optimization algorithms that restrict the parameters in the problem to positive quantities, even versions of the BFGS algorithm, but this restriction was not enforced in the *Python* function chosen from the *scipy.optimize* package to carry out the BFGS algorithm. Instead, to ensure that the viscosity in our mathematical model is positive, μ in equation 4.1 was replaced with η^2

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[\eta^2\frac{\mathrm{d}v}{\mathrm{d}x}\right] = -\rho g\alpha \qquad \text{for} \qquad a < x < b \tag{4.6}$$

and the inverse problem was solved for η. The viscosity can then be obtained using

$$\mu = \eta^2 \tag{4.7}$$

Thus μ remained positive, not only in the final solution, but throughout the iterative process used to find it.

Before solving the inverse problem the measured velocities were normalized by dividing by the maximum measured velocity. After solving, the solution, $\mu(x)$, was normalized by dividing by $\mu_m = \mu(x_m)$ where x_m is the location of the maximum (calculated) velocity, to obtain viscosity ratios analogous to those found in Chapter 2. The initial "guess" of the viscosity was a constant, $\mu(x) = C$ (or, more accurately stated for the final form of the inverse problem, $\eta(x) = \sqrt{C}$), where C was of the order of magnitude of the average of the solution ($\mu(x)$) obtained.

When the *Python* script was run, the final iteration of the BFGS algorithm generated the velocities plotted in figure 4.1 (which are very close to the measured values) using the viscosity variations shown in figure 4.2. The viscosity ratio obtained in Chapter 2 (*cf.* figure 2.3) is also plotted in figure 4.2 for comparison. The BFGS algorithm took 616 iterations to reach this solution.

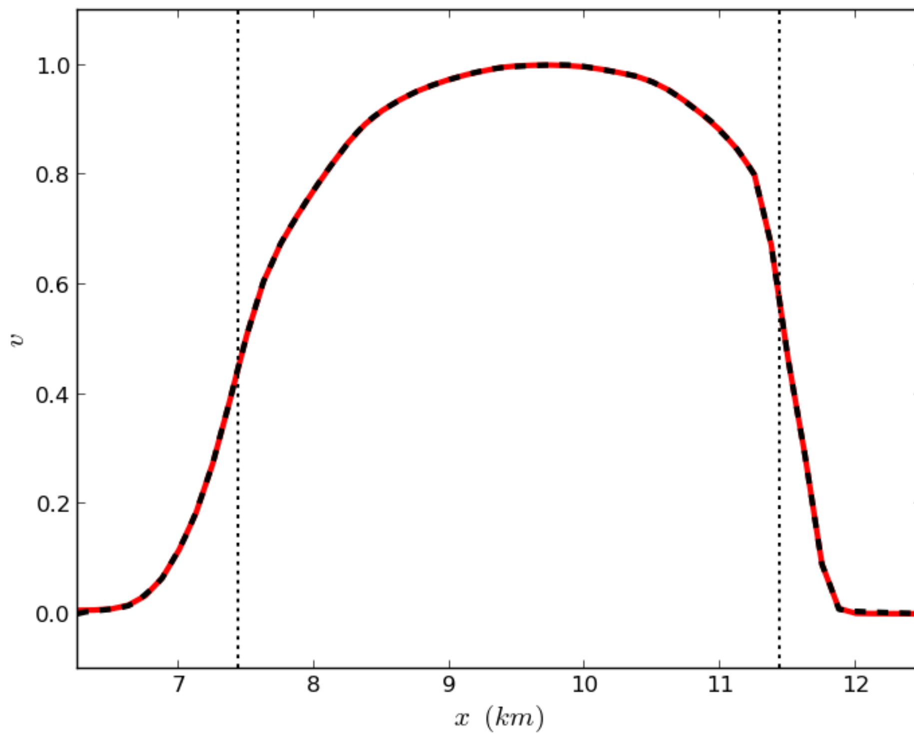

Figure 4.1: The velocity, v, obtained when solving the inverse problem is plotted as a dashed black line. The solid red curve shows the normalized velocity along the transect of Helheim Glacier. A good match of the measured velocities has been accomplished. The vertical dotted lines are at the "shear margins", defined here as the locations of the minima of viscosity (as plotted in figure 4.2).

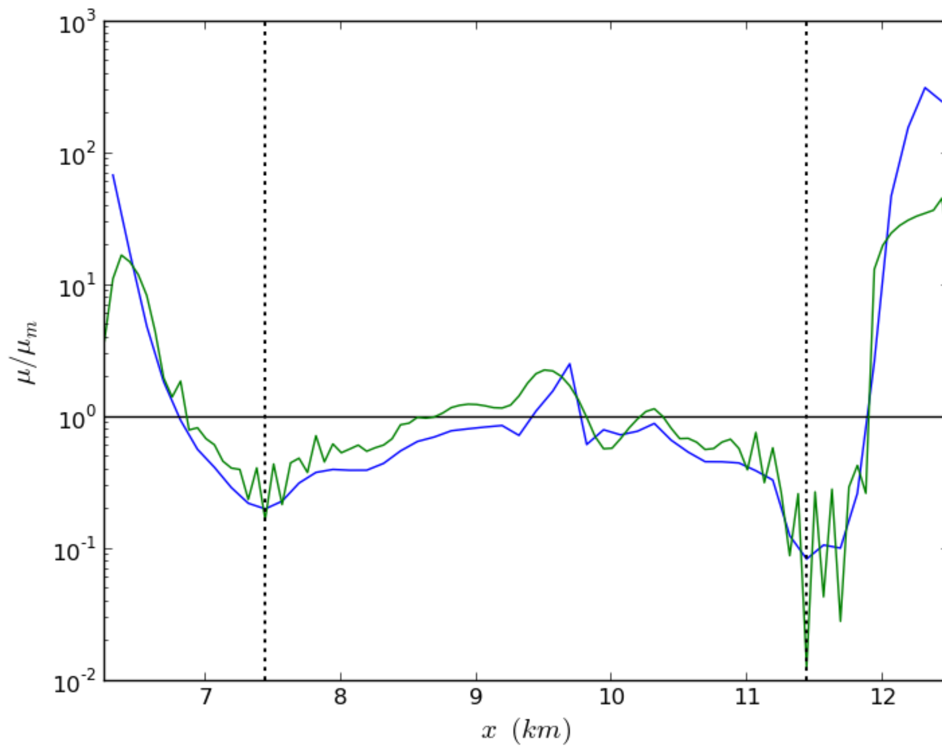

Figure 4.2: The viscosity ratio, $\frac{\mu}{\mu_m}$, obtained by solving the inverse problem is plotted as a green curve. The blue curve shows the viscosity ratio obtained in Chapter 2 for comparison. The vertical dotted lines are at the minima of the viscosity ratio, considered here to be the "shear margins."

4.4 Tikhonov Regularization

An undesirable feature of the viscosity ratio curve in figure 4.2 obtained by solving the inverse problem is that high frequency oscillations occur near its minima (*i.e.* at the shear margins). Such high frequency oscillations are somewhat common when numerically solving inverse problems. It is not expected that the viscosity of the ice would vary in this way although if it did the velocity might not be affected much (which is why the oscillations appear in the solution; at least in the finite element approximation the velocity does not oscillate even though the viscosity does). To reduce such oscillations a "regularization term" can be included in the objective function of the optimization problem. The regularization term is designed to cause the value of the objective function, which is being minimized, to grow if the undesired oscillations are present. Thus, in the process of minimizing the objective function, undesired oscillations are suppressed.

Within the oscillations $\left|\frac{\mathrm{d}\mu}{\mathrm{d}x}\right|$ becomes large. So an effective regularization term might penalize high values of $\left|\frac{\mathrm{d}\mu}{\mathrm{d}x}\right|$. Since $\mu = \eta^2$ penalizing high values of $\left|\frac{\mathrm{d}\eta}{\mathrm{d}x}\right|$ would also be appropriate. Instead of using absolute values ($\left|\frac{\mathrm{d}\eta}{\mathrm{d}x}\right|$), the smoother squaring function ($\left[\frac{\mathrm{d}\eta}{\mathrm{d}x}\right]^2$) is usually used. Adding a term penalizing high values of $\left[\frac{\mathrm{d}\eta}{\mathrm{d}x}\right]^2$ to equation 4.4 creates the objective function

$$J = \int_a^b \left([v(x) - \tilde{v}(x)]^2 + \gamma \left[\frac{\mathrm{d}\eta}{\mathrm{d}x}\right]^2 \right) dx \tag{4.8}$$

Here γ is a positive coefficient weighting the regularization term. The objective of the optimization problem becomes to minimize J in equation 4.8 instead of the J in equation 4.4.

Choice of the value of γ affects the viscosity obtained and how well the calculated velocities match the desired (measured) values. If γ is large, the solution of the optimization problem will be a constant viscosity ($\frac{\mathrm{d}\eta}{\mathrm{d}x} = 0$ everywhere so $\mu = \eta^2$ is constant) and a parabolic velocity curve is obtained whose maximum value is close to the maximum measured velocity but otherwise does not match the desired velocity curve very well. See figure 4.3. At the other extreme, if $\gamma = 0$ the solution in figures 4.1 and 4.2 is obtained. For this study a very subjective method was used to choose γ; values were tried until a velocity curve that was judged to be in acceptable agreement with the measured velocities was calculated with a viscosity containing an acceptably

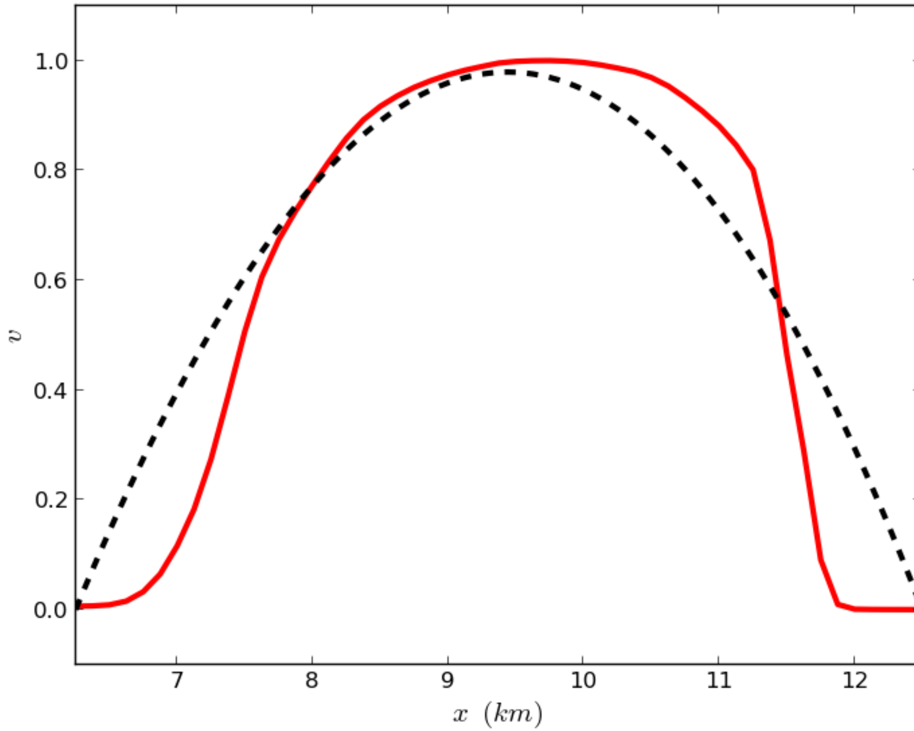

Figure 4.3: The velocity, v, obtained when solving the inverse problem posed as an optimization problem including a regularization term with a large value for γ is plotted as a dashed black line. The solid red curve shows the normalized velocity along the transect of Helheim Glacier. The calculated velocities do not match the measured velocities very well. Since the viscosity for this approximation is essentially constant, the associated viscosity ratio curve would be given by the solid black horizontal line in figure 4.2
.

small amount of oscillation. The resulting velocity and viscosity variation for the selected value of γ are plotted in figures 4.4 and 4.5.

The calculated velocity in figure 4.4 is in good agreement with the measured velocities within the shear margins but there are some noticeable discrepancies near the ends of the domain (*i.e.* near the edges of the glacier). Only a small remnant of the oscillations in the viscosity, so prevalent in figure 4.2, remains in figure 4.5, in the form of a local maximum that occurs near the right shear margin. A local maximum there is also seen in the viscosity ratio curve from Chapter 2 so it would seem that there is some feature of the velocity that causes it.

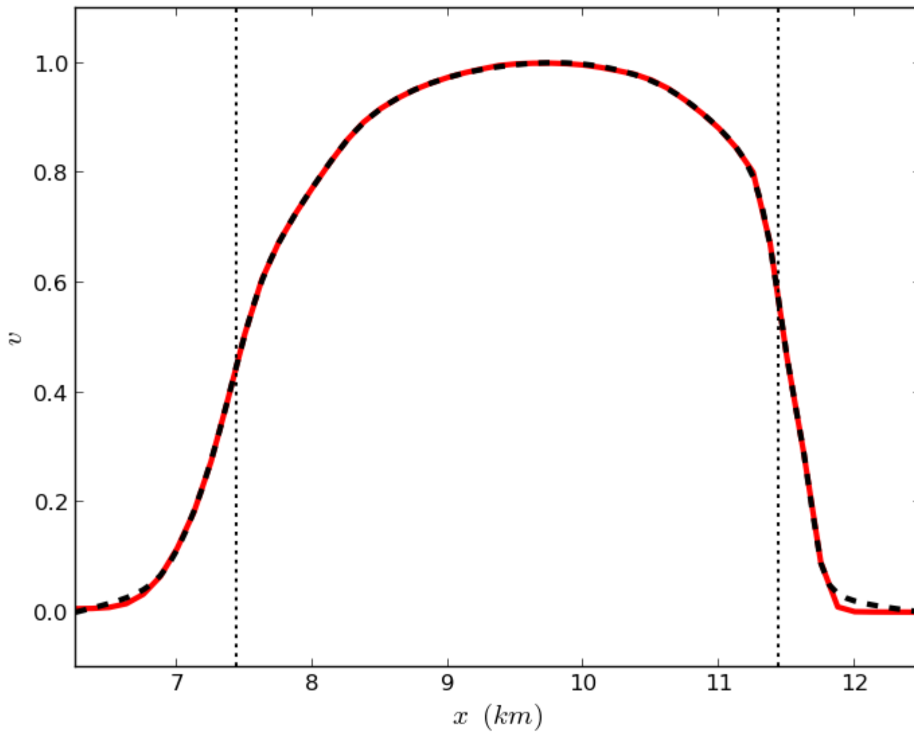

Figure 4.4: The velocity, v, obtained when solving the inverse problem posed as an optimization problem including a regularization term is plotted as a dashed black line. The solid red curve shows the normalized velocity along the transect of Helheim Glacier. The calculated velocities match the measured velocities well except near the edges. The vertical dotted lines are at the “shear margins”, defined here as the locations of the minima of viscosity (as plotted in figure 4.5).

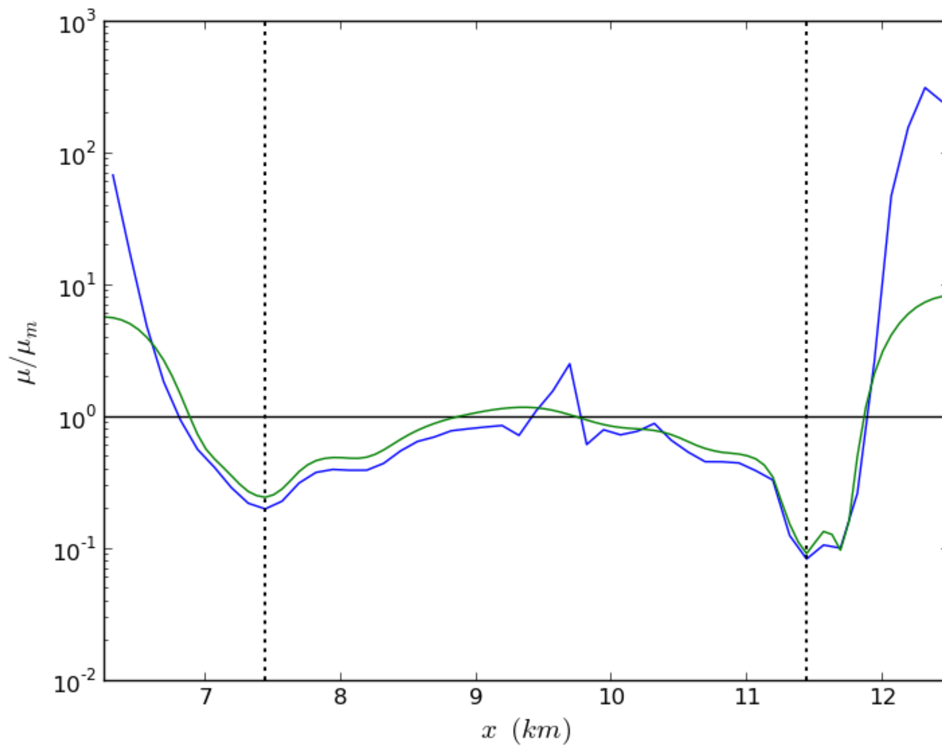

Figure 4.5: The viscosity ratio, $\frac{\mu}{\mu_m}$, obtained by solving the inverse problem posed as an optimization problem including a regularization term is plotted as a green curve. The blue curve shows the viscosity ratio obtained in Chapter 2 for comparison. The vertical dotted lines are at the minima of the viscosity ratio, considered here to be the "shear margins."

4.5 Evaluating the Exponent in Glen's Flow Law

In Chapter 2 it was shown that if the one-dimensional mathematical model presented in this thesis is appropriate and Glen's flow law is applicable then a plot of the viscosity ratio $\frac{\mu}{\mu_m}$ versus $\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|$ will be linear, with the slope of the line revealing the value of the exponent in the flow law (see equation 2.12). The viscosity ratio $\frac{\mu}{\mu_m}$ plotted versus x in figure 4.5 can be plotted versus $\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|$ where v is the calculated velocity in figure 4.4. Unlike in Chapter 2, instead of using finite differences to obtain $\frac{\mathrm{d}v}{\mathrm{d}x}$, here the finite element method is used to solve the differential equation $\epsilon = \frac{\mathrm{d}v}{\mathrm{d}x}$ (where v is given by equation 3.10) for ϵ. The resulting plot is shown in figure 4.6.

The blue triangles on the left half of figure 4.6 correspond to the portion of Helheim Glacier near the location of the maximum velocity. The almost horizontal arrangement (slope $m = 0$) of these triangles suggest that the ice there is behaving similarly to a Newtonian fluid ($n = 1$ in Glen's flow law, see equation 2.14).

If the ice was a Newtonian fluid, the velocity curve would be a parabola. (The solutions to equation 4.1 with constant μ are second order polynomials.) A least squares fit of a second order polynomial to the 21 velocity data points around the location of maximum velocity gives a good but not flawless fit to the velocity data along the transect. See figure 4.7.

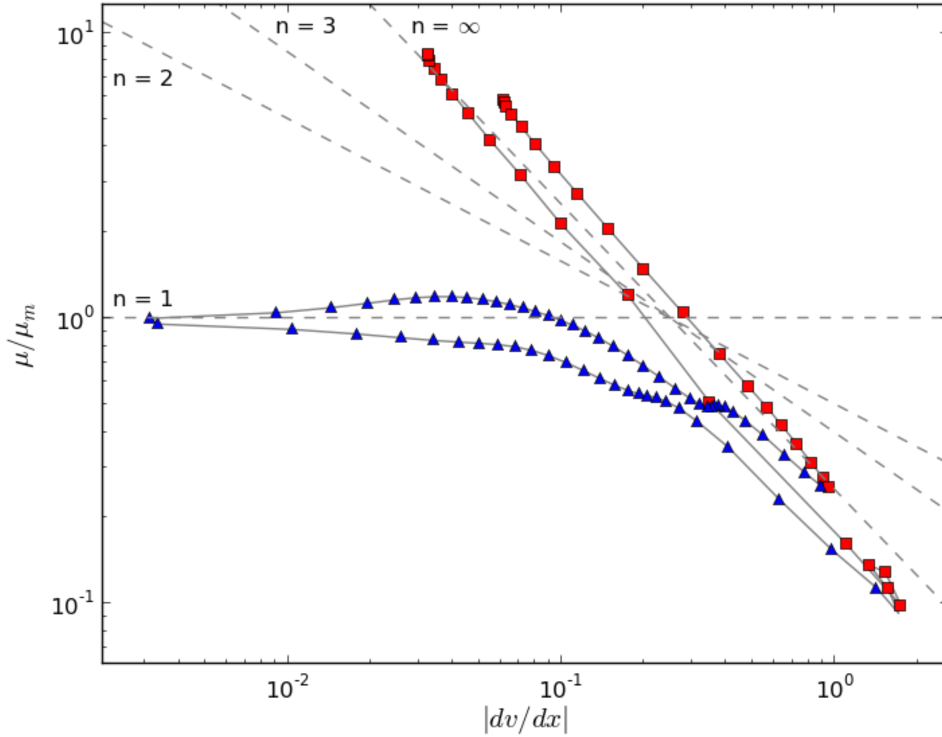

Figure 4.6: The viscosity ratio $\frac{\mu}{\mu_m}$ from figure 4.5 plotted versus $\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|$ from figure 4.4 on a log-log scale. The blue triangles correspond to points between the shear margins (shown as dotted vertical lines in figures 4.4 and 4.5). The red squares correspond to points outside of the shear margins. Data points adjacent in space are connected by line segments. As shown in equation 2.12, for fluids behaving according to Glen's flow law the points should fall on a line. The dashed lines in this figure have slopes corresponding to Glen's flow law exponents of $n = 1$ (for the horizontal line), $n = 2$, $n = 3$, and $n = \infty$ (for the line with slope -1).

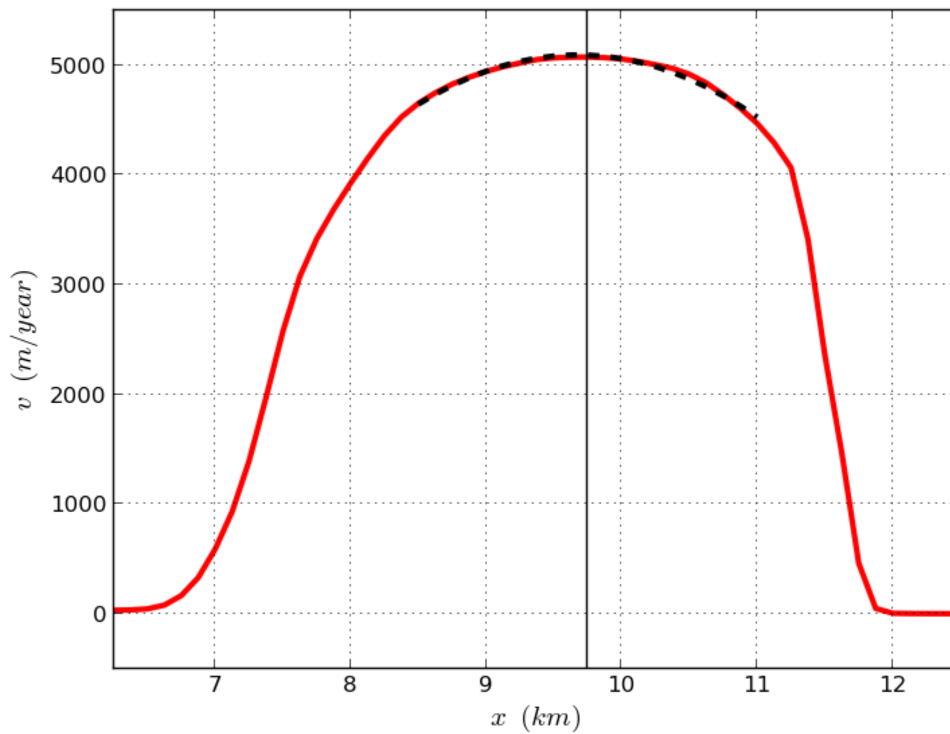

Figure 4.7: The solid red curve shows the velocity along the transect of Helheim Glacier. The dashed black line represents a least squares fit of a second order polynomial to the 21 data points around the location of maximum velocity (which is indicated by the vertical solid black line). A more detailed view of this parabola is available in panel (b) of figure 5.1.

Chapter 5

Using Least-Squares Fits to Evaluate the Exponent in Glen's Flow Law

After incorporating Glen's flow law in the differential equation introduced in Chapter 2 to model ice flow, the equation is solved, yielding an algebraic equation for the velocity of the ice involving three parameters. Choosing these parameters judiciously one can approximately match the measured velocities along the transect of Helheim Glacier but the quality of the match obtainable is influenced by the assumed value of the exponent in Glen's flow law. The value of the exponent in Glen's flow law which allows the best quality match is identified and presumed to be the "actual" value of the exponent in Glen's flow law for the ice in Helheim Glacier. The value identified is 1.6, in agreement with the value for the exponent in Glen's flow law found in Chapter 2.

5.1 Introduction

At the close of Chapter 4 if was noted that if μ is constant then the solutions to the equation

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[\mu\frac{\mathrm{d}v}{\mathrm{d}x}\right] = -\rho g\alpha \tag{5.1}$$

are second order polynomials. The best (in a least-squares sense) second order polynomial fit to the central part of the velocity curve for the transect of Helheim Glacier was shown in figure 4.7. If the representation of viscosity (μ) using Glen's flow law

(*cf.* equations 2.8, and 2.10)

$$\mu = \frac{1}{2}A^{-\frac{1}{n}}\dot{\varepsilon}^{-\frac{n-1}{n}} = \frac{1}{2}A^{-\frac{1}{n}}\left(\frac{1}{2}\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|\right)^{-\frac{n-1}{n}} = \tilde{\gamma}\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|^{-\frac{n-1}{n}} \tag{5.2}$$

is invoked, this parabola represents velocities for $n = 1$. (In equation 5.2, $\tilde{\gamma}$ is a constant, related to γ in equation 2.11 by $\tilde{\gamma} = \mu_m\gamma$. We assume in this chapter, as we did earlier, that the ice is isothermal and the surface slope does not vary along the transect.)

In this chapter, least squares fits of the central part of the velocity curve for the transect of Helheim Glacier are calculated using values of the exponent (n) in Glen's flow law other than one. The value of n that allows the best least squares fit is deemed the best value for the exponent in Glen's flow law.

5.2 Solution of the Differential Equation

Obtaining the least squares fits described in the preceding section requires a closed form solution to the differential equation that results from substituting equation 5.2 into equation 5.1

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[\tilde{\gamma}\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|^{-\frac{n-1}{n}}\frac{\mathrm{d}v}{\mathrm{d}x}\right] = -\rho g\alpha \tag{5.3}$$

As a first step, both sides of equation 5.3 are divided by the constant $\tilde{\gamma}$ and the positive constant

$$C_0 = \frac{1}{n+1}\left(\frac{\rho g\alpha}{\tilde{\gamma}}\right)^n > 0 \tag{5.4}$$

is defined such that

$$C_0^{1/n}(n+1)^{1/n} = \left[\frac{1}{n+1}\left(\frac{\rho g\alpha}{\tilde{\gamma}}\right)^n\right]^{1/n}(n+1)^{1/n} = \frac{\rho g\alpha}{\tilde{\gamma}} \tag{5.5}$$

so equation 5.3 can be written as

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|^{-\frac{n-1}{n}}\frac{\mathrm{d}v}{\mathrm{d}x}\right] = -C_0^{1/n}(n+1)^{1/n} \tag{5.6}$$

The reason for defining C_0 with the rather complicated expression given by equation 5.4 is that the solution to equation 5.6 can be written in a relatively simple form

$$v(x) = C_2 - C_0 \left| C_1 - x \right|^{n+1} \tag{5.7}$$

where C_1 and C_2 are arbitrary constants. Verifying that the v defined by 5.7 satisfies equation 5.6 is complicated by the appearance of absolute values in both the differential equation and its solution. However, by independently considering the two possible cases, $x \leq C_1$ and $x > C_1$, it is shown below that equation 5.7 is indeed the solution to equation 5.6. First, consider the case $x \leq C_1$.

If $x \leq C_1$ then $C_1 - x \geq 0$ so $|C_1 - x| = C_1 - x$ and (substituting into equation 5.7)

$$v(x) = C_2 - C_0 \left(C_1 - x \right)^{n+1} \tag{5.8}$$

which can be differentiated giving

$$\frac{\mathrm{d}v}{\mathrm{d}x} = C_0(n+1) \left(C_1 - x \right)^n \geq 0 \tag{5.9}$$

So $\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right| = \frac{\mathrm{d}v}{\mathrm{d}x}$ and

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}x}\left[\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|^{-\frac{n-1}{n}} \frac{\mathrm{d}v}{\mathrm{d}x}\right] = \frac{\mathrm{d}}{\mathrm{d}x}\left[\left(\frac{\mathrm{d}v}{\mathrm{d}x}\right)^{-\frac{n-1}{n}} \frac{\mathrm{d}v}{\mathrm{d}x}\right] = \frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{\mathrm{d}v}{\mathrm{d}x}\right)^{\frac{1}{n}} \\ = \frac{\mathrm{d}}{\mathrm{d}x}\left[C_0(n+1)\left(C_1 - x\right)^n\right]^{\frac{1}{n}} = -C_0^{1/n}(n+1)^{1/n} \end{aligned} \tag{5.10}$$

verifying that equation 5.6 is satisfied by the v given in equation 5.7 when $x \leq C_1$.

If $x > C_1$ then $C_1 - x < 0$ so $|C_1 - x| = x - C_1$ and (substituting into equation 5.7)

$$v(x) = C_2 - C_0 \left(x - C_1 \right)^{n+1} \tag{5.11}$$

which can be differentiated giving

$$\frac{\mathrm{d}v}{\mathrm{d}x} = -C_0(n+1) \left(x - C_1 \right)^n < 0 \tag{5.12}$$

So

$$\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right| = C_0(n+1) \left(x - C_1 \right)^n \tag{5.13}$$

and

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[\left|\frac{\mathrm{d}v}{\mathrm{d}x}\right|^{-\frac{n-1}{n}}\frac{\mathrm{d}v}{\mathrm{d}x}\right] = \frac{\mathrm{d}}{\mathrm{d}x}\left\{\left[C_0(n+1)\,(x-C_1)^n\right]^{-\frac{n-1}{n}}\left[-C_0(n+1)\,(x-C_1)^n\right]\right\}$$
$$= \frac{\mathrm{d}}{\mathrm{d}x}\left\{-C_0^{1/n}(n+1)^{1/n}(x-C_1)\right\} = -C_0^{1/n}(n+1)^{1/n} \quad (5.14)$$

verifying that equation 5.6 is satisfied by the v given in equation 5.7 when $x > C_1$.

Note that if $n = 1$, the absolute value in equation 5.7 is not necessary (since $|x|^2 = x^2$ for all x) and the solution $v(x)$ is a second order polynomial. This agrees with the earlier claim that the velocity curve (v versus x) for a Newtonian fluid ($n = 1$) is a parabola (a result more easily obtained by simply integrating equation 5.1 twice, assuming μ is a constant.)

The parameters C_1 and C_2 in equation 5.7 have easily understood meanings; C_2 is the maximum value obtained by $v(x)$ and C_1 is the value of x at which the maximum is attained.

5.3 Least Squares Fits

Least squares fits are used when one has data believed to be modeled by a mathematical function that contains parameters whose values are not known *a priori*. The goal is to use the data to find the values of the parameters such that the mathematical function matches the data as well as possible. For least squares fits the measure of how well the function matches the data is the sum of the squares of the residuals (described below and defined by equation 5.15). Introducing some mathematical notation, the situation is that there exists some data, consisting of a set of (x, y) pairs, $\{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$ and a function $f(x)$ which contains some parameters C_k for $k = 0, 1, 2, \ldots, m$. If the parameters are assigned values, the function $f(x)$ can be evaluated at each x_k. The difference between a calculated value $f(x_k)$ and the associated y_k is called a "residual". In a least squares fit one finds the values of the parameters which minimize the sum of the squares of the residuals

$$S = \sum_{k=1}^{N}(f(x_k) - y_k)^2 \quad (5.15)$$

For the ice along the transect of Helheim Glacier each data pair is a location along the transect together with the velocity at that location. If the ice is assumed to be

a Newtonian fluid the appropriate function to use as a model of the flow is a second order polynomial

$$f(x) = C_2 x^2 + C_1 x + C_0 \tag{5.16}$$

and goal is to find the values of C_0, C_1, and C_2 that minimize S defined by equation 5.15. The procedure used to minimize S is based on the fact that at the minimum the partial derivatives of S with respect to each of the parameters C_k is zero. Thus the solution to the minimization problem is the solution to the system of equations

$$\frac{\partial S}{\partial C_0} = 0 \tag{5.17}$$

$$\frac{\partial S}{\partial C_1} = 0 \tag{5.18}$$

$$\frac{\partial S}{\partial C_2} = 0 \tag{5.19}$$

When $f(x)$ is a polynomial (as in equation 5.16) these equations are linear algebraic equations that can be solved directly. This was done to generate the fit in figure 4.7.

A non-Newtonian model of the ice incorporating Glen's flow law is given by the solution to the differential equation in section 5.2 (equation 5.7). This model can be used in place of the polynomial above

$$f(x) = C_2 - C_0 \left| C_1 - x \right|^{n+1} \tag{5.20}$$

Once again, the values of C_0, C_1, and C_2 that minimize the sum of the squares of the residuals (S in equation 5.15) can be found by solving the system of equations 5.17, 5.18, and 5.19. But with $f(x)$ given by equation 5.20 this requires finding an approximate solution to a set of nonlinear algebraic equations using an iterative technique. In the *Python* script used to generate the results in the following section, the *fsolve* function from the *scipy.optimize* package was used to find the approximate solution. Alternately, the BFGS algorithm used in chapter 4 can be used to solve the problem of minimizing S.

5.4 Results

Least squares fits of equation 5.7 (*i.e.* equation 5.20) to the 21 data points around the location of maximum velocity on the transect of Helheim Glacier indicated by the

dashed red line in figure 2.1 were performed for the velocity data from 2 September 2010 to 13 September 2010 (plotted in figure 2.2) using values of n from 0.5 to 3.5. The resulting velocity curves for selected values of n are shown in figure 5.1. The fit in panel (b) of figure 5.1, representing a Newtonian fluid, is the same parabola shown in figure 4.7. A trend can be seen in figure 5.1: For small values of n (such as $n = 0.5$ in panel (a)) the fit is too “pointy” at its maximum to match the data and as n is increased the fit becomes “flatter” at its maximum until, for large values of n (such as $n = 3$ in panel (d)) the fit is too flat to match the data.

The minimal sum of the squares of the residuals (equation 5.15 evaluated using the parameter values of the least squares fit) provides a measure of the quality of a least squares fit, with smaller values indicating better fits. In this thesis a measure derived from the minimal sum of the squares of the residuals, an average size of the residuals, is reported

$$\overline{E} = \sqrt{\frac{S}{N}} = \sqrt{\frac{\sum_{k=1}^{N}(f(x_k) - y_k)^2}{N}} \tag{5.21}$$

(The definition of $\overline{E}$ resembles that of the standard deviation which is an average of the size of the deviations from the mean.) The notation $\overline{E}$ is meant to suggest an average error in the curve fitted to the data. A fit with a large value of $\overline{E}$ has more error than a fit with a small value of $\overline{E}$; small values of $\overline{E}$ are desirable. Values of $\overline{E}$ are included for each least squares fit in figure 5.1. The measure of error, $\overline{E}$, was also calculated for other fits with prescribed values of the exponent (n) in Glen’s flow law over the entire range $n = 0.5$ to $n = 3.5$ and are plotted versus n in figure 5.2. The plot in figure 5.2 reflects the trend described earlier; small values of n, near $n = 0.5$ result in high values of $\overline{E}$ (because the fit is too “pointy”) and large values of n, near $n = 3.5$, also result in high values of $\overline{E}$ (because the fit is too “flat”). In between, a minimal value of $\overline{E}$ is achieved at $n = 1.6$. Because it is the best fit, the velocity curve for this value was included in figure 5.1. Note that this value of the exponent in Glen’s flow law, $n = 1.6$, agrees with that found in Chapter 2.

The data used so far in this chapter (and plotted as red curves in figure 5.1) represents the period from 2 September 2010 to 13 September 2010. As mentioned in Chapter 2, the United States’ National Aeronautics and Space Administration (NASA) Making Earth System Data Records for Use in Research Environments (MEaSUREs) program also provides similar velocity data for Helhiem Glacier for other time periods

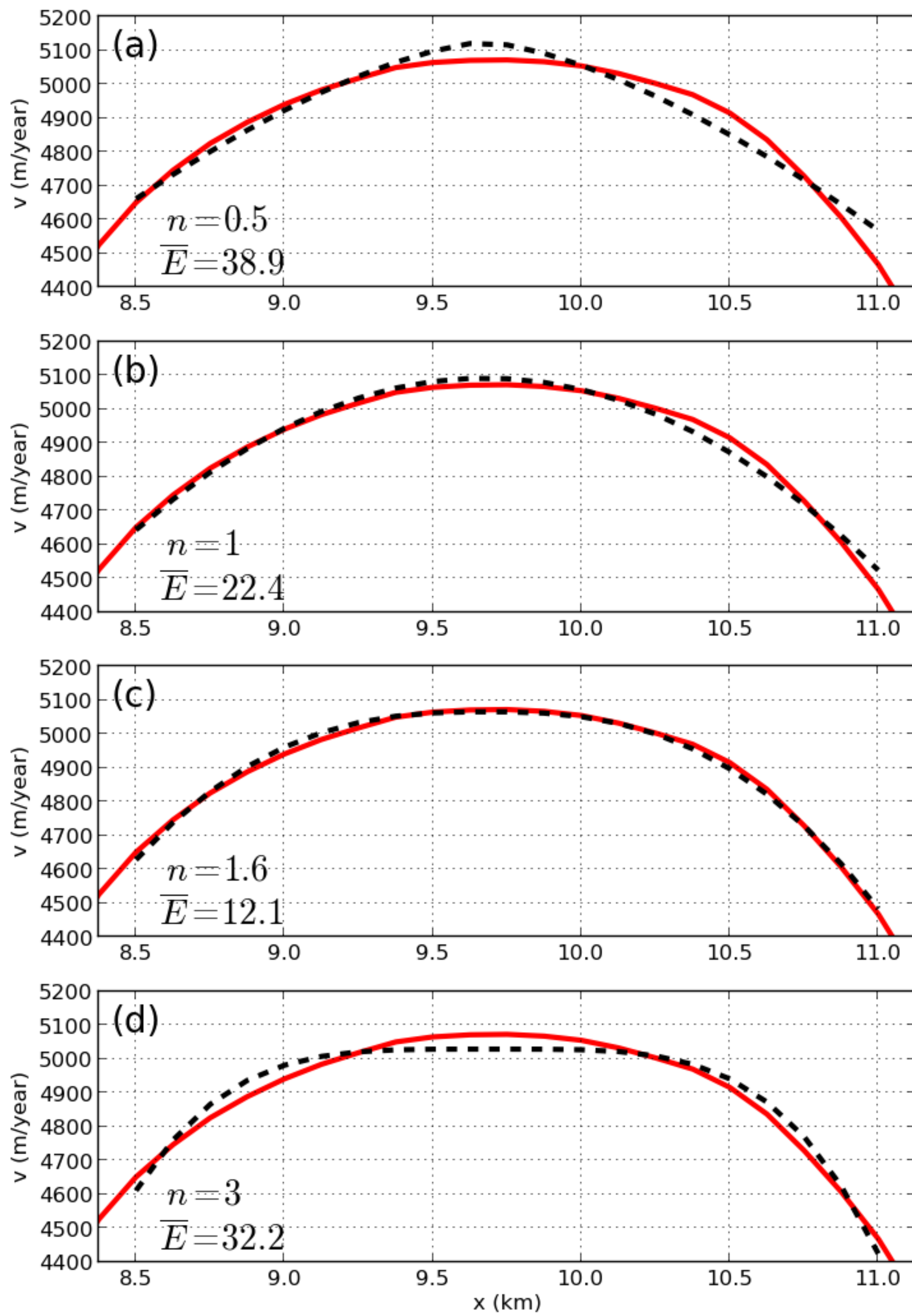

Figure 5.1: Least squares fits of $v(x) = C_2 - C_0 |C_1 - x|^{n+1}$ (equation 5.7) to the 21 data points around the location of maximum velocity on the transect of Helheim Glacier for Glen's flow exponents $n = 0.5$, $n = 1$, $n = 1.6$, and $n = 3$. In each part the solid red curve is the velocity along the transect of Helheim Glacier and the dashed black curve is the least squares fit to the data. $\overline{E}$ is a measure of the error in the fit (see equation 5.21).

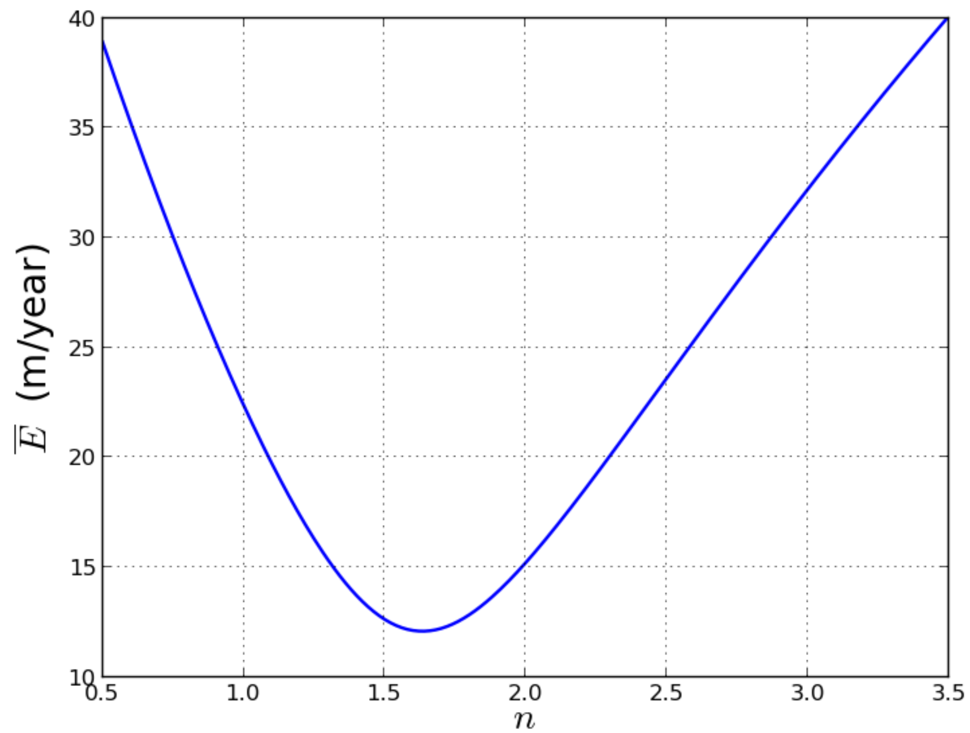

Figure 5.2: A measure of the error in the least squares fits, $\overline{E}$ (defined in equation 5.21), is plotted versus the prescribed exponent in Glen's flow law n for least squares fits like those shown in figure 5.1.

[11]. The analysis described in this chapter was applied, using the same transect, for each of these other time periods. Results of the analysis, specifically the minimal value of $\overline{E}$ and the value of n at which it was obtained, for all of the time periods for which data was available are given in table 5.1.

Table 5.1: The minimal value of $\overline{E}$ (a measure of the error in the fit) and the value of n (the exponent in Glen's flow law) at which it was obtained are reported for least squares fits of $v(x) = C_2 - C_0\,|C_1 - x|^{n+1}$ (equation 5.7) to data for all time periods for which NASA MEaSUREs data for Helheim Glacier was available. Bold font identifies the data set presented in detail in this thesis.

Data set Start date - End date	n	$\overline{E}$
27 Jan 09 - 07 Feb 09	1.70	15.6
25 Apr 09 - 06 May 09	1.61	15.2
19 Jun 09 - 30 Jun 09	1.23	13.6
24 Aug 09 - 04 Sep 09	1.69	19.2
20 Nov 09 - 01 Dec 09	1.59	8.1
23 Apr 10 - 04 May 10	1.54	13.7
09 Jul 10 - 20 Jul 10	1.62	17.1
02 Sep 10 - 13 Sep 10	**1.63**	**12.1**
18 Nov 10 - 29 Nov 10	1.57	14.9
08 Mar 11 - 19 Mar 11	1.51	14.7
04 Jun 11 - 15 Jun 11	1.10	21.7
07 Jul 11 - 18 Jul 11	1.48	10.5
09 Aug 11 - 20 Aug 11	1.32	9.2
16 Nov 11 - 27 Nov 11	1.84	12.8
Mean	1.53	14.2
Standard deviation	0.19	3.5

Chapter 6

Conclusion

A recapitulation of the methods used and conclusions reached in the thesis is given followed by a brief discussion of applying the methods to glaciers other than Helheim.

6.1 Summary

Measured surface velocities were used to investigate the variation of viscosity along a transect of Helheim Glacier, located on the south-east coast of Greenland. The analysis is based on an ordinary differential equation derived from the Stokes equations using the assumption that flow varies only in the transverse direction; variations along the flow and perpendicular to the surface of the ice are assumed to be negligible. This assumption is believed to be realistic near the center of the transect of the glacier but may not be appropriate near its edges. Thus the analysis and results focus on the fast moving portion of the glacier between the shear margins.

Using the mathematical model, the value of the exponent in Glen's flow law (a widely used constitutive equation for ice rheology) can be estimated from the measured velocities. Three different numerical approaches were used to perform this estimation:

1. Finite differences were used in a direct approach.
2. A finite element model was used in an approach involving an inverse problem.
3. Nonlinear least squares fits of the data were used.

Temperature variations were not considered.

Most numerical simulations of ice sheets use a value of three for the exponent in Glen's flow law. The results of the analysis in this thesis suggest that the behavior of ice in Helheim Glacier would be more accurately modeled using a smaller value. If true, this conclusion probably also applies elsewhere.

The best estimate of the exponent in Glen's flow law using the direct approach approximating derivatives with finite differences is $n = 1.6$. This value was confirmed using the nonlinear least squares fits of the data. Goldsby and Kohlstedt [14] describe a "superplastic" flow regime of ice for which they find an exponent of 1.8 (from laboratory experiments) and state that they believe that this regime is appropriate for glaciers and ice sheets. Our value, 1.6, is close to Goldsby and Kohlstedt's value, 1.8. The approach using a finite element model to solve an inverse problem indicates that the exponent in Glen's flow law for the ice near the center of the transect is 1.0; that is, the fast moving ice behaves as a Newtonian fluid. All three approaches indicate that the amount of "shear thinning" is less than that predicted by the widely used value $n = 3$.

6.2 Other Glaciers

The analysis and results in the thesis were presented for Helheim Glacier only. The MEaSUREs data set available for download from the National Snow and Ice Data Center also contains velocity data for most of the other outlet glaciers of the Greenland Ice Sheet. The data is organized into 45 study sites (one of which features Helheim Glacier) for dates from January 2009 to November 2011. A cursory examination of all of the study sites was undertaken, applying the first and third approaches listed above (a direct approach using finite differences, and nonlinear least squares fits of the data) to various transects of represented glaciers.

The mathematical model relies on the assumption that the ice flows in a straight line with no variation in velocity in the direction of flow. For many of the glaciers, finding a transect where these assumptions were even approximately met was impossible. The analysis also requires that there is no variation in velocity in the direction perpendicular to the surface of the ice. This assumption will not hold if basal traction is appreciable and is probably not appropriate for slower moving glaciers.

If the assumptions in the mathematical model incorporating Glen's flow law (see Chapter 5) are met, the velocity along the transect will be given by equation 5.7

(repeated below)

$$v(x) = C_2 - C_0 \left| C_1 - x \right|^{n+1} \tag{6.1}$$

This function has a single maximum (at $x = C_1$). For many transects of Greenland outlet glaciers the velocity has two or more local maxima indicating that the assumptions are not met. Other transects feature very "pointed" velocity curves such that the best least squares fit of equation 6.1 to the fastest part of the velocity curve is that with $n \approx 0$.

In the cursory examination of all of the study sites no glaciers were identified for which the analysis undertaken produced as consistent results as Helheim Glacier. By "consistent" I mean not only that the two approaches produced similar estimates of the exponent in Glen's flow law that were approximately equal for different time periods, but also that nearby transects also provided similar results.

The values for the exponent in Glen's flow law, n, generated by analysing transects of various glaciers ranged from zero to greater than three. The value $n = 1.6$ which was produced for Helheim Glacier was reproduced for some other glaciers but certainly not for all. The fact that the value of n produced by the analysis is dramatically different for different glaciers and even different for different transects of the same glacier is interpreted as meaning that the assumptions of the model are not satisfied by most glaciers.

Appendix A

Adjoint Methods

An introduction to Adjoint Methods, which are used to find the gradient of an objective function as required by optimization algorithms. Examples are included, culminating in a data assimilation problem from glaciology.

A.1 Introduction[1]

Suppose you have a problem involving a collection of parameters $\mathbf{p}$ (bold font indicates a vector) whose solution is $\mathbf{u} = \mathbf{F}(\mathbf{p})$. You want to find the parameters $\mathbf{p}$ that minimize (or maximize) a given (scalar) function $g(\mathbf{u})$. Since $\mathbf{u}$ is a function of the parameters $\mathbf{p}$, we can think of g as a function of $\mathbf{p}$. (Formally we could introduce a new function $\tilde{g}(\mathbf{p}) = g(\mathbf{F}(\mathbf{p}))$ but we believe the presentation is clearer, albeit less rigorous, without it.) Most efficient algorithms used to optimize g require knowledge of $\frac{\partial g}{\partial p_k}$ for each of the parameters p_k (components of $\mathbf{p}$.) Adjoint methods can be used to find these derivatives. We write this collection of partial derivatives as a vector $\frac{\partial g}{\partial \mathbf{p}} = \left(\frac{\partial g}{\partial p_1}, \frac{\partial g}{\partial p_2}, \cdots, \frac{\partial g}{\partial p_m}\right)$. In the matrix multiplications in this introductory section, this and other vectors are represented by column matrices.

The Jacobian matrix of the function $\mathbf{F}$ that maps the parameters $\mathbf{p}$ to the solution

[1]Most of the ideas in this introductory section are taken from section 3 of reference [15].

$\mathbf{u}$ is defined as

$$\frac{\partial \mathbf{u}}{\partial \mathbf{p}} = \begin{bmatrix} \frac{\partial u_1}{\partial p_1} & \frac{\partial u_1}{\partial p_2} & \cdots & \frac{\partial u_1}{\partial p_m} \\ \frac{\partial u_2}{\partial p_1} & \frac{\partial u_2}{\partial p_2} & \cdots & \frac{\partial u_2}{\partial p_m} \\ \vdots & & \ddots & \vdots \\ \frac{\partial u_n}{\partial p_1} & \frac{\partial u_n}{\partial p_2} & \cdots & \frac{\partial u_n}{\partial p_m} \end{bmatrix}$$

This Jacobian matrix can be used to approximate changes in the solution $\mathbf{u}$ resulting from small changes in the parameters $\mathbf{p}$. For an individual component of the solution $\mathbf{u}$,

$$\Delta u_i = \frac{\partial u_i}{\partial p_1}\Delta p_1 + \frac{\partial u_i}{\partial p_2}\Delta p_2 + \cdots + \frac{\partial u_i}{\partial p_m}\Delta p_m$$

The approximate changes for all components can be written compactly (using matrix multiplication) as

$$\Delta \mathbf{u} = \frac{\partial \mathbf{u}}{\partial \mathbf{p}}\Delta \mathbf{p}$$

To find the desired derivatives $\frac{\partial g}{\partial \mathbf{p}}$, consider

$$\begin{aligned} g(\mathbf{u}) &= g(u_1, u_2, \ldots, u_n) \\ &= g(u_1(p_1, p_2, \ldots p_m),\ u_2(p_1, p_2, \ldots p_m),\ \ldots,\ u_n(p_1, p_2, \ldots p_m)) \end{aligned}$$

Using the chain rule

$$\frac{\partial g}{\partial p_k} = \frac{\partial g}{\partial u_1}\frac{\partial u_1}{\partial p_k} + \frac{\partial g}{\partial u_2}\frac{\partial u_2}{\partial p_k} + \cdots + \frac{\partial g}{\partial u_n}\frac{\partial u_n}{\partial p_k}$$

Note that the derivatives of the components of $\mathbf{u}$ with respect to p_k are the elements of the kth *column* of the Jacobian matrix $\frac{\partial \mathbf{u}}{\partial \mathbf{p}}$. We can write the desired derivatives compactly using the transpose (also called the *adjoint*[2]) of the Jacobian matrix

$$\frac{\partial g}{\partial \mathbf{p}} = \left(\frac{\partial \mathbf{u}}{\partial \mathbf{p}}\right)^T \frac{\partial g}{\partial \mathbf{u}}$$

Use of the *adjoint* of a Jacobian matrix seems to be where *Adjoint Methods* get their name.

[2]The adjoint of a matrix Z whose elements are complex numbers is the transpose of the matrix whose elements are the complex conjugates of the elements of Z. For real valued matrices, such as the Jacobian matrices in this paper, the transpose and the adjoint are the same.

The method can be illustrated by a simple example where the function $\mathbf{F}$ mapping the parameters $\mathbf{p}$ to the solution $\mathbf{u}$ is given explicitly.

Example 1:

Suppose $\mathbf{u} = \begin{bmatrix} p_1^2 + p_2 \\ p_1 p_2 \end{bmatrix}$ and $g(\mathbf{u}) = u_1 + u_2^2$. We want to find $\dfrac{\partial g}{\partial \mathbf{p}} = \begin{bmatrix} \frac{\partial g}{\partial p_1} \\ \frac{\partial g}{\partial p_2} \end{bmatrix}$.

To calculate $\frac{\partial g}{\partial \mathbf{p}}$ using the adjoint method we begin by finding the Jacobian matrix

$$\frac{\partial \mathbf{u}}{\partial \mathbf{p}} = \begin{bmatrix} 2p_1 & 1 \\ p_2 & p_1 \end{bmatrix}$$

and

$$\frac{\partial g}{\partial \mathbf{u}} = \begin{bmatrix} 1 \\ 2u_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2p_1 p_2 \end{bmatrix}$$

Then

$$\frac{\partial g}{\partial \mathbf{p}} = \left(\frac{\partial \mathbf{u}}{\partial \mathbf{p}}\right)^T \frac{\partial g}{\partial \mathbf{u}} = \begin{bmatrix} 2p_1 & p_2 \\ 1 & p_1 \end{bmatrix} \begin{bmatrix} 1 \\ 2p_1 p_2 \end{bmatrix} = \begin{bmatrix} 2p_1 + 2p_1 p_2^2 \\ 1 + 2p_1^2 p_2 \end{bmatrix}$$

To check this solution we find $\frac{\partial g}{\partial \mathbf{p}}$ directly using

$$g(\mathbf{u}) = u_1 + u_2^2 = (p_1^2 + p_2) + (p_1 p_2)^2$$

from which we find

$$\frac{\partial g}{\partial p_1} = 2p_1 + 2p_1 p_2^2$$

and

$$\frac{\partial g}{\partial p_2} = 1 + 2p_1^2 p_2$$

A.2 A Typical Situation[3]

Often the problem to be solved (whose solution is $\mathbf{u} = \mathbf{F}(\mathbf{p})$) is expressed as a system of n equations, $\mathbf{f}(\mathbf{u}, \mathbf{p}) = \mathbf{0}$. For this problem we still have a function $g(\mathbf{u})$ for which we want to find the optimal parameters $\mathbf{p}$ and thus need $\frac{\partial g}{\partial \mathbf{p}} = \begin{bmatrix} \frac{\partial g}{\partial p_1} & \frac{\partial g}{\partial p_2} & \cdots & \frac{\partial g}{\partial p_m} \end{bmatrix}$. To

[3]Most of the ideas in section A.2 are taken from section 8.7 of reference [16].

be consistent with the matrix equations below, we have now written this collection of derivatives as a row matrix. We can obtain the elements of this row matrix using the chain rule (as in section A.1)

$$\frac{\partial g}{\partial p_k} = \frac{\partial g}{\partial u_1}\frac{\partial u_1}{\partial p_k} + \frac{\partial g}{\partial u_2}\frac{\partial u_2}{\partial p_k} + \cdots + \frac{\partial g}{\partial u_n}\frac{\partial u_n}{\partial p_k}$$

and the entire collection of derivatives can be written as

$$\frac{\partial g}{\partial \mathbf{p}} = \frac{\partial g}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \mathbf{p}}$$

The factors $\frac{\partial g}{\partial u_i}$ are obtained by differentiating $g(\mathbf{u})$ but the factors $\frac{\partial u_i}{\partial p_k}$ must be determined from $\mathbf{f}(\mathbf{u}, \mathbf{p}) = \mathbf{0}$. If we take the total derivative of both sides of the equation $f_i(\mathbf{u}, \mathbf{p}) = 0$ with respect to p_k we obtain

$$\frac{df_i}{dp_k} = \frac{\partial f_i}{\partial u_1}\frac{\partial u_1}{\partial p_k} + \frac{\partial f_i}{\partial u_2}\frac{\partial u_2}{\partial p_k} + \cdots + \frac{\partial f_i}{\partial u_n}\frac{\partial u_n}{\partial p_k} + \frac{\partial f_i}{\partial p_k} = 0$$

Doing this for every combination of the n elements of $\mathbf{u}$ and the m elements of $\mathbf{p}$ results in nm equations that can be represented compactly as

$$\frac{\partial \mathbf{f}}{\partial \mathbf{u}}\frac{\partial \mathbf{u}}{\partial \mathbf{p}} + \frac{\partial \mathbf{f}}{\partial \mathbf{p}} = 0$$

That is,

$$\begin{bmatrix} \frac{\partial f_1}{\partial u_1} & \frac{\partial f_1}{\partial u_2} & \cdots & \frac{\partial f_1}{\partial u_n} \\ \frac{\partial f_2}{\partial u_1} & \ddots & & \\ \vdots & & \ddots & \vdots \\ \frac{\partial f_n}{\partial u_1} & \frac{\partial f_n}{\partial u_2} & \cdots & \frac{\partial f_n}{\partial u_n} \end{bmatrix} \begin{bmatrix} \frac{\partial u_1}{\partial p_1} & \frac{\partial u_1}{\partial p_2} & \cdots & \frac{\partial u_1}{\partial p_m} \\ \frac{\partial u_2}{\partial p_1} & \ddots & & \\ \vdots & & \ddots & \vdots \\ \frac{\partial u_n}{\partial p_1} & \frac{\partial u_n}{\partial p_2} & \cdots & \frac{\partial u_n}{\partial p_m} \end{bmatrix} + \begin{bmatrix} \frac{\partial f_1}{\partial p_1} & \frac{\partial f_1}{\partial p_2} & \cdots & \frac{\partial f_1}{\partial p_m} \\ \frac{\partial f_2}{\partial p_1} & \ddots & & \\ \vdots & & \ddots & \vdots \\ \frac{\partial f_n}{\partial p_1} & \frac{\partial f_n}{\partial p_2} & \cdots & \frac{\partial f_n}{\partial p_m} \end{bmatrix} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & \ddots & & \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

This equation can be solved for the $\frac{\partial u_i}{\partial p_k}$ factors

$$\frac{\partial \mathbf{u}}{\partial \mathbf{p}} = -\left(\frac{\partial \mathbf{f}}{\partial \mathbf{u}}\right)^{-1}\frac{\partial \mathbf{f}}{\partial \mathbf{p}}$$

needed to obtain our goal

$$\frac{\partial g}{\partial \mathbf{p}} = \frac{\partial g}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{p}} = \frac{\partial g}{\partial \mathbf{u}} \left(- \left(\frac{\partial \mathbf{f}}{\partial \mathbf{u}} \right)^{-1} \frac{\partial \mathbf{f}}{\partial \mathbf{p}} \right)$$

Here $\frac{\partial g}{\partial \mathbf{u}}$ is a $1 \times n$ (row) matrix, $\frac{\partial \mathbf{f}}{\partial \mathbf{u}}$ is $n \times n$, and $\frac{\partial \mathbf{f}}{\partial \mathbf{p}}$ is $n \times m$. To avoid multiplying the two large matrices we perform the multiplication in the order

$$\frac{\partial g}{\partial \mathbf{p}} = - \left(\frac{\partial g}{\partial \mathbf{u}} \left(\frac{\partial \mathbf{f}}{\partial \mathbf{u}} \right)^{-1} \right) \frac{\partial \mathbf{f}}{\partial \mathbf{p}}$$

with the product $\lambda^T = \frac{\partial g}{\partial \mathbf{u}} \left(\frac{\partial \mathbf{f}}{\partial \mathbf{u}} \right)^{-1}$ found by solving the *adjoint problem*

$$\left(\frac{\partial \mathbf{f}}{\partial \mathbf{u}} \right)^T \lambda = \left(\frac{\partial g}{\partial \mathbf{u}} \right)^T$$

Note that $\frac{\partial \mathbf{f}}{\partial \mathbf{u}}$ is a Jacobian matrix; once again obtaining the desired partial derivatives ($\frac{\partial g}{\partial \mathbf{p}}$) uses the transpose (that is, *adjoint*) of a Jacobian matrix. Also note that this Jacobian matrix is the one used in Newton's method to (approximately) solve $\mathbf{f}(\mathbf{u}, \mathbf{p}) = \mathbf{0}$ (with fixed $\mathbf{p}$) iteratively.

Example 2:

Suppose

$$\begin{aligned} f_1(u_1, u_2, p_1, p_2) &= u_1 + u_2 + p_1 \\ f_2(u_1, u_2, p_1, p_2) &= u_1^3 - u_2 + p_2 \\ g(u_1, u_2) &= u_1^2 + u_2^2 \end{aligned}$$

The system of equations $f_1(u_1, u_2, p_1, p_2) = 0$ and $f_2(u_1, u_2, p_1, p_2) = 0$ has a solution, (u_1, u_2), that depends on the parameters p_1 and p_2. Changing the values of the parameters p_1 and p_2 causes the values of the variables u_1 and u_2 to change and thus the value of $g(u_1, u_2)$ also changes. We want to find

$$\frac{\partial g}{\partial \mathbf{p}} = \begin{bmatrix} \frac{\partial g}{\partial p_1} & \frac{\partial g}{\partial p_2} \end{bmatrix}$$

To calculate $\frac{\partial g}{\partial \mathbf{p}}$ using the adjoint method we begin by finding the Jacobian matrix

$$\frac{\partial \mathbf{f}}{\partial \mathbf{u}} = \begin{bmatrix} 1 & 1 \\ 3u_1^2 & -1 \end{bmatrix}$$

along with

$$\frac{\partial \mathbf{f}}{\partial \mathbf{p}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

and

$$\frac{\partial g}{\partial \mathbf{u}} = \begin{bmatrix} 2u_1 & 2u_2 \end{bmatrix}$$

Next we write the adjoint problem $\left(\frac{\partial \mathbf{f}}{\partial \mathbf{u}}\right)^T \lambda = \left(\frac{\partial g}{\partial \mathbf{u}}\right)^T$

$$\begin{bmatrix} 1 & 3u_1^2 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} 2u_1 \\ 2u_2 \end{bmatrix}$$

and solve it to find

$$\begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} 2u_2 + \frac{2u_1 - 2u_2}{3u_1^2+1} \\ \frac{2u_1 - 2u_2}{3u_1^2+1} \end{bmatrix}$$

The desired derivatives are now found from

$$\frac{\partial g}{\partial \mathbf{p}} = -\left(\frac{\partial g}{\partial \mathbf{u}} \left(\frac{\partial \mathbf{f}}{\partial \mathbf{u}}\right)^{-1}\right) \frac{\partial \mathbf{f}}{\partial \mathbf{p}} = -\lambda^T \frac{\partial \mathbf{f}}{\partial \mathbf{p}}$$

$$= -\begin{bmatrix} 2u_2 + \frac{2u_1 - 2u_2}{3u_1^2+1} & \frac{2u_1 - 2u_2}{3u_1^2+1} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} -2u_2 - \frac{2u_1 - 2u_2}{3u_1^2+1} & -\frac{2u_1 - 2u_2}{3u_1^2+1} \end{bmatrix}$$

To appreciate the meaning of these derivatives we consider a couple of specific value pairs for the parameters p_1 and p_2. First, if $p_1 = 0$ and $p_2 = 0$, the equation $\mathbf{f}(\mathbf{u}, \mathbf{p}) = \mathbf{0}$ has solution $\mathbf{u} = \mathbf{0}$ so $g(\mathbf{u}) = 0$. Since

$$g(u_1, u_2) = u_1^2 + u_2^2 \geq 0 \quad \text{(for any } u_1 \text{ and } u_2\text{)}$$

$g(\mathbf{u}) = 0$ must be a global minimum so we expect $\frac{\partial g}{\partial \mathbf{p}} = \mathbf{0}$. Indeed, substituting $u_1 = 0$ and $u_2 = 0$ into our expression

$$\frac{\partial g}{\partial \mathbf{p}} = \begin{bmatrix} -2u_2 - \frac{2u_1 - 2u_2}{3u_1^2+1} & -\frac{2u_1 - 2u_2}{3u_1^2+1} \end{bmatrix}$$

confirms this.

To gain further appreciation of $\frac{\partial g}{\partial \mathbf{p}}$ we consider a graphical representation of our example problem. If we plot solutions of

$$f_1(u_1, u_2, p_1, p_2) = u_1 + u_2 + p_1 = 0$$

in the (u_1, u_2) plane we obtain a line with slope -1 and vertical intercept $-p_1$. If we plot solutions of

$$f_2(u_1, u_2, p_1, p_2) = u_1^3 - u_2 + p_2 = 0$$

in the (u_1, u_2) plane we obtain a cubic curve that crosses the vertical axis at p_2 with slope 0. The solution to the system of equations $\mathbf{f}(\mathbf{u}, \mathbf{p}) = \mathbf{0}$ is represented by the point where the line and the cubic curve cross. The function g gives the square of the distance from the origin so circles centered on the origin represent contours of constant g. Curves for $p_1 = -2$, $p_2 = 0$ and $g = 2$ are shown in figure A.1. As can be seen from figure A.1, for this value of $\mathbf{p}$, the solution of $\mathbf{f}(\mathbf{u}, \mathbf{p}) = \mathbf{0}$ is $\mathbf{u} = (1, 1)$. If we increase p_1, the line with slope 1 will move down in the figure so the solution point will follow the cubic curve, decreasing its distance from the origin so g decreases. On the other hand, if we increase p_2, the cubic curve will move up so the solution point will follow the line with slope -1 which is tangent to the (dashed) circle representing a contour of constant g. Thus we expect $\frac{\partial g}{\partial p_1} < 0$ and $\frac{\partial g}{\partial p_2} = 0$. Substituting the solution $u_1 = 1$ and $u_2 = 1$ into our expression

$$\frac{\partial g}{\partial \mathbf{p}} = \begin{bmatrix} -2u_2 - \frac{2u_1 - 2u_2}{3u_1^2 + 1} & -\frac{2u_1 - 2u_2}{3u_1^2 + 1} \end{bmatrix}$$

gives $\frac{\partial g}{\partial \mathbf{p}} = \begin{bmatrix} -2 & 0 \end{bmatrix}$ confirming this. The reader is encouraged to use the graphical representation to find other parameter pairs where one of the components of $\frac{\partial g}{\partial \mathbf{p}}$ is zero and use the expression for $\frac{\partial g}{\partial \mathbf{p}}$ to confirm their findings.

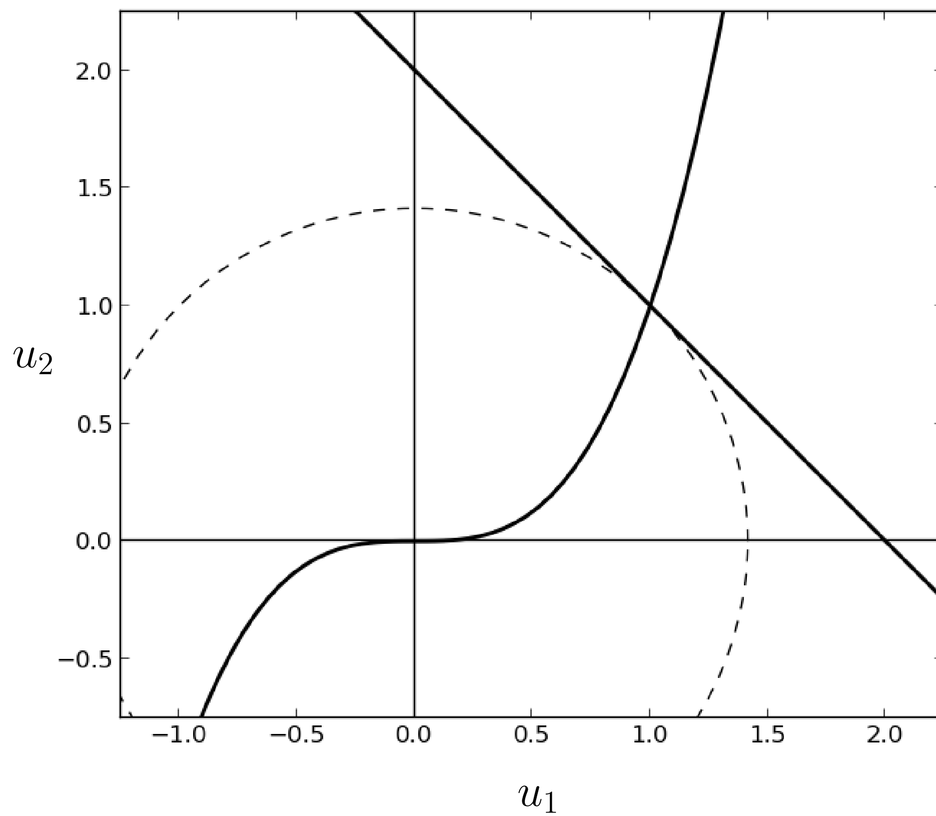

Figure A.1: Curves for the problem presented in Example 2 with $p_1 = -2$, $p_2 = 0$, and $g(\mathbf{u}) = 2$.

A.3 A Numerical Example

In this section we use finite differences to approximate the boundary value problem

$$c_2 u'' + c_1 u' + c_0 u = p(x) \quad \text{for} \quad 0 < x < 1, \quad \text{and} \quad u(0) = a_0, \quad u(1) = a_1$$

where

$$p(x) = p_0 + p_1 x + p_2 x^2 \quad \text{(a second order polynomial).}$$

After computing an approximate solution for particular values of the parameters c_2, c_1, c_0, p_0, p_1, p_2, a_0, and a_1, we use the adjoint method to calculate the rate of change of a function of the solution, $g(u)$, as we vary these parameters. Two different functions are considered: (1) g is the value of u at the midpoint of the domain, $g(u) = u(\frac{1}{2})$, and (2) g is the average value of u, $g(u) = \int_0^1 u(x)dx$.

To apply the finite difference method to our differential equation we divide the domain $[0, 1]$ into n subintervals each with length $\Delta x = \frac{1}{n}$. The endpoints of these subintervals are $x_0,\ x_1,\ \ldots,\ x_n$ where $x_k = k\Delta x$. We denote approximate values of u at each of these points as u_k. The first derivative u' at x_k is approximated by

$$u_k{}' = \frac{u_{k+1} - u_{k-1}}{2\Delta x}$$

The second derivative u'' at x_k is approximated by

$$u_k{}'' = \frac{u_{k-1} - 2u_k + u_{k+1}}{\Delta x^2}$$

Substituting these approximations into the differential equation gives

$$c_2 \left(\frac{u_{k-1} - 2u_k + u_{k+1}}{\Delta x^2} \right) + c_1 \left(\frac{u_{k+1} - u_{k-1}}{2\Delta x} \right) + c_0 u_k = p\left(x_k\right)$$

which can be rearranged to

$$\left(\frac{c_2}{\Delta x^2} - \frac{c_1}{2\Delta x} \right) u_{k-1} + \left(-\frac{2c_2}{\Delta x^2} + c_0 \right) u_k + \left(\frac{c_2}{\Delta x^2} + \frac{c_1}{2\Delta x} \right) u_{k+1} = p\left(x_k\right)$$

a linear algebraic equation that can be written for each of the interior points $k = 1, 2, \ldots, n-1$. At the endpoints of the domain we assert $u_0 = a_0$ and $u_n = a_1$. This collection of equations can be written compactly as $\mathbf{A}\mathbf{u} = \mathbf{b}$ where $\mathbf{A}$ is a tridiagonal

matrix, and $\mathbf{u}$ and $\mathbf{b}$ are column matrices. The matrix equation can be solved (using Gaussian elimination and back substitution, for example) for $\mathbf{u}$ giving an approximation to the solution of the boundary value problem.

Following the procedure in the preceding paragraph for the (somewhat arbitrary) parameter values $c_2 = 1$, $c_1 = -2$, $c_0 = 1$, $p_0 = 1$, $p_1 = 1$, $p_2 = -5$, $a_0 = 0$, and $a_1 = 0$ generates the approximation shown in figure A.2.

We can use the adjoint method to find the rate of change of a function of $\mathbf{u}$ with respect to the parameters c_2, c_1, c_0, p_0, p_1, p_2, a_0, and a_1. We consider first the simple function $g(\mathbf{u}) = u_{n/2} \approx u\left(\frac{1}{2}\right)$.

To utilize the procedure described in section A.2, the system of equations represented by the matrix equation $\mathbf{Au} = \mathbf{b}$ is written in the form $\mathbf{f}(\mathbf{u}, \mathbf{p}) = \mathbf{0}$ by subtracting the right hand sides from both sides each equation. Here $\mathbf{p} = [c_2\ c_1\ c_0\ p_0\ p_1\ p_2\ a_0\ a_1]$. To find $\frac{\partial g}{\partial \mathbf{p}}$ we need three matrices:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{u}} = \mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \left(\frac{c_2}{\Delta x^2} - \frac{c_1}{2\Delta x}\right) & \left(-\frac{2c_2}{\Delta x^2} + c_0\right) & \left(\frac{c_2}{\Delta x^2} + \frac{c_1}{2\Delta x}\right) & 0 & 0 & \cdots & 0 \\ 0 & \left(\frac{c_2}{\Delta x^2} - \frac{c_1}{2\Delta x}\right) & \left(-\frac{2c_2}{\Delta x^2} + c_0\right) & \left(\frac{c_2}{\Delta x^2} + \frac{c_1}{2\Delta x}\right) & 0 & \cdots & 0 \\ \vdots & & & \ddots & & & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 \end{bmatrix}$$

$$\frac{\partial \mathbf{f}}{\partial \mathbf{p}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ \frac{u_0 - 2u_1 + u_2}{\Delta x^2} & \frac{u_2 - u_0}{2\Delta x} & u_1 & -1 & -x_1 & -x_1^2 & 0 & 0 \\ \frac{u_1 - 2u_2 + u_3}{\Delta x^2} & \frac{u_3 - u_1}{2\Delta x} & u_2 & -1 & -x_2 & -x_2^2 & 0 & 0 \\ & & & \vdots & & & & \\ \frac{u_{n-2} - 2u_{n-1} + u_n}{\Delta x^2} & \frac{u_n - u_{n-2}}{2\Delta x} & u_{n-1} & -1 & -x_{n-1} & -x_{n-1}^2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$

$$\frac{\partial g}{\partial \mathbf{u}} = [0\ 0\ 0\ \cdots\ 0\ 0\ 1\ 0\ 0\ \cdots\ 0\ 0\ 0]$$

After solving the adjoint problem

$$\left(\frac{\partial \mathbf{f}}{\partial \mathbf{u}}\right)^T \lambda = \left(\frac{\partial g}{\partial \mathbf{u}}\right)^T$$

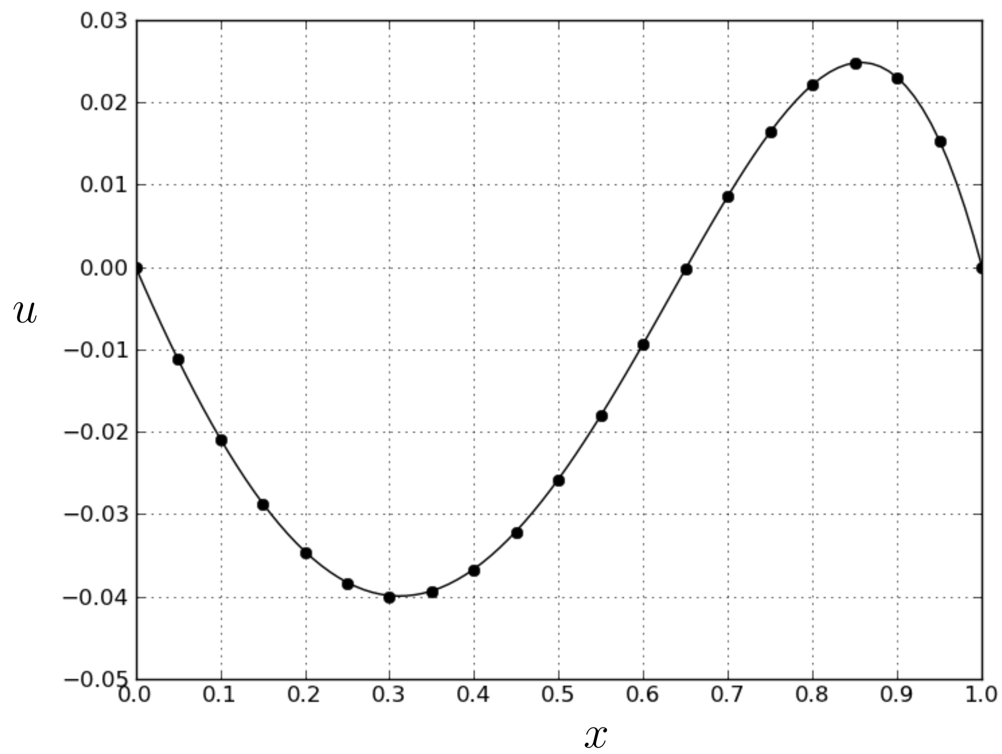

Figure A.2: Solution to the boundary value problem $u'' - 2u' + u = 1 + x - 5x^2$ for $0 < x < 1$, and $u(0) = 0$, $u(1) = 0$. The solid curve is the exact solution, the black circles are an approximate solution found using finite differences with $n = 20$.

for λ we can calculate

$$\frac{\partial g}{\partial \mathbf{p}} = -\lambda^T \frac{\partial \mathbf{f}}{\partial \mathbf{p}}$$

The results for the example problem whose solution is shown in figure A.2 are given in Table A.1.

The values of the partial derivatives found using the adjoint method were verified by perturbing the parameters one at a time. For each parameter (c_2, c_1, c_0, p_0, p_1, p_2, a_0, and a_1) an additional finite difference approximation was calculated with that parameter increased by 0.01 giving new values of $\mathbf{u}$. Calling these new values $\tilde{\mathbf{u}}$ we can approximate the partial derivative of g with respect to the perturbed parameter by $\frac{g(\tilde{\mathbf{u}})-g(\mathbf{u})}{0.01}$. As seen in Table A.1, these approximations are very close to the values found using the adjoint method.

To calculate the partial derivatives of $g(u) = \int_0^1 u(x)dx$ we use (the composite) Simpson's rule to approximate this integral,

$$g(\mathbf{u}) = \frac{\Delta x}{3}\left[u_0 + 4u_1 + 2u_2 + 4u_3 + 2u_4 + \cdots + 2u_{n-2} + 4u_{n-1} + u_n\right]$$

from which we have

$$\frac{\partial g}{\partial \mathbf{u}} = \frac{\Delta x}{3}[1 \;\; 4 \;\; 2 \;\; 4 \;\; 2 \;\; \cdots \;\; 2 \;\; 4 \;\; 1]$$

Again, $\frac{\partial g}{\partial \mathbf{p}} = -\lambda^T \frac{\partial \mathbf{f}}{\partial \mathbf{p}}$ where λ is the solution of the adjoint problem $\left(\frac{\partial \mathbf{f}}{\partial \mathbf{u}}\right)^T \lambda = \left(\frac{\partial g}{\partial \mathbf{u}}\right)^T$. Results are included in Table A.1.

Table A.1: Partial derivatives for $g(u) = u\left(\frac{1}{2}\right)$ and $g(u) = \int_0^1 u(x)dx$ using a finite difference approximation to the boundary value problem $c_2 u'' + c_1 u' + c_0 u = p_0 + p_1 x + p_2 x^2$ for $0 < x < 1$, and $u(0) = a_0$, $u(1) = a_1$ with $c_2 = 1$, $c_1 = -2$, $c_0 = 1$, $p_0 = 1$, $p_1 = 1$, $p_2 = -5$, $a_0 = 0$, and $a_1 = 0$.

	$g(u) = u\left(\frac{1}{2}\right)$		$g(u) = \int_0^1 u(x)dx$	
Partial derivative	Value from adjoint method	Numerical verification	Value from adjoint method	Numerical verification
$\frac{\partial g}{\partial c_2}$	0.04372056	0.04315389	0.02133546	0.02104484
$\frac{\partial g}{\partial c_1}$	0.00762168	0.00763637	0.00424091	0.00424846
$\frac{\partial g}{\partial c_0}$	-0.00262876	-0.00263144	-0.00157897	-0.00158068
$\frac{\partial g}{\partial p_0}$	-0.12775518	-0.12775518	-0.08625040	-0.08625040
$\frac{\partial g}{\partial p_1}$	-0.05862544	-0.05862544	-0.04027710	-0.04027710
$\frac{\partial g}{\partial p_2}$	-0.03210644	-0.03210644	-0.02305057	-0.02305057
$\frac{\partial g}{\partial a_0}$	0.82464012	0.82464012	0.71847410	0.71847410
$\frac{\partial g}{\partial a_1}$	0.30311507	0.30311507	0.36777630	0.36777630

A.4 A Data Assimilation Example From Glaciology

Mathematical models of glaciers typically include parameters which are difficult to estimate directly from field measurements. For example, some type of frictional coefficient is required at the base of the glacier where the ice rests upon the earth. The basal traction (described mathematically using this frictional coefficient) affects the velocity of the ice throughout the glacier. Thus it is plausible to use the more easily measured velocity at the upper surface of the glacier to determine the frictional coefficient. In this concluding section we present an example problem using a simplified geometry to demonstrate the use of the adjoint method to accomplish this.

The example problem is related to a benchmark problem called "Experiment D" from the Ice Sheet Model Intercomparison Project for Higher Order Models (ISMIP-HOM) [17]. In ISMIP-HOM Experiment D, a sheet of ice with uniform thickness 1000 meters is resting on a plane inclined at an angle of $0.1°$. A domain of length 20 km with periodic boundary conditions at each end is considered. A mathematical model is used to predict the velocity of the ice, which varies not only throughout the thickness of the ice but also along the length of the domain due to variations in basal traction. There is no variation in the horizontal direction transverse to the inclination.

In ISMIP-HOM Experiment D, the basal traction is prescribed and ice velocities are calculated. We call this a "forward problem." Here we are interested in a related "inverse problem": given the surface velocity, find the basal traction.

The model used is a finite element approximation of Stokes' equation[4] along with conservation of mass. Basal traction is modeled using the boundary condition

$$\mu \frac{\partial u}{\partial y} = \beta^2 u$$

where μ is the viscosity (which varies according to Glen's flow law[5]), u is the velocity component along the incline, and β^2 is a frictional coefficient (y is the coordinate

[4]Stokes' equation represents Newton's second law, $\sum \mathbf{F} = m\mathbf{a}$, for a viscous fluid when the inertial term, $m\mathbf{a}$, is negligible.

[5]Glen's flow law is a nonlinear constitutive equation used to model the relationship between stress and strain rates in ice. This relationship is temperature dependent but the model used here assumes that the ice is isothermal.

direction perpendicular to the inclined plane supporting the ice sheet.) In ISMIP-HOM experiment D

$$\beta^2 = 1000 + 1000 \sin\left(\frac{2\pi x}{L}\right)$$

where L is the length of the domain (20 km for the problem presented here.) This frictional coefficient is plotted as a heavy black line in the lower left portion of figure A.3. The upper left portion of figure A.3 shows the velocity component u at the (top) surface of the ice for a solution of the "forward" problem (using $\beta^2 = 1000 + 1000 \sin\left(\frac{2\pi x}{L}\right)$) plotted as a heavy black line.

Now, for the inverse problem. Suppose we are given the surface velocity component u plotted as a heavy black line in the upper left portion of figure A.3. We hope to use this information to discover the basal friction coefficient β^2 (plotted as a heavy black line in the lower left portion of figure A.3.) To do so we pose the problem as an optimization problem: Find β^2 such that the surface velocity component u found by solving the "forward" problem minimizes the error given by:

$$g(u) = \int_0^L (u - u_d)^2 \, dx$$

where u_d is the desired surface velocity. To discretize the function β^2 a trigonometric expansion is used,

$$\beta^2 = p_0 + \sum_{k=1}^{n/2} p_{2k-1} \sin\left(\frac{2\pi kx}{L}\right) + p_{2k} \cos\left(\frac{2\pi kx}{L}\right)$$

A variety of algorithms for solving optimization problems exist [18]. Here we choose the Broyden-Fletcher-Goldfarb-Shano (BFGS) algorithm which uses partial derivatives of the objective function ($g(u)$) with respect to the parameters (p_k's) that can be varied. These partial derivatives ($\frac{\partial g}{\partial p_k}$) are *exactly* what the adjoint method described in the previous sections can provide. Starting with a "guess" for β^2, an approximate solution to the inverse problem is obtained by repeating the following steps:

1. Solve the "forward" problem with the current estimate of β^2.

2. Evaluate the error $g(u)$.

3. Evaluate the partial derivatives $\frac{\partial g}{\partial p_k}$ using the adjoint method.

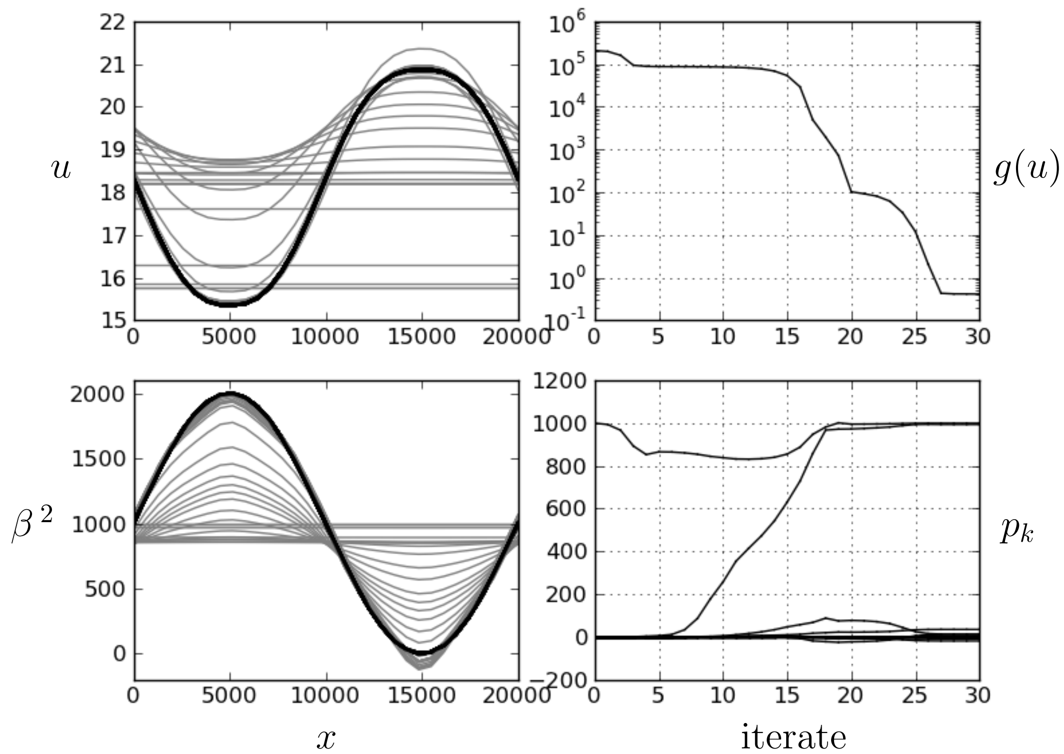

Figure A.3: Solution of an inverse problem. The component of the ice velocity in the direction parallel to the bed (u, in meters per annum) at the (top) surface of the ice is shown in the upper left. This velocity evolves towards the desired solution (the heavy black curve) as the basal traction (β^2, plotted in the lower left) is changed. The upper right shows the error ($g(u)$) decreasing as the coefficients (p_k) in the trigonometric expansion of β^2 (shown in the lower right) change.

4. Calculate an improved estimate of β^2 (that is, the p_k's) according to the BFGS algorithm.

The results of applying this process are shown in figure A.3. The initial guess was $\beta^2 = 1000$; that is, $p_0 = 1000$ and $p_k = 0$ for $1 \le k \le n$ where, here, $n = 30$. The gray curves in the left portion of figure A.3 show approximations of u and β^2 as the process progressed. The original guess of $\beta^2 = 1000$ (represented by a horizontal line in the lower left portion of figure A.3) resulted in the bottommost horizontal line in the upper left portion of figure A.3. As the estimate of β^2 improves the curves approach the desired values plotted as heavy black curves. The upper right portion of figure A.3 shows the error $g(u)$ decreasing throughout the process. Further improvement can be made by allowing more steps. The lower right portion of figure A.3 shows the evolution of the coefficient (p_k) values. The upper curve represents p_0 which starts at 1000, dips down for a while but returns to a value very near 1000. The curve that increases from zero to near 1000 represents p_1. The remainder of the curves, which remain near zero, represent p_2 through p_{30}. In the optimal solution these values are all zero, but the largest, p_9, was 34.8 when the process was terminated. Again, further improvement can be made by allowing more steps.

Appendix B

Source Code

Source code for Python scripts that carry out most of the calculations described in the thesis are provided.

B.1 helheim.py

The *Python* script *helheim.py* performed the calculations and created the figures for Chapter 2, "Evaluating the Exponent in Glen's Flow Law Using Surface Velocities on Helheim Glacier". Prior to running this script, data files were downloaded from the National Snow and Ice Data Center, see

`http://nsidc.org/data/docs/measures/nsidc0481_joughin/`

and saved in the directory structure accessed by the script. If "w" is included as a command line argument, *helheim.py* writes the files *velocity.data* and *viscosity.data* read by the scripts used for subsequent chapters.

```
# This script reads Helheim surface velocity data files downloaded from
# ftp://sidads.colorado.edu/pub/DATASETS/nsidc0481_MEASURES_greenland_V01/
# described at
# http://nsidc.org/data/docs/measures/nsidc0481_joughin/
# It then extracts velocities along a transect and calculates viscosity
# information along this transect.
# Written by Glen Granzow on May 29, 2013.

from matplotlib import pyplot
from struct import unpack
from sys import argv as command_line_arguments
import numpy, glob, os.path
```

```
discard_points    = 'd' in command_line_arguments
write_output_file = 'w' in command_line_arguments

########################################################
# Get the names of available data files and select one #
########################################################

def menu(options, default=1):
  print '<<<< OPTIONS >>>>'
  field_width = str(len(str(len(options))))
  for i in range(len(options)):
    print ('%'+field_width+'d: %s') % (i+1, options[i])
  try:
    return options[int(raw_input('Enter option (1-%d): ' % len(options)))-1]
  except:
    print 'Using default:', options[default-1], '\n'
    return options[default-1]

directory = '/home/glen/python/fenics/icestream/TSXdata/'
available_files = glob.glob(directory+'TSX_E66.50N*')
available_files = [x[-1] for x in map(os.path.split, available_files)]
original_data_set = 'TSX_E66.50N_02Sep10_13Sep10'
default = 1 + available_files.index(original_data_set)
print 'The original data set used was', original_data_set,
print '(option %d)\n' % default

filename = menu(available_files, default=default)
filename = os.path.join(directory, filename, filename)

print '\nReading files:\n', filename, '\n'

######################
# Read a geodat file #
######################

input_file = open(filename+'.vx.geodat')

line = input_file.readline()
while line:
  print line,
  if line[0] not in ('#', ';') and 'nx' not in dir():
```

```
    nx, ny = map(int, line.split())
  line = input_file.readline()
input_file.close()

n = nx*ny
bytes = 4*n

###########################
# Read the velocity files #
###########################

def readVelocityFile(extension):
  input_file = open(filename+extension, 'rb')
  raw_data = input_file.read(bytes)
  input_file.close()
  format = '>'+str(n)+'f' # Note: '>' indicates big-endian, 'f' is float
  data = numpy.array(unpack(format, raw_data))
  masked_data = numpy.ma.masked_array(data, mask=(data<-1e9))
  return masked_data.reshape(ny,nx)

vx = readVelocityFile('.vx')  # x-component
vy = readVelocityFile('.vy')  # y-component
v  = (vx*vx+vy*vy)**0.5       # magnitude

########################################
# Zoom in on the magnitude and plot it #
########################################

xmin, xmax = 200, 400
ymin, ymax = 200, 400

v = v[ymin:ymax,xmin:xmax]
ny, nx = v.shape

pyplot.ion()
pyplot.figure(1)
pyplot.clf()
pyplot.imshow(v, origin='lower')
pyplot.colorbar().set_label('|v| (m/year)')
pyplot.xticks(())
pyplot.yticks(())
pyplot.axis((0,nx,0,ny))
```

```
# Add some arrows pointing in the direction of flow

pyplot.quiver(range(0,xmax-xmin,10), range(0,ymax-ymin,10),
              vx[ymin:ymax:10,xmin:xmax:10], vy[ymin:ymax:10,xmin:xmax:10])

#####################
# Plot the transect #
#####################

x = (89, 59)
y = (105, 145)

midpoint = (sum(x)/2, sum(y)/2)
slope = -1.0/(float(y[1]-y[0])/float(x[1]-x[0]))

def line(x):
  return (x-midpoint[0])*slope + midpoint[1]

pyplot.plot((0,nx), map(line,(0,nx)), 'r', linewidth=2, dashes=(10,6))

#########################################
# Get the velocities along the transect #
#########################################

def interpolate(x, y):
  return (1-x)*y[0] + x*y[1]

t, v_t = list(), list()
ds = 0.1*(1+slope*slope)**0.5

for i in range(nx):
  y = line(i)
  j = int(y)
  if j >= ny-1:
    break
  if v.mask[j,i] or v.mask[j+1,i]:
    continue
  z = interpolate(y-j, v[j:j+2,i])
  t.append(i*ds)
  v_t.append(z)
```

```
pyplot.figure(2)
pyplot.clf()
pyplot.plot(t, v_t, '-')
pyplot.xlim(xmax=t[-1])
pyplot.xlabel('distance along the transect (km)')
pyplot.ylabel('|v| (m/year)')
pyplot.ylim(ymin=-500)

#####################################################
# Write the velocities along the transect to a file #
#####################################################

if write_output_file:
  print 'writing velocity.data'
  output_file = open('velocity.data', 'w')

  for i in range(len(t)):
    output_file.write('%f %f\n' % (t[i], v_t[i]))

  output_file.close()

#####################################################
# Find and plot viscosity ratios within the glacier #
#####################################################

peak = numpy.argmax(v_t)
d2udx2 = (v_t[peak-1] - 2*v_t[peak] + v_t[peak+1])/(ds*ds)
print 'd2udx2 =', d2udx2

for i in range(peak-5, 0, -1):
  if v_t[i-1] > v_t[i]:
    left_edge = i
    break

for i in range(peak+5,len(t)):
  if v_t[i+1] > v_t[i]:
    right_edge = i
    break

ratio = (right_edge-left_edge)*[0]
dudx  = (right_edge-left_edge)*[0]
```

```
for i in range(left_edge, right_edge):
  dudx[i-left_edge] = (v_t[i+1]-v_t[i])/ds
  ratio[i-left_edge] = d2udx2*(t[i]+ds/2-t[peak])/dudx[i-left_edge]

pyplot.figure(3)
pyplot.clf()
pyplot.plot([x+ds/2 for x in t[left_edge:right_edge]], ratio, '-')
pyplot.yscale('log')
pyplot.axhline(1, color='black', linestyle='-')
pyplot.xlabel('distance along the transect (km)')
pyplot.ylabel('$\mu / \mu_m$')

########################################
# Write the viscosity ratios to a file #
########################################

if write_output_file:
  print 'writing viscosity.data'
  output_file = open('viscosity.data', 'w')
  t_mid = [x+ds/2 for x in t[left_edge:right_edge]]

  for i in range(len(ratio)):
    output_file.write('%f %f\n' % (t_mid[i], ratio[i]))

  output_file.close()

########################################################
# Investigate the relation between viscosity and du/dx #
########################################################

pyplot.figure(4)
pyplot.clf()
pyplot.plot(numpy.abs(dudx),ratio,'-', color='gray')
pyplot.xscale('log')
pyplot.yscale('log')
pyplot.xlabel('$|dv/dx|$', fontsize=15, labelpad=0)
pyplot.ylabel('$\mu/\mu_m$', fontsize=15)

def fit_line(x, y, *args):
  pyplot.plot(x, y, *args)
  nnegative = sum([yy<=0 for yy in y])
  if nnegative > 0:  # use masked arrays
```

```
    print '\nWARNING: Point(s) removed due to nonpositive ratio (%d)' % nnegative
    numpy_ = numpy.ma
  else:
    numpy_ = numpy
  x, y = map(numpy_.log10, (x,y))
  p = numpy_.polyfit(x,y,1)
  r = numpy_.corrcoef(x,y)[0,1]
  R2 = r*r
  print 'p =', p, '  n =', 1/(p[0]+1)
  print 'R squared =', R2
  return p

m = peak-left_edge
valley = (numpy.argmin(ratio[:m-5]), (m+5)+numpy.argmin(ratio[m+5:]))
fit1 = fit_line(map(abs,dudx[:valley[0]]+dudx[valley[1]+1:]),
                ratio[:valley[0]]+ratio[valley[1]+1:], 'sr')
if discard_points:
  fit2 = fit_line(map(abs,dudx[valley[0]+1:m-4]+dudx[m+4:valley[1]]),
                  ratio[valley[0]+1:m-4]+ratio[m+4:valley[1]], '^b')
else:
  fit2 = fit_line(map(abs,dudx[valley[0]+1:valley[1]]),
                  ratio[valley[0]+1:valley[1]], '^b')

# Find the point where the two fit lines cross

x = (fit1[1]-fit2[1])/(fit2[0]-fit1[0])
y = numpy.polyval(fit1, x)

# Draw lines through this point with slopes corresponding to Glen's flow law exponents

pyplot.figure(4)
xmin, xmax, ymin, ymax = pyplot.axis()

def line(xx):
  return 10**(y + slope*(numpy.log10(xx)-x))

for n in (1,2,3,'$\infty$'):
  if n == '$\infty$':
    slope = -1
  else:
    slope = -(n-1.0)/n
  pyplot.plot((xmin, xmax), map(line, (xmin, xmax)), '--', color='gray')
```

```
  pyplot.text(1.8*xmin, 1.1*line(1.8*xmin), 'n = '+str(n))

pyplot.axis((xmin, xmax, 7*ymin, ymax))

##########################################################
# Add vertical lines to the velocity and viscosity plots #
##########################################################

for figure in (2,3):
  pyplot.figure(figure)
  pyplot.axvline(t[peak], color='black')
  pyplot.axvline(t[left_edge],  color='black', linestyle='--')
  pyplot.axvline(t[right_edge], color='black', linestyle='--')
  pyplot.axvline(t[left_edge+valley[0]]+ds/2, color='gray', linewidth=1.5, dashes=(2,3))
  pyplot.axvline(t[left_edge+valley[1]]+ds/2, color='gray', linewidth=1.5, dashes=(2,3))

try:
  __IPYTHON__
except:
  raw_input('Press ENTER to end the program')
```

B.2 forward1D.py

The *Python* script *forward1D.py* created the figure in Chapter 3, "A Forward Problem: Solving for Velocity Given Viscosity". This script reads the files *velocity.data* and *viscosity.data* written by *helheim.py*.

```
# This script solves the boundary value problem problem:
# d/dx(mu du/dx) = C with u(0) = u(L) = 0.0
# where mu is the viscosity found by helheim.py.
# Written by Glen Granzow on July 29, 2013.

from dolfin import *
from matplotlib import pyplot
from sys import argv as command_line_arguments

######################
# Read File Function #
######################

def read_file(filename):
  input_file = open(filename, 'r')
```

```
  line = input_file.readline()
  data = list()
  while line:
    data.append(map(float, line.split()))
    line = input_file.readline()
  input_file.close()
  return zip(*data)

############################################################
# An Expression type object that does linear interpolation #
############################################################

class Interpolate1D(Expression):

  def __init__(self, x, y, verbose=0):
#   x = a list of independent variable values
#   y = a list of the corresponding dependent variable values
    self.data = [x, y]
    self.verbose = verbose

  def eval(self, values, x):
    if x[0] < self.data[0][0]:
      values[0] = self.data[1][0]
      if self.verbose:
        print 'WARNING: Interpolate1D is extrapolating!', x[0], '<', self.data[0][0]
      if self.verbose > 1: raw_input('Press ENTER to continue')
    elif x[0] > self.data[0][-1]:
      values[0] = self.data[1][-1]
      if self.verbose:
        print 'WARNING: Interpolate1D is extrapolating!', x[0], '>', self.data[0][-1]
      if self.verbose > 1: raw_input('Press ENTER to continue')
    else:
      for i in range(1,len(self.data[0])):
        if x[0] <= self.data[0][i]:
          alpha = (self.data[0][i]-x[0])/(self.data[0][i]-self.data[0][i-1])
          values[0] = alpha*self.data[1][i-1] + (1-alpha)*self.data[1][i]
          return

################
# Main Program #
################
```

```
if __name__ == '__main__':

# Read the measured velocities from file

  x_velocity, velocity = read_file('velocity.data')
  x_velocity, velocity = x_velocity[50:101], velocity[50:101]
  domain = (x_velocity[0], x_velocity[-1])
  n_elements = 2*(len(velocity)-1)

# Plot the measured velocities

  pyplot.ion()
  pyplot.figure(1)
  pyplot.clf()
  pyplot.plot(x_velocity, velocity, '-r', linewidth=3)
  pyplot.ylim(ymin=-500)
  pyplot.xlabel('$x\ \ (km)$', fontsize=15)
  pyplot.ylabel('$v\ \ (m/year)$', fontsize=15)
  pyplot.xlim(domain)

# Set up and solve the Finite Element model

  mesh = IntervalMesh(n_elements, *domain)
  V    = FunctionSpace(mesh, 'Lagrange', 1)
  u    = TrialFunction(V)
  v    = TestFunction(V)

  x_viscosity, viscosity = read_file('viscosity.data')
  mu = Interpolate1D(x_viscosity, viscosity, True)

  if 'x' in command_line_arguments:
#   An equivalent way to incorporate the viscosity data
    print 'Using an alternate way to incorporate the viscosity data'
    from numpy import array
    midpoint_interpolants = (array(viscosity[:-1])+array(viscosity[1:]))/2
    mu = Function(V)
    mu.vector()[0]      = viscosity[0]
    mu.vector()[1:-1:2] = array(viscosity)
    mu.vector()[2:-1:2] = midpoint_interpolants
    mu.vector()[-1]     = viscosity[-1]

  solution = Function(V)
```

```
  boundary_conditions = DirichletBC(V, Constant(0.0), 'on_boundary')

  if 'b' in command_line_arguments:
#   Use the measured velocities at the endpoints as boundary conditions
    boundary_conditions = [DirichletBC(V, Constant(velocity[0]),
                           'on_boundary && near(x[0],%f)' % domain[0]),
                           DirichletBC(V, Constant(velocity[-1]),
                           'on_boundary && near(x[0],%f)' % domain[-1])]

# The constant 443 was chosen to match the maximum measured velocity

  L = Constant(443.247856338) * v * dx
  a = mu * u.dx(0) * v.dx(0) * dx

  solve(a == L, solution, boundary_conditions)

# Plot the calculated velocities

  pyplot.plot(mesh.coordinates(), solution.vector().array(), '--k', linewidth=3)
  print 'maximum measured velocity/maximum calculated velocity =',
  print  max(velocity)/max(solution.vector())

# Plot mu

  if 'p' in command_line_arguments:
    pyplot.figure(2)
    pyplot.clf()
    pyplot.plot(x_viscosity, viscosity, '-r')
    pyplot.plot(mesh.coordinates(), project(mu,V).vector().array(), '-b')
    pyplot.yscale('log')
    pyplot.xlim(domain)

  try:
    __IPYTHON__
  except:
    raw_input('Press ENTER to exit the program')
```

B.3 inverse1D.py

The *Python* script *inverse1D.py* created all but the last figure in Chapter 4, “An Inverse Problem: Solving for Viscosity Given Velocity”. This script reads the files

velocity.data and *viscosity.data* written by *helheim.py*.

```
# This script solves an ODE constrained optimization problem:
# Find eta such that the solution to
# d/dx(eta^2 du/dx) = -c with u(0) = u(L) = 0.0 and constant c
# minimizes J = integral from 0 to L of (u - g(x))^2 (+ optional regularization)
# where g(x) is the surface velocity on Helheim Glacier.
# Written by Glen Granzow on July 30, 2013.

from dolfin import *
from dolfin_adjoint import *
from matplotlib import pyplot
from scipy.optimize import fmin_bfgs
from sys import argv, stdout
import numpy

set_log_active(True)

##############################################
# Read the measured velocity data from file. #
##############################################

from forward1D import read_file, Interpolate1D

x_velocity, velocity = read_file('velocity.data')
x_velocity, velocity = x_velocity[50:101], velocity[50:101]
v_max = max(velocity)
velocity = [v/v_max for v in velocity] # normalize the velocity

g = Interpolate1D(x_velocity, velocity, True)  # the goal

##########################################
# Define the "forward" problem and solve #
# with eta equal to a constant function. #
##########################################

domain = (x_velocity[0], x_velocity[-1])
n_elements = 2*(len(velocity)-1)
c = 1.0

adj_reset()  # reset dolfin_adjoint

mesh = IntervalMesh(n_elements, *domain)
```

```
V    = FunctionSpace(mesh, 'Lagrange', 1)
u    = TrialFunction(V)
v    = TestFunction(V)
eta  = Function(V)     # eta is the square root of the viscosity,
eta.vector()[:] = 3.0  # initialized to a constant

solution = Function(V)
boundary_conditions = DirichletBC(V, Constant(0.0), 'on_boundary')

if 'b' in argv:
  print '\nUsing measured velocities as boundary conditions\n'
  boundary_conditions = [DirichletBC(V, Constant(velocity[0]),
                        'on_boundary && near(x[0],%f)' % domain[0]),
                        DirichletBC(V, Constant(velocity[-1]),
                        'on_boundary && near(x[0],%f)' % domain[-1])]

L = Constant(c) * v * dx
a = eta * eta * u.dx(0) * v.dx(0) * dx

solve(a == L, solution, boundary_conditions)

################################################################
# Plot the "goal" which is the measured velocities on Helheim. #
################################################################

pyplot.ion()
pyplot.figure(1)
pyplot.clf()
if 'f' in argv: # Puts each plot in a separate figure
  pyplot.xlabel('$x \ \ (km)$', fontsize=15)
  pyplot.ylabel('$v$', fontsize=15)
  linewidth = 3
else:
  pyplot.subplot(221)
  linewidth = 1.5
pyplot.plot(mesh.coordinates(), project(g,V).vector().array(),
            'r', linewidth=linewidth)
pyplot.ylim(-0.1,1.1)
pyplot.xlim(domain)

######################################################
# Define the functional, J, that is to be minimized. #
```

```
######################################################

if ('t' in argv):  # Tikhonov Regularization
  try:
    alpha = float(argv[-1])
  except:
    alpha = 2.0e-6
  print '\nThe weight of the Tikhonov regularization term is',
  print 'alpha = %g\n' % alpha
  J = (solution-g)**2*dx + Constant(alpha)*eta.dx(0)**2*dx
else:
  J = (solution-g)**2*dx

#####################################
# Solve the "inverse problem", i.e. #
# find the optimum function eta.    #
#####################################

set_log_active(False)

def error(coefficients):
  stdout.write('.')
  stdout.flush()
  adj_reset()
  eta.vector()[:] = coefficients
  solve(a == L, solution, boundary_conditions)
  return assemble(J)

def derivatives(coefficients):
  d = compute_gradient(Functional(J*dt[FINISH_TIME]),
                                  InitialConditionParameter(eta))
  return d.vector().array()

coefficients = eta.vector().array()
optimum = fmin_bfgs(error, coefficients, derivatives, gtol=1e-6)

#####################
# Plot the results. #
#####################

# Plot the velocity, u (found using the optimal eta) versus x.
```

```
pyplot.plot(mesh.coordinates(), solution.vector().array(), '--k',
            linewidth=linewidth)

# Plot the viscosity ratio versus x.

if 'f' in argv:
  pyplot.figure(3)
  pyplot.clf()
  pyplot.xlabel('$x \ \ (km)$', fontsize=15)
  pyplot.ylabel('$\mu / \mu_m$', fontsize=15)
else:
  pyplot.subplot(223)

x_viscosity, viscosity = read_file('viscosity.data')
pyplot.plot(x_viscosity, viscosity)
pyplot.yscale('log')
pyplot.axhline(1, color='black')

m = numpy.argmax(solution.vector().array())
eta_m = optimum[m]
print 'm =', m, 'where x =', mesh.coordinates()[m],
print 'v =', solution.vector()[m], 'eta =', eta_m
mu_m = eta_m*eta_m  # viscosity at x_m (eta is the square root of viscosity)
print 'average eta is', sum(optimum)/len(optimum)
viscosity_ratio = [x*x/mu_m for x in optimum]
print 'viscosity was normalized by dividing by mu_m =', mu_m

pyplot.plot(mesh.coordinates(), viscosity_ratio)
pyplot.xlim(domain)

# Find the "shear margins" (minima of viscosity)

minima = (numpy.argmin(viscosity_ratio[:m]),
        m+numpy.argmin(viscosity_ratio[m:]))

if 't' not in argv or alpha < 0.1: # plot vertical lines at the shear margins
  for plot in (1,3):
    if 'f' in argv:
      pyplot.figure(plot)
    else:
      pyplot.subplot(220+plot)
    for minimum in minima:
```

```
      pyplot.axvline(mesh.coordinates()[minimum], color='black',
                     linewidth=1.5, dashes=(2,3))

# Plot the viscosity ratio versus |du/dx|

if 'f' in argv:
  pyplot.figure(2)
  pyplot.clf()
  pyplot.xlabel('$|dv/dx|$', fontsize=15, labelpad=0)
  pyplot.ylabel('$\mu / \mu_m$', fontsize=15)
else:
  pyplot.subplot(222)
dudx = project(solution.dx(0),V)
abs_dudx = map(abs,dudx.vector())
pyplot.loglog(abs_dudx, viscosity_ratio, color='gray')
pyplot.plot(abs_dudx[:minima[0]], viscosity_ratio[:minima[0]], 'sr',
        abs_dudx[minima[1]+1:], viscosity_ratio[minima[1]+1:], 'sr')
pyplot.plot(abs_dudx[minima[0]+1:minima[1]],
            viscosity_ratio[minima[0]+1:minima[1]], '^b')
margin = 1.5
xmin, xmax = min(abs_dudx)/margin, margin*max(abs_dudx)
ymin, ymax = min(viscosity_ratio)/margin, margin*max(viscosity_ratio)
pyplot.axis((xmin, xmax, ymin, ymax))

# Draw lines with slopes corresponding to Glen's flow law exponents

p = numpy.polyfit(numpy.log10(abs_dudx[:minima[0]]+abs_dudx[minima[1]+1:]),
    numpy.log10(viscosity_ratio[:minima[0]]+viscosity_ratio[minima[1]+1:]),1)
print 'p =', p

x_intercept = -p[1]/p[0]

def line(x):
  return 10**(slope*(numpy.log10(x)-x_intercept))

for n,x,y in zip((1,2,3,'$\infty$'),(0.0023,0.0023,0.009,0.028),(1.1,6.5,10,10)):
  if n == '$\infty$':
    slope = -1
  else:
    slope = -(n-1.0)/n
  pyplot.plot((xmin, xmax), map(line, (xmin, xmax)), '--', color='gray')
  pyplot.text(x, y, 'n = '+str(n))
```

```
try:
  __IPYTHON__
except:
  raw_input('Press ENTER to exit the program')
```

B.4 curvefit.py

The *Python* script *curvefit.py* performed the calculations and created the figures in Chapter 5, "Using Least-Squares Fits to Evaluate the Exponent in Glen's Flow Law". This script reads the file *velocity.data* written by *helheim.py*.

```
# Fit the curve associated with Glen's flow law
# to the velocity data output by helheim.py.
# Written by Glen Granzow on August 9, 2013.

# Valid command line arguments:
# 'a' - Use absolute value in v(x).
# 'b' - Use scipy's fmin_bfgs instead of fsolve.
# 'f' - Create a figure for the thesis.
# 'l' - Loop over values of n.
# 'z' - Zoom in on the plot of v versus x.
# If the last three command line arguments are numbers, they are used as
# an initial guess of A, B, and C, which are the parameters in v(x).

from matplotlib import pyplot
from scipy import optimize
import numpy, sys

##########################
### Read File Function ###
##########################

def read_file(filename):
  input_file = open(filename, 'r')
  line = input_file.readline()
  data = list()
  while line:
    data.append(map(float, line.split()))
    line = input_file.readline()
  input_file.close()
  return zip(*data)
```

```
####################
### MAIN PROGRAM ###
####################

if __name__ == '__main__':

# Read the measured velocities from file.

  x_velocity, velocity = read_file('velocity.data')

# Get user input.

  if 'l' in sys.argv or 'f' in sys.argv: # special case: loop over values of n
    n_points = 21
    n = 0.5
    sys.argv.append('a')
  else:
    try:
      n_points = int(raw_input('Enter the number of points to fit: '))
    except:
      n_points = 21

    try:
      n = float(raw_input("Enter the exponent in Glen's flow law: "))
    except:
      n = 1.0

# Fit the curve.

  i_m = numpy.argmax(velocity)
  i_first = i_m - n_points/2
  i_last  = i_first + n_points
  print 'Fitting', len(velocity[i_first:i_last]),
  print "points for Glen's flow law with n =", n

  def v(A,B,C,x):
    if 'a' in sys.argv:
      return B - C*abs(A-x)**(n+1)
    else:
      return B - C*(A-x)**(n+1)
```

```
def Error(ABC): # Sum of the squares of the residuals
  global fit_values
  A, B, C = ABC
  fit_values = map(lambda x: v(A,B,C,x), x_velocity[i_first:i_last])
  return sum([(v_fit - v_data)**2 for v_fit, v_data
               in zip(fit_values,velocity[i_first:i_last])])

def dEdv(A,B,C,j):
  return 2 * (v(A,B,C,x_velocity[j]) - velocity[j])

def dvdA(A,B,C,j):
  if 'a' in sys.argv:
    return (-1)**(x_velocity[j]<=A) * C*(n+1)*abs(A-x_velocity[j])**n
  else:
    return - C*(n+1)*(A-x_velocity[j])**n

def dvdB(A,B,C,j):
  return 1

def dvdC(A,B,C,j):
  if 'a' in sys.argv:
    return -abs(A-x_velocity[j])**(n+1)
  else:
    return -(A-x_velocity[j])**(n+1)

def dEdA(A,B,C): # f1(A,B,C)
  return sum([dEdv(A,B,C,k)*dvdA(A,B,C,k) for k in range(i_first,i_last)])

def dEdB(A,B,C): # f2(A,B,C)
  return sum([dEdv(A,B,C,k)*dvdB(A,B,C,k) for k in range(i_first,i_last)])

def dEdC(A,B,C): # f3(A,B,C)
  return sum([dEdv(A,B,C,k)*dvdC(A,B,C,k) for k in range(i_first,i_last)])

try:
  initial_guess = numpy.array(map(float, sys.argv[-3:]))
except:
  A = x_velocity[i_m]
  B = velocity[i_m]
  deltaX = 0.5*(x_velocity[i_last-1]-x_velocity[i_first])
  deltaV = B - 0.5*(velocity[i_first] + velocity[i_last-1])
  C = deltaV/deltaX**(n+1)
```

```
    initial_guess = numpy.array((A,B,C))

# Solve the system of equations:
# f1(A,B,C) = 0, f2(A,B,C) = 0, and f3(A,B,C) = 0.

  def system_of_equations(ABC):
    A,B,C = ABC
    return numpy.array((dEdA(A,B,C), dEdB(A,B,C), dEdC(A,B,C)))

  print 'initial_guess =', initial_guess
  if 'b' in sys.argv:
    A,B,C = optimize.fmin_bfgs(Error, initial_guess, system_of_equations)
  else:
    A,B,C = optimize.fsolve(system_of_equations, initial_guess)

  print 'A =', A, '\nB =', B, '\nC =', C

# Calculate the sum of the squares of the residuals.

  SS = Error((A,B,C))
  E = (SS/n_points)**0.5
  print 'SS_res = %.1f \nE = %.1f' % (SS, E)

# Plot the results.

  pyplot.ion()
  pyplot.figure(1)
  pyplot.clf()
  i, j = i_m-28, i_m+23
  pyplot.plot(x_velocity[i:j], velocity[i:j], '-r', linewidth=3)
  pyplot.plot(x_velocity[i_first:i_last], fit_values, '--k', linewidth=3)
  pyplot.xlabel('$x \ \ (km)$', fontsize=15)
  pyplot.ylabel('$v \ \ (m/year)$', fontsize=15)
  if 'z' in sys.argv:
    pyplot.subplots_adjust(bottom=0.25)
    pyplot.axis((x_velocity[i_first-1], x_velocity[i_last], 4200, 5200))
    pyplot.text(8.57, 4330, '$n = %g$'%n, fontsize=20)
    pyplot.text(8.55, 4235, '$\overline{E} = %.1f$'%E, fontsize=20)
  else:
    pyplot.subplots_adjust(bottom=0.1)
    pyplot.axis((x_velocity[i_m-28], x_velocity[i_m+22], -500, 5500))
    pyplot.axvline(x_velocity[i_m], color='black')
```

```
  pyplot.grid(True)

# Loop over n to find minimum E

  if 'l' in sys.argv:
    n_list, E_list = list(), list()
    for n in numpy.linspace(0.5,3.5,301):
      initial_guess = numpy.array((A,B,C))
      A, B, C = optimize.fsolve(system_of_equations, initial_guess)
      SS = Error((A,B,C))
      E = (SS/n_points)**0.5
      print 'n = %.2f, A = %.1f, B = %.1f, C = %.1f, E = %.1f' % (n,A,B,C,E)
      n_list.append(n)
      E_list.append(E)
    index = numpy.argmin(E_list)
    print '\nMinimum E = %f at n = %f' % (E_list[index], n_list[index])

    pyplot.clf()
    pyplot.plot(n_list, E_list, linewidth=1.75)
    pyplot.ylim(ymax=40)
    pyplot.xlabel('$n$', fontsize=20)
    pyplot.ylabel('$\overline{E}$  (m/year)', fontsize=20)
    pyplot.grid(True)

# Loop to create a figure for the thesis

  if 'f' in sys.argv:
    pyplot.figure(2, figsize=(8.125, 1.75*6.125))
    pyplot.clf()
    pyplot.subplots_adjust(bottom=0.05, top=0.95)
    subplot = 411
    for n in (0.5,1,1.6,3):
      initial_guess = numpy.array((A,B,C))
      A, B, C = optimize.fsolve(system_of_equations, initial_guess)
      SS = Error((A,B,C))
      E = (SS/n_points)**0.5
      print 'n = %.2f, A = %.1f, B = %.1f, C = %.1f, E = %.1f' %(n,A,B,C,E)
      pyplot.subplot(subplot)
      pyplot.plot(x_velocity, velocity, '-r', linewidth=3)
      pyplot.plot(x_velocity[i_first:i_last], fit_values, '--k', linewidth=3)
      pyplot.axis((x_velocity[i_first-1], x_velocity[i_last], 4400, 5200))
      pyplot.text(8.43, 5080, '('+'abcd'[subplot-411]+')', fontsize=20)
```

```
    pyplot.text(8.60, 4570, '$n = %g$'%n, fontsize=20)
    pyplot.text(8.58, 4435, '$\overline{E} = %.1f$'%E, fontsize=20)
    pyplot.ylabel('v (m/year)')
    pyplot.grid(True)
    subplot += 1
  pyplot.xlabel('x (km)')

try:
  __IPYTHON__
except:
  raw_input('Press ENTER to end the program')
```

B.5 adjoint.py

The *Python* script *adjoint.py* performed the calculations and created the figure in Section A.3, “A Numerical Example”, of Appendix A, “Adjoint Methods”.

```
# This script approximates the solution to the boundary problem
# c_2 u" + c_1 u' + c_0 u = f(x), u(0) = a_0, u(1) = a_1
# where f(x) = p_0 + p_1 x + ... + p_n x^n (a polynomial)
# using finite differences.
# The adjoint method is then used to find dg/dc, dg/dp, and dg/da
# for g = u(0.5) and g = average(u).
# Written by Glen Granzow on October 15, 2012.

from copy import deepcopy
from matplotlib import pyplot
from math import sqrt, sin, cos, exp

#############################
### POLYNOMIAL OPERATIONS ###
#############################

def polynomial(p, x): # a recursive method for polynomial evaluation
  if len(p) == 1:
    return p[0]
  return p[0] + x * polynomial(p[1:],x)

def f(x):
  return polynomial(p, x)

# The rest of the polynomial operations were written to test the
```

```
# particular solution part of the exact solution.

def derivative(p):
  return [k*p[k] for k in range(len(p))][1:]

def add(a, b):
  if len(a) > len(b):
    return_value = deepcopy(a)
    for k in range(len(b)):
      return_value[k] += b[k]
  else:
    return_value = deepcopy(b)
    for k in range(len(a)):
      return_value[k] += a[k]
  return return_value

def times(a, p):
  return [a * coefficient for coefficient in p]

def check_particular_solution():
  u = exact_solution.coefficients
  u_prime = derivative(u)
  u_double_prime = derivative(u_prime)
  print '\nChecking the particular solution...'
  print add(times(c[2], u_double_prime),
        add(times(c[1], u_prime), times(c[0], u)))
  print 'should match p'
  print p

######################
### EXACT SOLUTION ###
######################

class Exact_solution():

# The form of the homogeneous solution depends on the discriminant
# of the characteristic equation, r^2 + c1 r + c0 = 0.

  def homogeneous_solution_0(self, x):
    return exp(self.alpha*x)*(self.A*cos(self.omega*x)
                            + self.B*sin(self.omega*x))
```

```
  def homogeneous_solution_1(self, x):
    return (self.A+self.B*x)*exp(self.r*x)

  def homogeneous_solution_2(self, x):
    return self.A*exp(self.r1*x) + self.B*exp(self.r2*x)

# The particular solution is a polynomial.
# If c0 = 0 and c1 = 0 it has degree two greater than the degree of f.
# If c0 = 0 but c1 is not zero it has degree one greater than f.
# Otherwise its degree is the same as the degree of f.

  def particular_solution(self, x):
    return polynomial(self.coefficients, x)

# Initialization

  def __init__(self, c, p):
    if c[2] == 0:
      raise ValueError('c[2] must be nonzero.')

#   Find the coefficients in the particular solution.

    if c[1] == 0 and c[0] == 0:
      self.coefficients = [0,0] + [p[k]/(c[2]*(k+1)*(k+2)) for k in range(len(p))]
    elif c[0] == 0:
      self.coefficients = [p[-1]/len(p)/c[1]]
      for k in range(len(p)-2, -1, -1):
        self.coefficients.append(
            (p[k]/(k+1) - c[2]*(k+2)*self.coefficients[-1])/c[1])
      self.coefficients.append(0.0)
      self.coefficients.reverse()
    else:
      self.coefficients = [p[-1]/c[0]]
      if len(p) > 1:
        k = len(p)-2
        self.coefficients.append(
            (p[k] - c[1]*(k+1)*self.coefficients[-1])/c[0])
        for k in range(len(p)-3, -1, -1):
          self.coefficients.append(
            (p[k] - c[1]*(k+1)*self.coefficients[-1] -
              c[2]*(k+1)*(k+2)*self.coefficients[-2])/c[0])
      self.coefficients.reverse()
```

```
#   Select the appropriate form for the homogeneous solution and set
#   the constants A and B to satisfy the boundary conditions.

    self.discriminant = c[1]*c[1] - 4*c[2]*c[0]

    if self.discriminant < 0:
      print 'u(x) = exp(alpha*x) * (A*cos(omega*x) + B*sin(omega*x))'
      self.alpha = -c[1]/(2.0*c[2])
      self.omega = sqrt(-self.discriminant)/(2.0*c[2])
      self.A =  a[0]-self.particular_solution(0.0)
      self.B =((a[1]-self.particular_solution(1.0))/exp(self.alpha)
                 - self.A*cos(self.omega)) / sin(self.omega)
      self.homogeneous_solution = self.homogeneous_solution_0

    elif self.discriminant == 0:
      print 'u(x) = (A + B*x) * exp(r*x)'
      self.r = -c[1]/(2.0*c[2])
      self.A =  a[0] - self.particular_solution(0.0)
      self.B = (a[1] - self.particular_solution(1.0))/exp(self.r) - self.A
      self.homogeneous_solution = self.homogeneous_solution_1

    else: # self.discriminant > 0
      print 'u(x) = A*exp(r1*x) + B*exp(r2*x)'
      self.r1 = (-c[1] + sqrt(self.discriminant))/(2.0*c[2])
      self.r2 = (-c[1] - sqrt(self.discriminant))/(2.0*c[2])
      matrix = [[None, 1.0, 1.0], [exp(self.r1), exp(self.r2), 0.0]]
      rhs    = [a[0] - self.particular_solution(0.0),
                a[1] - self.particular_solution(1.0)]
      self.A, self.B = solve_tridiagonal(matrix, rhs)
      self.homogeneous_solution = self.homogeneous_solution_2

# Evaluation

  def __call__(self, x):
    return self.particular_solution(x) + self.homogeneous_solution(x)

#########################
### MATRIX OPERATIONS ###
#########################

def finite_difference_matrices(c, p, a):
```

```
  A = [[None, 1, 0]] # first row of tri-diagonal matrix A in Ax = b
  b = [a[0]]         # first row of right hand side vector b in Ax = b

  for row in range(1,n-1):
    x = float(row)/(n-1)
    A.append([c[2]/dx_squared-c[1]/(2*dx), c[0]-2.0*c[2]/dx_squared,
              c[2]/dx_squared+c[1]/(2*dx)])
    b.append(polynomial(p, x))

  A.append([0, 1, 0]) # the last 0 is not really in the matrix
  b.append(a[1])
  return A, b

def solve_tridiagonal(A, b):
# Gaussian elimination
  A[0][2] /= A[0][1]
  for row in range(1, len(A)):
    A[row][1] -= A[row][0] * A[row-1][2]
    b[row]    -= A[row][0] * b[row-1]
    A[row][2] /= A[row][1]
    b[row]    /= A[row][1]
# Back substitution
  for row in range(-2, -(len(A)+1), -1):
    b[row] -= A[row][2] * b[row+1]
  return b

def transpose(A): # only for a tridiagonal matrix
  for row in range(1, len(A)):
    A[row-1][2], A[row][0] = A[row][0], A[row-1][2]
  return A

def multiply(a, B):
# a is a row matrix, B is a rectangular matrix
  return_value = len(B[0]) * [0]
  for column_of_B in range(len(B[0])):
    for row_of_B in range(len(B)):
      return_value[column_of_B] += a[row_of_B] * B[row_of_B][column_of_B]
  return return_value

def dot(a, b):
  return sum([aa*bb for aa, bb in zip(a,b)])
```

```
####################
### MAIN PROGRAM ###
####################

if __name__ == '__main__':

# Parameters:

  c = (1.0, -2.0, 1.0)  # coefficients in the differential equation
  p = (1.0, 1.0, -5.0)  # coefficients of the polynomial f
  a = (0.0, 0.0)        # boundary condition values

  print '\nc = %s\np = %s\na = %s\n' % (c, p, a)

# Print some information about the exact solution.

  exact_solution = Exact_solution(c, p)

  for item in dir(exact_solution):
    if isinstance(eval('exact_solution.'+item), float):
      print item, '=', eval('exact_solution.'+item)
  print 'coefficients =', exact_solution.coefficients

  check_particular_solution()

# Calculate an approximate solution.

  n = 21             # number of nodes
  dx = 1.0/(n-1)     # node spacing
  dx_squared = dx*dx
  A, b = finite_difference_matrices(c, p, a)
  copy_of_A = deepcopy(A)  # save a copy for the adjoint method
  approximate_solution = solve_tridiagonal(A, b)

# Plot the exact and approximate solutions

  pyplot.ion()
  pyplot.clf()
  xx = [i/100.0 for i in range(101)]
  pyplot.plot(xx, map(exact_solution, xx), 'k-')

  xx = [i/float(n-1) for i in range(n)]
```

```
  pyplot.plot(xx, approximate_solution, 'ok')

  pyplot.xticks([k/10.0 for k in range(11)])
  pyplot.grid(True)
#  pyplot.xlabel('x')
#  pyplot.ylabel('u')

# Use the adjoint method to find partial derivatives.

  def print_partial_derivatives(dgdcpa):
    print 'dgdc =', dgdcpa[:len(c)]
    print 'dgdp =', dgdcpa[len(c):-2]
    print 'dgda =', dgdcpa[-2:]

  dfdcpa = [(len(c)+len(p))*[0.0]+[-1,0]]
  u = approximate_solution
  for k in range(1,n-1):
    x = float(k)/(n-1)
    dfdcpa.append([u[k], (u[k+1]-u[k-1])/(2*dx),
                  (u[k-1]-2*u[k]+u[k+1])/dx_squared] +
                  [-x**k for k in range(len(p))] + [0,0])
  dfdcpa.append((len(c)+len(p))*[0.0]+[0,-1])

# First, use g = u(0.5)

  dgdu = n * [0.0]
  index = n/2
  dgdu[index] = 1.0

  A = deepcopy(copy_of_A)
  lamda = solve_tridiagonal(transpose(A), dgdu)
  dgdcpa = [-z for z in multiply(lamda, dfdcpa)]
  x = float(index)/(n-1)
  print '\nFor g = u(%f):' % x
  print_partial_derivatives(dgdcpa)

  if c[0] == 0.0 and c[1] == 0.0:
    exact_dgdp = [c[2]*(x**k-x)/k/(k-1) for k in range(2,2+len(p))]
    print 'exact dgdp =', exact_dgdp

# Next, use g = average(u)
```

```
  for quadrature in ("trapezoidal rule", "Simpson's rule"):
    if quadrature == "trapezoidal rule":
      dgdu = map(lambda z: z/(n-1), [0.5] + (n-2)*[1.0] + [0.5])
    else:
      dgdu = map(lambda z: z/(3.0*(n-1)), [1] + (n-2)/2*[4,2] + [4,1])

    A = deepcopy(copy_of_A)
    lamda = solve_tridiagonal(transpose(A), dgdu)
    dgdcpa = [-z for z in multiply(lamda, dfdcpa)]
    print '\nFor g = average(u) using', quadrature
    print_partial_derivatives(dgdcpa)

  if c[0] == 0.0 and c[1] == 0.0:
    exact_dgdp = [-c[2]*(0.5-1.0/(k+3))/(k+2)/(k+1) for k in range(len(p))]
    print 'exact dgdp =', exact_dgdp

# Check the partial derivatives numerically by perturbing the
# parameters one at a time.

  perturbation = 0.01
  numerical_dgdcpa = [[],[],[]]
  trapezoidal_rule = map(lambda z: z/(n-1), [0.5] + (n-2)*[1.0] + [0.5])
  simpsons_rule = map(lambda z: z/(3.0*(n-1)), [1] + (n-2)/2*[4,2] + [4,1])
  integral = (dot(trapezoidal_rule, approximate_solution),
              dot(simpsons_rule,    approximate_solution))
  plot = False

  for k in range(len(c)+len(p)+len(a)):
    parameters = list(c + p + a)
    parameters[k] += perturbation
    A, b = finite_difference_matrices(
      parameters[:len(p)], parameters[len(p):-2], parameters[-2:])
    new_u = solve_tridiagonal(A, b)
    if plot:
      pyplot.plot(xx, new_u, '--')
    numerical_dgdcpa[0].append((new_u[index]-u[index])/perturbation)
    numerical_dgdcpa[1].append((dot(trapezoidal_rule, new_u)-integral[0])/perturbation)
    numerical_dgdcpa[2].append((dot(simpsons_rule, new_u)-integral[1])/perturbation)

  print '\nNumerical estimates:'
  for k in range(3):
    print
```

```
    print_partial_derivatives(numerical_dgdcpa[k])

  try:
    __IPYTHON__
  except:
    raw_input('Press ENTER to exit the program')
```

B.6 ismiphomD.py

The *Python* script *ismiphomD.py* creates a figure like that in Section A.4, “A Data Assimilation Example From Glaciology”, of Appendix A, “Adjoint Methods”. Note: This script has been modified (to enable it to run using the most recent versions of the *dolfin* and *dolfin-adjoint* packages) since the figure in Section A.4 was actually created; the output from the script given here differs marginally from that in the text.

```
# This script finds the basal traction from the surface velocity
# in ISMIPHOM Experiment D.
# This version writes beta_squared in the form:
# beta_squared = C_0 + sum [C_(2i-1) sin(omega_i x) + C_(2i) cos(omega_i x)]
# where omega_i = i * 2 * pi /length, for i = 1, 2, ..., n.
# and finds the C_k's using Scipy's implementation of the BFGS method
# (which uses derivatives provided by dolfin-adjoint.)
# Written by Glen Granzow on November 7, 2012.
# Modified on August 14, 2013 to accommodate changes in FEniCS.

from dolfin import *
from dolfin_adjoint import *
from numpy import array
from scipy.optimize import fmin_bfgs
from matplotlib import pyplot
from sys import argv as command_line_arguments
from sys import stdout

verbose = True
set_log_level(INFO)
set_log_active(True)

################################
# Plotting and input functions #
################################
```

```
def plotUvsX(y_coordinate):
  x = mesh.coordinates()[:nx+1,0]
  y = [solution(x_coordinate, y_coordinate)[0] for x_coordinate in x]
  pyplot.plot(x, y, color='gray')
  y = [goalFunction(x_coordinate, y_coordinate)[0] for x_coordinate in x]
  pyplot.plot(x, y, linewidth=2, color='black')
  pyplot.ylabel('u')

def plotError(error, iterate=[-1], previous=[None]):
  if iterate[0] > 0:
    pyplot.plot([iterate[0]-1]+iterate, previous+[error], color='black')
    pyplot.xlim(0, max(30, len(iterates)-1))
    pyplot.yscale('log')
    pyplot.grid(True)
  iterate[0] += 1
  previous[0] = error

def plotCoefficients(coefficients, iterate=[-1], previous=[]):
  colors = 'bgrcmy'
  if iterate[0] > 0:
    for i in range(len(coefficients)):
      pyplot.plot([iterate[0]-1]+iterate, [previous[i]]+[coefficients[i]],
                  colors[i%len(colors)])
      previous[i] = coefficients[i]
    pyplot.xlim(0, max(30, len(iterates)-1))
    pyplot.grid(True)
  else:
    for c in coefficients:
      previous.append(c)
  iterate[0] += 1
  pyplot.xlabel('iterate')

def plotBetaSquared(c):
  x_coordinates = mesh.coordinates()[:nx+1,0]
  f = beta_squared.replace('Expression("', '').replace('[0])"', '')
  y = [eval(f) for x in x_coordinates]
  pyplot.plot(x_coordinates, y, color='gray')
  c = [1000, 1000] + (len(c)-2)*[0]
  y = [eval(f) for x in x_coordinates]
  pyplot.plot(x_coordinates, y, linewidth=2, color='black')
  pyplot.xlabel('x')
  pyplot.ylabel('$\\beta\,^2$')
```

```
def plot4(coefficients, error):
  pyplot.subplot(221)
  plotUvsX(height)
  pyplot.subplot(222)
  plotError(error)
  pyplot.subplot(223)
  plotBetaSquared(coefficients)
  pyplot.subplot(224)
  plotCoefficients(coefficients)
  pyplot.draw()

def menu(options, default=3):
  print '<<<< OPTIONS >>>>'
  for i in range(len(options)):
    print str(i+1) + '.', options[i]
  option = raw_input('Enter option (1-'+str(len(options))+'): ')
  try:
    return options[int(option)-1]
  except:
    return options[default-1]

def prompt(message, default=[5], type=int):
  response = raw_input(message)
  if len(response) == 0:
    return default
  try:
    return map(type,response.split())
  except:
    return default

####################
# Model Parameters #
####################

lengths = (5.0e3, 10.0e3, 20.0e3, 40.0e3, 80.0e3, 160.0e3)
mesh_size = ((16,16), (16,16), (32,16), (32,8), (64,8), (64,4))

print '\nPlease select the domain size.\n'
length = menu(lengths)

nx, ny = mesh_size[lengths.index(length)]
```

```
height = 1000.0      # Ice thickness
n = 3.0              # Exponent in Glen's flow law
A = 1.0e-16          # Flow parameter
B = A**(-1.0/n)      # Coefficient in effective viscosity
rho = 910.0          # Density of ice
g   = 9.81           # Gravitational constant
alpha = 0.1*pi/180.0 # Angle of inclination

string = '\nPlease enter the number of terms to be used in the ' +\
         'trigonometric expansion of beta squared: '
nTerms = prompt(string, default=[nx-1])[0]
coefficients = [1000.0, 1000.0] + (nTerms-2)*[0.0]

beta_squared = '(c[0]'
if len(coefficients) > 1:
  for i in range(1,len(coefficients)):
    omega = 2*((i+1)/2) * pi / length
    if i % 2 == 1:
      string = '%s + c[%d]*Expression("sin(%s*x[0])")'
    else:
      string = '%s + c[%d]*Expression("cos(%s*x[0])")'
    beta_squared = string % (beta_squared, i, omega)
beta_squared = beta_squared + ')'

print '\nbeta_squared =', beta_squared
print '\nc =', coefficients

#########################################
# Create mesh and define function space #
#########################################

if verbose:
  print '\nCreating the mesh and function space ...'
  stdout.flush()

mesh = RectangleMesh(0.0, 0.0, length, height, nx, ny)

# Classes for boundary conditions

if verbose:
  print 'Defining classes for the boundary conditions ...'
```

```
  stdout.flush()

TOP    = 1
BOTTOM = 2

class TopBoundary(SubDomain):
  def inside(self, x, on_boundary):
    return on_boundary and near(x[1], height)

class BottomBoundary(SubDomain):
  def inside(self, x, on_boundary):
    return on_boundary and near(x[1], 0)

class PeriodicBoundary(SubDomain):
  def inside(self, x, on_boundary):
    return near(x[0],0) and on_boundary

  def map(self, x, y):
    y[0] = x[0] - length
    y[1] = x[1]

boundary = MeshFunction("size_t", mesh, mesh.topology().dim()-1)
boundary.set_all(0)
TopBoundary().mark(boundary, TOP)
BottomBoundary().mark(boundary, BOTTOM)

# The function space accommodates the three unknowns:
# 1. The x component of velocity (u)
# 2. The y component of velocity (v)
# 3. Pressure (p)

pbc = PeriodicBoundary()
velocity = VectorFunctionSpace(mesh, "Lagrange", 2, constrained_domain=pbc)
pressure =       FunctionSpace(mesh, "Lagrange", 1, constrained_domain=pbc)
functionSpace = MixedFunctionSpace((velocity, pressure))
velocity = functionSpace.sub(0)
pressure = functionSpace.sub(1)

bc = DirichletBC(velocity.sub(1), Constant(0), boundary, BOTTOM)
ds = Measure('ds')[boundary]

#######################################
```

```
# Create the trial and test functions #
########################################

if verbose:
  print 'Creating the trial and test functions ...'
  stdout.flush()

uvp = TrialFunction(functionSpace)
phi =  TestFunction(functionSpace)
solution = Function(functionSpace)

(u, v), p          = split(uvp)
(phi1, phi2), phi3 = split(phi)

##################################
# Define the variational problem #
##################################

if verbose:
  print 'Defining the variational problem ...'
  stdout.flush()

def invariant_squared(u,v):
  return 0.50*(u.dx(0)**2 + v.dx(1)**2) \
         + 0.25*(u.dx(1)+v.dx(0))**2 \
         + Constant(1.0e-20)

viscosity = Constant(0.5*B)*invariant_squared(u,v)**((1.0-n)/(2.0*n))

c = [Constant(a) for a in coefficients]

a = ((2*viscosity*u.dx(0)-p)*phi1.dx(0) + viscosity*(v.dx(0)+u.dx(1))*phi1.dx(1)
   + (2*viscosity*v.dx(1)-p)*phi2.dx(1) + viscosity*(u.dx(1)+v.dx(0))*phi2.dx(0)
   + (u.dx(0)+v.dx(1))*phi3 ) * dx \
   + eval(beta_squared) *u*phi1 * ds(BOTTOM)

a -= (Constant(rho*g*sin(alpha))*phi1 - Constant(rho*g*cos(alpha))*phi2) * dx

F = action(a, solution)
J = derivative(F, solution, uvp)
problem = NonlinearVariationalProblem(F, solution, bc, J)
solver  = NonlinearVariationalSolver(problem)
```

```
solver.parameters['newton_solver']['maximum_iterations'] = 10
solver.parameters['newton_solver']['absolute_tolerance'] = 1.0e-3
solver.parameters['newton_solver']['relative_tolerance'] = 5.0e-4

#################################
# Solve the variational problem #
#################################

if verbose:
  print 'Solving the variational problem ...'
  stdout.flush()

set_log_active('l' in command_line_arguments)

# Solve to find the desired surface velocities

solver.solve()
goalFunction = Function(functionSpace)
goalFunction.vector()[:] = solution.vector().array()

# Use BFGS optimization to find the original coefficients

string = '\nPlease enter a guess for the coefficients of the ' +\
         'trigonometric expansion of beta squared: '
coefficients = prompt(string, default=[1000.0], type=float)
if len(coefficients) > nTerms:
  coefficients = coefficients[:nterms]
  print 'Only the first %d terms will be used:' % nTerms
elif len(coefficients) < nTerms:
  coefficients += (nTerms - len(coefficients)) * [0.0]

pyplot.ion()
pyplot.figure(1)
pyplot.clf()

def f(coefficients):
  global functional
  adj_reset()
  for i in range(nTerms):
    c[i].assign(coefficients[i])
  print ('c = ['+len(c)*' %8.5g'+']') % tuple([float(cc) for cc in c])
  stdout.flush()
```

```
  try:
    nIterations = solver.solve()[0]
  except RuntimeError as message:
    print message
    print 'Trying again with an initial guess of zero everywhere.'
    adj_reset()
    solution.vector()[:] = 0.0
    nIterations = solver.solve()[0]
  form = ((solution[0]-goalFunction[0])**2)*ds(TOP)
  functional = Functional(form*dt[FINISH_TIME])
  error = assemble(form)
  plot4(coefficients, error)
  iterates.append((coefficients, error))
  print 'error = %f  (%d iterations)' % (error, nIterations)
  return error

def derivatives(coefficients):
  gradient = list()
  for i in range(nTerms):
    gradient.append(compute_gradient(functional, ScalarParameter(c[i]), forget=False))
  print ('D = ['+nTerms*' %8.5g'+']') % tuple(gradient)
  return array(map(float,gradient))

c[1].assign(0.0)
iterates = list()
solution.vector()[:] = 0.0
f(coefficients)
optimum = fmin_bfgs(f, coefficients, derivatives, gtol=1e-3, maxiter=30)
print 'optimum =', optimum

pyplot.figure(2)
pyplot.clf()
pyplot.subplot(221)
plotUvsX(height)
pyplot.subplot(223)
plotBetaSquared(optimum)
pyplot.subplot(224)
pyplot.plot(abs(optimum[::2]), 'o-', label='cos')
pyplot.plot(range(1,len(optimum[1::2])+1), abs(optimum[1::2]), 'o-', label='sin')
pyplot.yscale('log')
pyplot.legend(loc='best')
pyplot.xlabel('frequency')
```

```
pyplot.grid(True)

print 'min(optimum[2:]) =', min(optimum[2:])
print 'max(optimum[2:]) =', max(optimum[2:])

try:
  __IPYTHON__
except:
  raw_input('Press ENTER to end the program')
```

Bibliography

[1] Cuffey, K. M. and W. S. B. Paterson (2010) *The physics of glaciers.* Fourth edition. Amsterdam, etc., ACADEMIC PRESS.

[2] IPCC (2007) *Climate Change 2007: The Physical Science Basis. Contribution of Working Group I to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change* [Solomon, S., D. Qin, M. Manning, Z. Chen, M. Marquis, K.B. Averyt, M.Tignor and H.L. Miller (eds.)], CAMBRIDGE UNIVERSITY PRESS.

[3] Bindschadler, Robert A., Sophie Nowicki, Ayako Abe-Ouchi, Andy Aschwanden, Hyeungu Choi, Jim Fastook, **Glen Granzow**, Ralf Greve, Gail Gutowski, Ute Herzfeld, Charles Jackson, Jesse Johnson, Constantine Khroulev, Anders Levermann, William H. Lipscomb, Maria A. Martin, Mathieu Morlighem, Byron R. Parizek, David Pollard, Stephen F. Price, Diandong Ren, Fuyuki Saito, Tatsuru Sato, Hakime Seddik, Helene Seroussi, Kunio Takahashi, Ryan Walker, Wei Li Wang (2013) *Ice-sheet model sensitivities to environmental forcing and their use in projecting future sea level (the SeaRISE project).* JOURNAL OF GLACIOLOGY, Vol. 59, No. 214, pp 195-224.

[4] Greve, Ralf and Heinz Blatter (2009) *Dynamics of Ice Sheets and Glaciers.* Dordrecht, etc., SPRINGER.

[5] van der Veen, C. J. (2013) *Fundamentals of Glacier Dynamics.* Boca Raton, CRC PRESS.

[6] Joughin, I., B.E. Smith, I.M. Howat, T. Scambos, T. Moon (2010) *Greenland Flow Variability from Ice-Sheet-Wide Velocity Mapping.* JOURNAL OF GLACIOLOGY, 56 (197), pp. 415-430.

[7] Logg, A., K. A. Mardal, G. N. Wells et al. (2012) *Automated Solution of Differential Equations by the Finite Element Method.* Heidelberg, etc., SPRINGER.

[8] Logg, A. and G. N. Wells (2010) *DOLFIN: Automated Finite Element Computing.* ACM Transactions on Mathematical Software, 37(2).

[9] Farrell, Patrick E., David A. Ham, Simon W. Funke and Marie E. Rognes (2012) *Automated derivation of the adjoint of high-level transient finite element programs.* arXiv:1204.5577

[10] Weertman, J. (1973) *Creep in Ice.* In Whalley, E., S. J. Jones and L. W. Gold (editors), *Physics and Chemistry of Ice*, Ottawa, Royal Society of Canda, 320-337.

[11] Joughin, I., I. Howat, B. Smith, T. Scambos (2011) *MEaSUREs Greenland Ice Velocity: Selected Glacier Site Velocity Maps from InSAR.* Boulder, Colorado, National Snow and Ice Data Center. Digital media.

[12] Bamber, J. L., J. A. Griggs, R. T. W. L. Hurkmans, J. A. Dowdeswell, S. P. Gogineni, I. Howat, J. Mouginot, J. Paden, S. Palmer, E. Rignot, and D. Steinhage (2013) *A new bed elevation dataset for Greenland.* The Cryosphere, 7, 499-510.

[13] Nocedal, Jorge, Stephen J. Wright (1999) *Numerical Optimization.* New York, etc., Springer.

[14] Goldsby, D. L., and D. L. Kohlstedt (2001) *Superplastic deformation of ice: Experimental observations.* Journal of Geophysical Research, 106, 11,017-11,030.

[15] Errico, Ronald M. (1997) *What Is an Adjoint Model?* Bulletin of the American Meteorological Society, Vol. 78, No. 11.

[16] Strang, Gilbert (2007) *Computational Science and Engineering.* Wellesley-Cambridge Press.

[17] Pattyn, F., L. Perichon, A. Aschwanden, B. Breuer, B. de Smedt, O. Gagliardini, G. H. Gudmundsson, R. C. A. Hindmarsh, A. Hubbard, J. V. Johnson, T. Kleiner, Y. Konovalov, C. Martin, A. J. Payne, D. Pollard, S. Price, M. Ruckamp, F. Saito, O. Souek, S. Sugiyama, and T. Zwinger (2008) *Benchmark experiments for higher-order and full-Stokes ice sheet models (ISMIPHOM).* The Cryosphere, 2, 95108.

[18] Press, William H., Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery (2007) *Numerical Recipes. The Art of Scientific Computing*, Third Edition. Cambridge University Press.