The School of Mathematics

THE UNIVERSITY
of EDINBURGH

Automated Code Generation for Simplified Glaciological Models

Dissertation Presented for the Degree of
MSc in Computational Applied Mathematics

August 2019

Abstract

To numerically study complex and diverse differential equations, the need for automatic generation of specialized code is imperative. This relieves the need for users to write specialized code and also affords flexibility in being able to solve various differential equations without reliance on specialized libraries. In this report we use software from the FEniCS Project to study finite element discretizations of a shallow shelf model and a hybrid L1L2 model of glaciological flow. For the shallow shelf model, FEniCS allows for linearization of the variational form of the problem so that we may directly apply a Newton solver to find a solution. Newton's method is unstable for high-resolution domains and so we introduce a first-order approximated Newton's method. For the hybrid model, it is necessary to construct a fixed-point iteration on velocity to compute a solution to the problem. This process is made efficient FEniCS's use of JIT compilation, which allows for reuse of generated code through the iterations.

Acknowledgments

Own Work Declaration

I declare that the following work is my own except where otherwise noted.

Contents

List of Figures

1 Introduction

Partial differential equations (PDEs) are used in myriad disciplines to model physical phenomena, including fluid dynamics, electromagnetism, and mathematical biology. Analytic techniques to solve PDEs are useful but limited to being able to solve only very simple models. This has resulted in the continuous development of software and libraries developed for the solution of PDEs. The finite element method is a technique for converting PDEs into a computable form. There is a balance to be struck with these numerical methods, with a common belief that high performance requires specialization of libraries suited for only particular problems without the flexibility to adapt to new demands. Over time the belief in the necessity of specialization has waned with developments in automatic generation of optimized code [14, 15, 16, 26]. While there exist various software aimed at automating the finite element method, including FreeFEM++ [28] and Sundance [22], in this report we will focus on the FEniCS Project, an open-source software for solving PDEs [6].

The advantage of FEniCS over other software is that it uses compiler technology to generate software where possible instead of just compiling and integrating hand-coded software libraries [30]. A critical discovery was an optimized method of computing finite element matrices that made finite element computation as efficient as finite different computation while preserving the geometric generality unique to the finite element method [15, 16]. FEniCS also makes use of Just-in-Time (JIT) compilation [21], which allows the user to modify and quickly re-simulate the models used in the program. Thus, FEniCS allows a much larger array of models to be explored. As the first system to implement the full Periodic Table of Finite Elements [2], FEniCS also allows users to implement more complicated models utilizing elements that were not previously available. Additionally, FEniCS supports generating finite element meshes and also supports interfacing with components in different domains, for example, importing meshes from other packages such as Gmsh [7].

To demonstrate the use of FEniCS in solving real-world problems, we will examine PDE systems for several simplified glaciological models.

1.1 Glaciological models

We now present a brief overview of glaciological models which are studied in Chapters 3–4.

Figure 1 shows the cross section of a grounded ice sheet with attached floating ice shelf in a Cartesian coordinate system. Here x and y are in the horizontal plane and z is positive upward. Then the first-order momentum balance equations in Cartesian coordinates are given by the Blatter-Pattyn equations [4, 27]:

$$\partial_x\big[\nu(4u_x + 2v_y)\big] + \partial_y\big[\nu((u_y + v_x)\big] + \partial_z(\nu u_z) = \rho g s_x, \tag{1.1}$$

$$\partial_x\big[\nu(v_x + u_y)\big] + \partial_y\big[\nu((4v_y + 2u_x)\big] + \partial_z(\nu v_z) = \rho g s_y, \tag{1.2}$$

$$\nu = \frac{B}{2}\left[u_x^2 + v_y^2 + u_x v_y + \frac{1}{4}\big(u_y + v_x\big)^2 + \frac{1}{4}u_z^2 + \frac{1}{4}v_z^2\right]^{\frac{1-n}{2n}}, \tag{1.3}$$

where u and v are velocities in the x and y directions, respectively, ν is the effective viscosity of the ice, s is the elevation of the upper ice surface, ρ is the density, g is the gravitational acceleration, n is set equal to 3, and B is the temperature-dependent rate factor. Simplified models reduce the complexity of this problem at the cost of physical realism. Let us now examine two simplified glaciological models: the Shallow Shelf Approximation and a hybrid L1L2 model.

The Shallow Shelf Approximation (SSA) is an effective "sliding law" where basal stress is zero and longitudinal stresses dominate [5]. It is a standard model for ice stream flow where the sliding velocity is large. It is a vertically integrated, 2-dimensional model derived in [24] of lower computational dimension than (1.1)–(1.3) that still indicates the effects of horizontal stress.

However, it does not capture the effect of vertical shear and is less effective in regions where vertical variations in speed are important. A full mathematical formulation of the model is given in Chapter 3 and in [10].

The hybrid L1L2 model [8] accounts for vertical shear in the stress balance. It is a 2-dimensional simplification of (1.1)–(1.3). By vertically integrating these equations, a system of 2-D elliptic PDEs for the horizontal components of the velocity are derived. A more detailed mathematical formulation is provided in Chapter 4 and in [8]. This formulation is similar to other momentum balances such as that in [13], but the derivation from [8] is from a variational principle. The numerical implementation of this model solves the system of PDEs in an iterative loop. This scheme is similar to the "iteration on viscosity" method of solving the SSA balance [23].

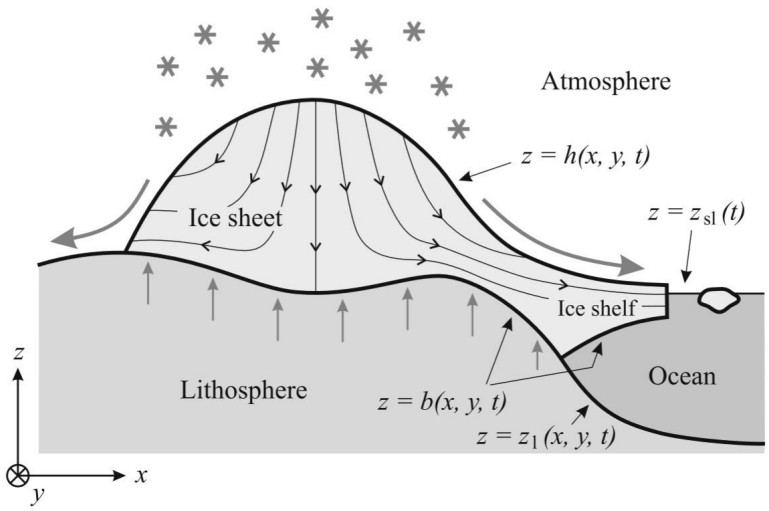

Figure 1: Geometry of glacier with attached ice shelf[1].

1.2 Outline of thesis

The remainder of this report is organized as follows. In Chapter 2 we examine the use of FEniCs to solve partial differential equations. We focus on the DOLFIN and UFL libraries and study key tols needed to define and solve weak forms of PDEs. We then solve several demo problems, including the Poisson problem and the heat equation, to explore the problem-solving abilities of FEniCS. In Chapter 3, which largely follows [25], we present an implementation of the shallow shelf model. We examine the use of a first-order method that is used as an approximation to Newton's method. In Chapter 4 we develop an implementation of the hybrid L1L2 model presented in [8]. Finally we conclude this report with Chapter 5, where we summarize the results and provide possible avenues for future research.

[1]Taken from [12]: Fig. 5.1

2 FEniCS

In this section we will discuss the use of FEniCS, a platform for solving PDEs using the finite element method. We will focus in particular on the libraries DOLFIN and UFL. DOLFIN is the main programming interface and problem-solving environment of FEniCS. It generates problem-specific code and computes the automated solution of differential equations using the finite element method. DOLFIN uses UFL to express variational forms in a language close to mathematical notation. After introducing key features of these two libraries, we discuss several problems that demonstrate how to use these libraries to solve well-known differential equations.

2.1 Code generation

2.1.1 DOLFIN

This section largely follows [21]. DOLFIN is a library available as a part of the FEniCS Project that computes the automated solution of PDEs using the finite element method. It is the main programming interface and problem-solving environment of the FEniCS Project. DOLFIN relies on a form complier to generate problem-specific code. The expression of variational forms is handled by the UFL library, as discussed in Section 2.1.2.

Automated code generation allows DOLFIN to efficiently assemble a variational form of any rank (i.e., of any number of arguments) from a large class of variational forms. In order to have efficient code generation, it is necessary to isolate the parts of the program for which code must be generated, and make use of reusable library components in a general purpose language for the remaining parts of the program. DOLFIN partitions user input into two subsets: data that may only be handled efficiently by special purpose code, and data that can be handled efficiently by general purpose library components, which may be implemented as reusable library components in a general purpose language. The former may include a finite element variational problem and the finite elements used to define it. The latter includes data structures and algorithms for linear algebra, computational meshes, and representations of functions. The assembly algorithm, and in particular the innermost loop of the assembly algorithm, varies for each problem depending on, but not limited to, the particular formulation and choice of finite element function space of each particular problem. As an efficient and general solution, DOLFIN utilizes reusable components at higher levels and relies on a form compiler to generate the code for the innermost loop from a user-defined variational form. If this form compuler is implemented as a just-in-time (JIT) compiler, it is possible to automatically generate, compile, and execute generated code at run-time on demand.

The central classes of DOLFIN include the linear algebra classes, mesh classes, finite element classes, and function classes. The interaction between these classes is visualized in Figure 2. DOLFIN assembles a user-defined variational form on any mesh for a wide range of finite elements using any user-defined or built-in linear algebra backend. The linear algebra classes consist mostly of wrapper classes for external libraries, including PETSc [3], uBLAS [31], and MTL4 [11]. Because the linear algebra interface is implemented based on C++ polymorphism, DOLFIN is able to implement a common assembly algorithm for all matrices, vectors, and scalars for all linear algebra backends. Additionally, the assembly algorithm is common for all simplex meshes in one, two, and three space dimensions. DOLFIN supports a wide range of finite elements, as detailed in Section 4.3 of [21], which are not included in a library of finite elements but rather, implemented by a form compiler.

Finite element assembly The `assemble` function is one of the free functions provided by DOLFIN. Given a variational form, this function assembles a matrix from a bilinear form, a vector from a linear form, and a scalar value from a functional form. The DOLFIN assembly algorithm is both general and efficient due to the automated generation of code for the evaluation of the element tensor, as well as for the mapping of degrees of freedom. Thus, the algorithm may

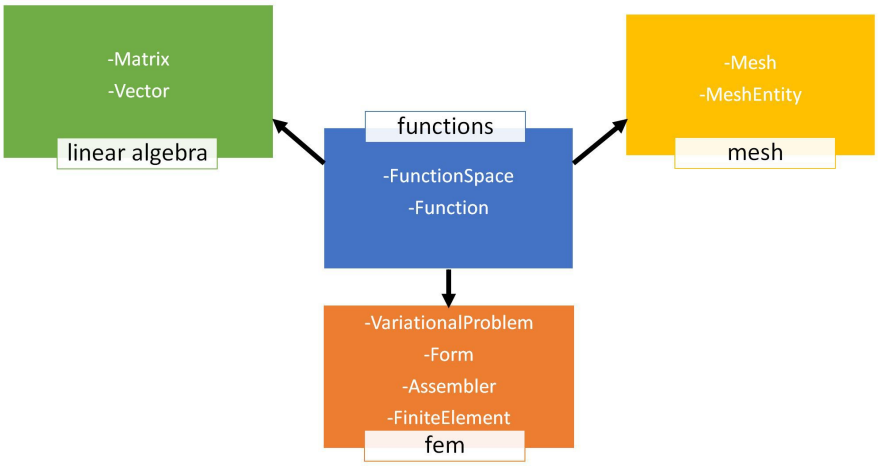

Figure 2: UML diagram of the central components and classes of DOLFIN[2].

call the generated code on each cell of the mesh regardless of the complexity of the form, since it simply provides coefficient and mesh data to the generated code and assembles the computed results. The algorithm for computing the element tensor is determined by the form compiler.

To assemble a `Form a`, the user may simply use the statement

```
A = assemble(a)
```

and the bilinear form will be given by `A`.

Solving variational forms The `solve` function is another free function provided by DOLFIN, and is utilized in many of the problems examined further in this report. It may be used to solve either linear systems or variational problems. A linear variational problem of the form $a(u, v) = L(v)$ for all v may be solved with the statement

```
solve(a == L, u, ...)
```

where `a` is a bilinear form, `L` is a linear form, and `u` is a `Function` that contains the solution. Other optional arguments may be supplied to specify boundary conditions or solver parameters. Solver parameters include the absolute and relative tolerance and the maximum number of iterations. An example of usage of this statement is given below:

```
solve(a == L, u, bcs=bcs,
      solver_parameters={"linear_solver": "lu","maximum_iterations":1e3})
```

where `bc` is some specified boundary condition.

We may also solve nonlinear variational problems of the form

$$F(u; v) = 0 \text{ for all } v$$

with the statement

```
solve(F == 0, u, ...)
```

where `F` is linear in the test function v but may be nonlinear in u, and `u` is a `Function` that contains the solution. Optional arguments that may be included are boundary conditions, the Jacobian, or solver parameters. If the Jacobian is not supplied then it is computed through automatic differentiation of the residual `F`. An example of usage of this statement is given below:

```
solve(F == 0, u, bcs, J=J,
      solver_parameters={"linear_solver": "lu"},
      form_compiler_parameters={"optimize": True})
```

[2]Adapted from Fig. 3 of [21].

2.1.2 Unified Form Language (UFL)

This section largely follows [20]. The Unified Form Language, or UFL, is a domain specific language that defines the language used to express PDEs as finite element discretizations of variational forms. It also provides algorithms that the form compiler can use to simplify the compilation process and create output that is usable by DOLFIN to efficiently assemble linear systems and compute solutions to PDEs.

Two main goals motivating the development of UFL are automatic differentiation of expressions and forms and improving the performance of the form compiler to handle more complicated equations efficiently.

Defining forms Forms expressed in UFL are intended for finite element discretization followed by compilation to efficient code for computing the element tensor. In general, UFL is designed to express forms of the generalized form

$$a(w^1, \ldots, w^n; \phi^1, \ldots, \phi^r) = \sum_{k=1}^{n_c} \int_{\Omega_k} I_k^c \; dx + \sum_{k=1}^{n_e} \int_{\partial\Omega_k} I_k^e \; ds + \sum_{k=1}^{n_i} \int_{\Gamma_k} I_k^i \; dS.$$

Here, Ω denotes the entire domain, $\partial\Omega$ denotes the external boundary of the domain, and Γ denotes the set of interior facets of the triangulation. Integration is expressed by multiplication with a measure, with UFL supporting the measures `dx`, `ds`, and `dS` which correspond to cell integrals, exterior facet integrals, and interior facet integrals respectively.

Consider the Poisson equation given (17.8)–(17.9) of [20] with two different boundary conditions on $\partial\Omega_0$ and $\partial\Omega_1$:

$$a(w; u, v) = \int_\Omega w \nabla u \cdot \nabla v \; dx, \tag{2.1}$$

$$L(f, g, h; v) = \int_\Omega f v \; dx + \int_{\Omega_0} g^2 v \; ds + \int_{\Omega_1} h v \; ds. \tag{2.2}$$

The two forms above can be expressed in UFL as

```
a = w*dot(grad(u), grad(v))*dx
L = f*v*dx + g**2*v*ds(0) + h*v*ds(1)
```

The form arguments are divided into two groups: the basis functions $\{\phi^1, \ldots, \phi^r\}$ and the coefficient functions $\{w^1, \ldots, w^n\}$, which are all functions in some discrete function space with a basis. A form with one or two basis function arguments, i.e., $r = 1, 2$, is called a linear or bilinear form respectively and is assembled to vectors and matrices respectively. A form depending only on coefficient functions, i.e., $r = 0$ is called a functional. The programs included in this project deal with these three forms.

Defining expressions Basis function arguments represent any function ϕ^j in the basis of the finite element space $V^{\delta,j}$:

$$\phi^j \in \{\phi_k^j\}, \quad V^{\delta,f} = \text{span}\{\phi_k^j\}.$$

Functions in the finite element space are represented by `Argument`. The ordering of the arguments to a form is determined by the order in which the form arguments were declared in the UFL code. However, we can use `TestFunction` and `TrialFunction` in declarations of functions instead of `Argument` in order to ignore the relative ordering of the arguments. `Argument` is only needed for forms with arity $r > 2$, a case not addressed in this report. In the code excerpt below, `phi`, `v`, and `u` are all functions in the finite element space `V`:

```
phi = Argument(V)
v = TestFunction(V)
u = TrialFunction(V)
```

Coefficient functions are represented by `Coefficient`, where each coefficient function w represents a discrete function in some finite element space V^δ, usually a sum of basis functions $\phi_k \in V^\delta$ with coefficients w^k:

$$x = \sum_{k=1}^{|V^\delta|} w_k \phi_k.$$

Coefficient problems can then be declared as

```
w = Coefficient(V)
c = Constant(cell)
```

It is also possible declare mixed finite element spaces of the form $V^\delta = V^{\delta,0} \times V^{\delta,1}$ and create form arguments in the mixed finite element space. We can then extract subfunctions using `split`, which can handle arbitrary mixed elements:

```
V = V0.ufl_element() * V1.ufl_element()
u = Coefficient(V)
u0, u1 = split(u)
```

A shorthand notation for splitting arguments is

```
v0, v1 = TestFunctions(V)
u0, u1 = TrialFunctions(V)
f0, f1 = Coefficients(V)
```

Differential operators UFL implements derivatives with respect to three kinds of variables: spatial derivatives, user-defined variables, and derivatives of a form or function with respect to the coefficients of a discrete function, i.e., a `Coefficient` or `Constant`.

Spatial derivatives of the form $\frac{\partial f}{\partial x_i}$ can be expressed as:

```
df = Dx(f, i)
# or, equivalently
df = f.dx(i)
```

where `df` represents the derivative of `f` in the spatial direction `x_i`. UFL also has several compound spatial derivative operators: `div`, `grad`, and `curl`.

User-defined variables can be used to represent arbitrary expressions. For example, let g be an arbitrary expression assignmented to a variable v and an expression $f(v)$ be defined as a function of v. It is then possible to implement differentiation of f with respect to the user-defined variable v:

```
g = sin(cell.x[0])
v = variable(g)
f = exp(v**2)
h = diff(f, v)
```

For this example, we have $g = \sin x_0$, $f = e^{v^2}$, and $h = \frac{\partial f(v)}{\partial v}$.

Differentiating forms The form operator `derivative` declares the derivative of a form with respect to a `Coefficient` representing coefficients of a discrete function. This is important in Chapter 3 where we linearize a nonlinear variational form. The derivative of a form `L` with respect to the coefficients of a function `w` is calculated by

```
a = derivative(L, w, u)
```

where `u` is a basis function argument in the same finite element space as `w`.

Consider for example a finite element space V^δ with some basis, a function w in V^δ, and a function $f = f(w)$ we want to minimize, i.e., we seek

$$F(w; \phi_i) = \frac{\partial f(w)}{\partial w_i}, \quad i = 1, \ldots, \left|V^\delta\right|$$

where $V^\delta = \text{span}\{\phi_k\}$. Using the functional $f(w) = \int_\Omega \frac{1}{2} w^2 \, dx$, this can be implemented as

```
v = TestFunction(V)
w = Coefficient(V)
f = 0.5 * w**2 * dx
F = derivative(f, w, v)
```

where we find `F = w*v*dx`

2.2 Demo problems

Let us now consider some example problems that explore the functionality of FEniCS as described in the previous section. These example problems are adapted from demo problems presented in [18] and [19].

2.2.1 Poisson equation

In this section we use FEniCS to solve the Poisson equation, the boundary value problem given by

$$-\nabla^2 u(\mathbf{x}) = f(\mathbf{x}), \qquad \mathbf{x} \text{ in } \Omega \tag{2.3}$$

$$u(\mathbf{x}) = u_D(\mathbf{x}), \qquad \mathbf{x} \text{ on } \partial\Omega, \tag{2.4}$$

where $u(\mathbf{x})$ is the unknown function, $f(\mathbf{x})$ is a prescribed function, ∇^2 is the Laplace operator, Ω is the spatial domain, and $\partial\Omega$ is the boundary of Ω [19, Chapter 2].

We must first find the variational formulation of this problem. The general procedure to do this is to multiply the PDE by a test function v, integrate the resulting equation over the domain Ω, optionally perform integration by parts of terms with second-order derivatives, and optionally apply natural boundary conditions. Applying this procedure to the Poisson equation, we first multiply (2.3) by the test function v and integrate over the domain Ω:

$$-\int_\Omega (\nabla^2 u) v \, dx = \int_\Omega f v \, dx. \tag{2.5}$$

Note that we use dx to represent the differential element for integration over the domain Ω and ds to represent the differential element for integration over $\partial\Omega$ We now apply integration by parts to the second-order derivative of u:

$$-\int_\Omega (\nabla^2 u) v \, dx = \int_\Omega \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} \frac{\partial u}{\partial n} v \, ds, \tag{2.6}$$

where n is the outward normal direction on the boundary.

We next apply the requirement that the test function v vanishes on the parts of the boundary where the solution u is known, which for the Poisson problem is given by (2.4). Therefore this means that $v = 0$ on the boundary $\partial\Omega$, and so (2.6) simplifies to

$$-\int_\Omega (\nabla^2 u) v \, dx = \int_\Omega \nabla u \cdot \nabla v \, dx, \tag{2.7}$$

which we substitute into (2.5) to obtain the variational form of the original boundary-value problem:

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx. \tag{2.8}$$

In order to have a unique solution u that lies in some infinite-dimensional function space V, called the *trial space*, we must require that (2.8) holds for all test functions v in some suitable space $\hat{V}$, called the *test space*.

The final variational problem is stated as: find $u \in V$ such that

$$\int_\Omega \nabla u \cdot \nabla v \; dx = \int_\Omega f v \; dx \quad \forall v \in \hat{V}, \tag{2.9}$$

where the trial and test spaces V and $\hat{V}$, respectively, are defined as

$$V = \{v \in H^1(\Omega) : v = u_D \text{ on } \partial\Omega\},$$
$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}.$$

The finite element method finds an approximate solution of the continuous variational problem described by (2.9) by replacing the infinite-dimensional function spaces V and $\hat{V}$ by discrete trial and test spaces $V^\delta \subset V$ and $\hat{V}^\delta \subset \hat{V}$. Thus, the *discrete* variational problem is stated as: find $u^\delta \in V^\delta$ such that

$$\int_\Omega \nabla u^\delta \cdot \nabla v \; dx = \int_\Omega f v \; dx \quad \forall v \in \hat{V}^\delta. \tag{2.10}$$

We have now defined a unique approximate numerical solution of the Poisson equation as a discrete variational problem that FEniCS can automatically solve.

For the implementation of this problem, let us choose $u(x, y) = 1 + x^2 + 2y^2$ as the solution to the problem. Then it must be that $u_D(x, y) = 1 + x^2 + 2y^2$ and $f(x, y) = -6$.

Let us now examine an implementation of the above problem in FEniCS 2019.1.0, modified from demo_poisson.py from Section 16 of [1].

```
from fenics import *

# create mesh and define function space
mesh = UnitSquareMesh(8,8)
V = FunctionSpace(mesh,'P',1)

# define boundary conditions: u_d = 1 + x^2 + y^2
u_d = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]',degree=2)

# function to define boundary value on boundary
def boundary(x,on_boundary):
  return on_boundary

bc = DirichletBC(V,u_d,boundary)

# define test and trial spaces
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(-6.0) # rhs of strong form
lhs = dot(grad(u),grad(v))*dx
rhs = f*v*dx # rhs of weak form

# compute solution
u = Function(V)
solve(lhs == rhs,u,bc) # u contains the answer

# writing out the answer:
File("poisson1.pvd") << (u,0.0) # 2nd arg is timestamp
```

The first line

```
from fenics import *
```

imports classes from the FEniCS library. The line

```
mesh = UnitSquareMesh(8, 8)
```

defines a uniform finite element mesh over the unit square with the parameters specifying that the square should be divided into 8×8 rectangles. Each rectangle is divided into a pair of triangles so that the total number of triangles, or cells, in the mesh is 128, and the total number of vertices in the mesh is $9 \times 9 = 81$.

After the mesh is created, we create a finite element function space V with the statement

```
V = FunctionSpace(mesh,'P',1)
```

The first argument specifies the mesh on which the function space is to be defined. The second argument specifies the type of element, where in this case `'P'` indicates the standard Lagrange family of elements. The third argument specifies the degree of the finite element.

In the variational problem we have the trial and test spaces V and $\hat{V}$, with the only difference between them being their boundary conditions. In FEniCS the boundary conditions are not specified as part of the function space, so the test and trial functions may be constructed on the same space, `V`:

```
u = TrialFunction(V)
v = TestFunction(V)
```

We then define the boundary condition $u = u_D$ on $\partial\Omega$ with the statement

```
bc = DirichletBC(V,u_d,boundary)
```

The first argument is the function space on which the boundary condition is defined. The second argument `u_D` is an `Expression` object that defines the values of the solution on the boundary. `Expression` objects are used to represent mathematical functions and are constructed with a string containing a mathematical expression written with C++ syntax. The typical construction for `u_D` is

```
u_D = Expression(formula, degree=1)
```

where `formula` is a string containing a mathematical expression in C++ syntax. In this problem, we have $u_D(x, y) = 1 + x^2 + y^2$, which is written in FEniCS as

```
u_d = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]',degree=2)
```

The degree is chosen to be 2 so that `u_D` may represent the exact quadratic solution to the test problem. The third argument `boundary` is a function or object that defines points belonging to the boundary where the boundary condition should be applied:

```
def boundary(x,on_boundary):
  return on_boundary
```

This is a boolean function that returns `True` if the given point `x` lies on the Dirichlet boundary and `False` otherwise. The argument `on_boundary` is `True` if `x` is on the physical boundary of the mesh. Therefore, since we seek to return `True` for all points on the boundary, it suffices to return the supplied value of `on_boundary`.

We now define the source term $f = -6$:

```
f = Constant(-6.0)
```

We can now define the variational problem:

```
a = dot(grad(u),grad(v))*dx
L = f*v*dx
```

where `a` is the bilinear form $a(u, v)$ and `L` is the linear form $L(v)$.

FEniCS can now compute the solution, which we store in the variable `u` with the statements:

```
u = Function(V)
solve(a == L,u,bc)
```

Here we have redefined the variable `u`, which was initially used as a `TrialFunction` to represent the unknown in the bilinear form `a`. It is now redefined as a `Function` object used to represent the solution. Finally, FEniCS handles solving the finite element variational problem using the `solve` command as detailed in Section 2.1.1.

2.2.2 Heat equation

In this section we expand upon the properties of FEniCS explored with the Poisson problem by studying the time-dependent heat equation [19, Chapter 3.1]. The main idea is to solve a sequence of variational problems while applying standard time-stepping methods.

The time-dependent heat equation is given by:

$$\frac{\partial u}{\partial t} = \nabla^2 u + f \qquad \text{in } \Omega \times (0, T], \tag{2.11}$$

$$u = u_D \qquad \text{on } \partial\Omega \times (0, T], \tag{2.12}$$

$$u = u_0 \qquad \text{at } t = 0, \tag{2.13}$$

where u varies with space and time, the source function f and the boundary balues u_D may also vary with space and time, and the initial condition u_0 is a function of space.

To solve this time-dependent PDE, we may discretize the time derivative by a finite difference approximation. Then at each discrete timestep, we will have a stationary problem which we may turn into a variational formulation and solve in the same fashion as the Poisson problem. Let us use the superscript n to denote a quantity at time $t = t_n$. Then at time level $t = t_{n+1}$, we have

$$\left(\frac{\partial u}{\partial t}\right)^{n+1} = \nabla^2 u^{n+1} + f^{n+1}. \tag{2.14}$$

Let us then approximate the time derivative using a backward Euler discretization:

$$\left(\frac{\partial u}{\partial t}\right)^{n+1} \approx \frac{u^{n+1} - u^n}{\Delta t}, \tag{2.15}$$

where Δt is a unit of discretized time. We then obtain the backward Euler discretization of the heat equation:

$$\frac{u^{n+1} - u^n}{\Delta t} = \nabla^2 u^{n+1} + f^{n+1}. \tag{2.16}$$

We now rearrange (2.16) so the left-hand side contains all the terms with the unknown u_{n+1} and the right-hand side contains only computed terms:

$$u_0 = u_0, \tag{2.17}$$

$$u^{n+1} - \Delta t \nabla^2 u^{n+1} = u^n + \Delta t f^{n+1}, \quad n = 0, 1, 2, \ldots \tag{2.18}$$

We thus have a sequence of stationary problems for u_{n+1}, which we may solve given u_0. To get the weak form of this problem, we multiply (2.18) by a test function $v \in \hat{V}$ and integrate over the domain Ω. If we let the variable u denote u_{n+1} then we obtain

$$\int_\Omega (uv + \Delta t \nabla u \cdot \nabla v)\, dx = \int_\Omega (u^n + \Delta t f^{n+1}) v\, dx, \tag{2.19}$$

which can be written as

$$a(u, v) = L_{n+1}(v),$$

where the bilinear form $a(u, v)$ is the left-hand side of (2.19) and the linear form $L_{n+1}(v)$ is the right-hand side.

We must also approximate the initial condition given by (2.17), which can be turned into the

variational problem

$$\int_\Omega ux\ dx = \int_\Omega u_0 v\ dx. \tag{2.20}$$

This is exactly the Galerkin L^2 projection of the initial value u_0 onto the finite element space. In FEniCS, we can calculate u^0 by using the `project` or `interpolate` function on u_0.

For the implementation, let us consider the test problem $u(x, y, t) = 1 + x^2 + \alpha y^2 + \beta t$. Then by substituting this into the heat equation we find that we must have the boundary value $u_D(x, y, t) = 1 + x^2 + \alpha y^2 + \beta t$, the initial value $u_0(x, y) = 1 + x^2 + \alpha y^2$, and the source function $f(x, y, t) = \beta - 2 - 2\alpha$.

Let us now consider an implementation of the time-dependent heat equation using FEniCS 2019.1.0, modified from ft03_heat.py from the section "The heat equation" of [18].

```
from fenics import *

T = 2.0
num_steps = 10
dt = T/num_steps

alpha = 3; beta = 1.2

mesh = UnitSquareMesh(10,10)
space = FunctionSpace(mesh,'P',1)

u_d = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',degree=2,alpha=alpha
    ,beta=beta,t=0)

def boundary(x,on_boundary):
  return on_boundary

bc = DirichletBC(space,u_d,boundary)

# define u_n: u_0 is the projection of the initial u_d onto the function space
u_n = project(u_d,space)

# define test and trial spaces
u = TrialFunction(space)
v = TestFunction(space)
f = Constant(beta - 2 - 2*alpha)

F = u*v*dx + dt*dot(grad(u),grad(v))*dx - (u_n + dt*f)*v*dx
a,L = lhs(F),rhs(F)

u = Function(space)
t = 0

for n in range(num_steps):
  t += dt
  u_d.t = t

  solve(a == L,u,bc)

  # Update previous solution
  u_n.assign(u)
```

The new consideration in this problem is the variation in time, which affects the boundary condition $u_D(x, y, t)$. To address this, we may create an `Expression` object that has time t as a parameter:

```
u_d = Expression('1 + x[0]*x[0] + alpha*x[1]*x[1] + beta*t',degree=2,alpha=alpha
    ,beta=beta,t=0)
```

Here `t` is initialized as `t = 0`, and it can be updated later by

```
u_d.t = t
```

The boundary and the boundary conditions are implemented in the same way as with the Poisson problem. We then use the variable `u` to denote the unknown variable u_{n+1} at the new time step and `u_n` for u_n. As stated previously, we can calculate u^0 by using the `project` or `interpolate` function on u_0. In this case, we have chosen `project`:

```
u_n = project(u_d,space)
```

although we could have alternatively used

```
u_n = interpolate(u_d,space)
```

`project(u,space)` finds the Galerkin L^2 projection of `u` onto the discretized finite element space `space`. Therefore, it results in approximate values at the nodes. `interpolate(u,space)` evaluates `u` at each degree of freedom of `space` and so ensures (to machine precision) values at the nodes.

Previously for the Poisson problem, we directly specified the bilinear form a and the linear form L. For this problem, we demonstrate a convenient feature of FEniCS where we may define an abstract formulation of the form

$$F(u;v) = 0$$

and supply F to FEniCS. FEniCS will then automatically determine which terms in F go into the bilinear form and which go into the linear form. For our problem, we can rearrange (2.19) into the form

$$F_{n+1}(u;v) = \int_\Omega \left(uv + \Delta t \nabla u \cdot \nabla v - (u^n + \Delta t f^{n+1})v\right)\, dx, \tag{2.21}$$

so that

$$F_{n+1}(u;v) = 0.$$

Then a and L can be constructed simply by defining F:

```
u = TrialFunction(space)
v = TestFunction(space)
f = Constant(beta - 2 - 2*alpha)

F = u*v*dx + dt*dot(grad(u),grad(v))*dx - (u_n + dt*f)*v*dx
a,L = lhs(F),rhs(F)
```

Finally, as before, we use `u` to store the solution, and we implement a time-stepping loop to compute the solution at each time:

```
u = Function(space)
t = 0

for n in range(num_steps):
  t += dt
  u_d.t = t

  solve(a == L,u,bc)

  # Update previous solution
  u_n.assign(u)
```

Note that in the last line, the variable `u_n` is used to store the solution at the previous time step. It is necessary to use the `assign` member function here in order to have `u_n` as a separate variable from `u` that stores the values of `u`.

2.2.3 Nonlinear Poisson equation

In this section we study how to solve nonlinear PDEs using FEniCS.

Let us consider the nonlinear Poisson equation presented in [19, Chapter 3.2]:

$$-\nabla \cdot \big((q(u)\nabla u)\big) = f, \tag{2.22}$$

in the domain Ω with $u = u_D$ on the boundary of the domain $\partial\Omega$. We assume $q(u)$ is not constant in u so the equation is nonlinear. To find the variational formulation, we multiply the PDE by a test function $v \in \hat{V}$ and integrate over the domain. The variational formulation of the problem is: find $u \in V$ such that

$$\int_\Omega (q(u)\nabla u \cdot \nabla v - fv)\, dx = 0 \quad \forall v \in \hat{V}, \tag{2.23}$$

and

$$V = \{v \in H^1(\Omega) : v = u_D \text{ on } \partial\Omega\},$$
$$\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}.$$

We can define the left-hand side of (2.23) as $F(u; v)$:

$$F(u; v) = \int_\Omega (q(u)\nabla u \cdot \nabla v - fv)\, dx \tag{2.24}$$

As before, we discretize the problem by restricting V and $\hat{V}$ to a pair of discrete spaces.

For the implementation, take $q(u) = 1 + u^2$ and $u(x, y) = 1 + x + 2y$. Then by plugging these equations into (2.22) we find we must have $f = -10x - 20y - 10$.

Let us now examine an implementation of the above problem in FEniCS 2019.1.0, modified from demo_nonlinear-poisson.py from Section 14 of [1].

```
from fenics import *

def q(u):
  '''
  Return nonlinear coefficient q(u) = 1 + u^2
  '''
  return 1 + u*u

mesh = UnitSquareMesh(8,8)
space = FunctionSpace(mesh,'P',1)

# Define boundary value u_d
u_d = Expression('1 + x[0] + 2*x[1]',degree=1)
f = Expression('-10*x[0] - 20*x[1] - 10',degree=1)

def boundary(x,on_boundary):
  return on_boundary

bc = DirichletBC(space,u_d,boundary)

u = Function(space)
v = TestFunction(space)

F = q(u)*dot(grad(u),grad(v))*dx - f*v*dx

solve(F==0,u,bc)

# plot(u)
File("nonlin_poisson.pvd") << (u,0.0)
```

The new consideration in this problem is that we must use a nonlinear solver. It is important to note that for the nonlinear problem, the unknown function `u` used to construct the variational form must be defined as a `Function`, not as a `TrialFunction` as in the linear case:

```
u = Function(space)
v = TestFunction(space)
F = q(u)*dot(grad(u),grad(v))*dx - f*v*dx
```

The nonlinear solver is then set up simply by defining the formula for F (2.24) and using the statement

```
solve(F == 0, u, bc)
```

instead of the usual `solve(a == L, u, bc)` used in the linear case. The `solve` function takes the nonlinear equations and symbolically derives the Jacobian matrix. It then applies Newton's method to compute the solution. When the code is run, FEniCS reports the progress of each Newton iteration, and we find that the problem converges in eight iterations with a tolerance of 10^{-9}.

3 Shallow shelf model

In this section we examine a FEniCS 2019.1.0 implementation of the shallow shelf model for glaciological flow given by the Shelfy Stream Model presented in [10]. The implementation follows [25]. The shallow shelf model for glaciological flow is a model for a marine ice sheet that consists of a grounded portion sliding over its bed and an attached ice shelf. Assuming that slip at the base of the ice is fast compared to shearing across the ice thickness, [24] derived the model for horizontal components of ice velocity:

$$\partial_x\big(4h\nu u_x + 2h\nu v_y\big) + \partial_y\big(h\nu(u_y + v_x)\big) - \tau_b^x = \rho g h s_x, \tag{3.1}$$

$$\partial_y\big(4h\nu v_y + 2h\nu u_x\big) + \partial_x\big(h\nu(u_y + v_x)\big) - \tau_b^y = \rho g h s_y, \tag{3.2}$$

$$\nu = \left(\frac{B}{2}\right)^{-1/n} \left| u_x^2 + v_y^2 + u_x v_y + \frac{1}{4}\big(u_y + v_x\big)^2 \right|^{\frac{1-n}{2n}}, \tag{3.3}$$

where h is vertical thickness, s is surface elevation, ρ is density, u and v are horizontal components of ice velocity, $\boldsymbol{\tau}_b$ is basal stress determined by the sliding law $f(\mathbf{u})$, ν is the viscosity, and B and n are the coefficients in Glen's law [17]. We consider $n = 3$ for all models in this report. This model is defined on a periodic domain with a constant bed slope in the x direction and no bed slope in the y direction.

The code for the FEniCS implementation of the shallow shelf model is provided in Appendix A. The parameters used in this implementation are found in [8] and [9]. The domain is 40×40 km with 1 km resolution and a constant bed slope of $-0.5°$ in the x direction. The bed has a uniform thickness of $H = 1000$ m. The Glen's law parameter B is defined as $B = 2.1544 \cdot 10^5$ Pa(m a^{-1})$^{-1/3}$ and $n = 3$ as in Table 1 of [8]. A linear sliding law $f(\mathbf{u}) = \beta^2\mathbf{u}$ is used with sliding coefficient β^2 defined as

$$\beta^2 = 1000 - 750\exp\left(-\left(\frac{r}{5}\right)^2\right), \tag{3.4}$$

where r is the distance from the center of the domain in kilometers.

We first define the periodic boundary condition[3]. In order to maintain consistency in the units of measurement, we convert all distances to meters. Therefore, the periodic boundary condition is defined on a 40,000 m$\times$ 40,000 m domain with 40 grids along each axis so that the resolution is 1,000 m. Due to the shape of the domain, the mesh is created with a `RectangleMesh`:

```
L_x = 40*10**3 # (GH pg 11)
L_y = 40*10**3
nx = ny = 40
mesh = RectangleMesh(Point(0,0),Point(L_x,L_y),nx,ny,"crossed")
```

It is then necessary to defined a mixed finite element function space since the velocity vector $\mathbf{u}$ has components u and v. The periodic boundary condition is included as an argument in the `FunctionSpace` definition that specifies that all functions in `V` and `ME` have periodic boundary conditions defined by `periodic_bc`. `U` is defined as a `Function` in the mixed space `ME` which is then split into the components `u` and `v`:

```
U = Function(ME,name="u")

# split mixed functions
u, v = split(U)
```

The sliding coefficient β^2 is then defined. The equation for β^2 (3.4) is defined as an `Expresssion` that is interpolated into the finite element space `V`:

```
beta_2 = Function(V, name="F")
```

[3]Doubly periodic boundary condition code supplied by primary supervisor, and as used in [25]

```
fn = Expression("1000 - 750*exp(-1*(pow(x[0]-L_x/2, 2) + pow(x[1] - L_y/2, 2))
    /25e6)",L_x = float(L_x),L_y = float(L_y),element=beta_2.function_space().
    ufl_element())
beta_2.interpolate(fn)
```

The basal stress term $\boldsymbol{\tau}_b = \beta^2 \mathbf{u}$ given by (12) from [9] can then be defined:

```
tau = beta_2 * U
```

We may now begin to construct the variational form of (3.1)–(3.2). To obtain the weak form of (3.1), as usual, we multiply the equation by a test function ζ_u and integrate over the domain Ω. Rearranging so all terms are on the same side, we obtain the abstract formulation

$$L_0 = \int_\Omega \left(\partial_x \zeta_u \left(4h\nu u_x + 2h\nu v_y \right) + \partial_y \zeta_u \left(h\nu(u_y + v_x) \right) + \zeta_u \tau_b^x + \zeta_u \rho g h s_x \right) dx. \tag{3.5}$$

Similarly, the weak form of (3.2) is found to be

$$L_1 = \int_\Omega \left(\partial_y \zeta_v \left(4h\nu v_y + 2h\nu u_x \right) + \partial_x \zeta_v \left(h\nu(u_y + v_x) \right) + \zeta_v \tau_b^y + \zeta_v \rho g h s_y \right) dx, \tag{3.6}$$

where ζ_v is another test function.

To implement the viscosity ν (3.3), the partial derivatives of u and v must be calculated, which can easily be found with the use of the UFL differential operators described in Section 2.1.2:

```
u_x = u.dx(0)
u_y = u.dx(1)
v_x = v.dx(0)
v_y = v.dx(1)
```

After defining all necessary constants, we now have all the elements required to construct ν and the variational form of (3.1)–(3.2):

```
nu = ((B/2)) * abs(u_x**2 + v_y**2 + u_x*v_y + 0.25*( u_y +v_x )**2 + Constant(1
    e-12))**((1-n)/(2*n))
```

Here it is important to note the addition of a small ϵ term, $\epsilon = 10^{-12}$ m^2 a^{-2}, in order to avoid errors resulting from division by 0. We can now construct the variational problem to be solved:

```
L0 = grad(zeta_u)[0]*(4*h*nu*u_x + 2*h*nu*v_y)*dx + grad(zeta_u)[1]*(h*nu*(u_y +
     v_x))*dx + zeta_u*tau[0]*dx + zeta_u*rho*g*h*s_x*dx
L1 = grad(zeta_v)[1]*(4*h*nu*v_y + 2*h*nu*u_x)*dx + grad(zeta_v)[0]*(h*nu*(u_y +
     v_x))*dx + zeta_v*tau[1]*dx + zeta_v*rho*g*h*s_y*dx
L = L0+L1
```

This is a nonlinear problem, so a `NonlinearVariationalSolver` is used. The chosen nonlinear method is Newton's method, so the Jacobian of the problem must be provided. This is easily calculated using the `derivative` form operator described in Section 2.1.2:

```
Jac = derivative(L,U,dU)
```

We can then define the `NonlinearVariationalProblem`, which is used to define the `NonlinearVariationalSolver` to solve the problem:

```
problem = NonlinearVariationalProblem(L, U, J=Jac)
solver = NonlinearVariationalSolver(problem)
```

With the chosen solver parameters, the solver converges after 10 iterations. Figure 3 provides a visualization of the magnitude of the final velocity $\mathbf{u}$ over the domain.

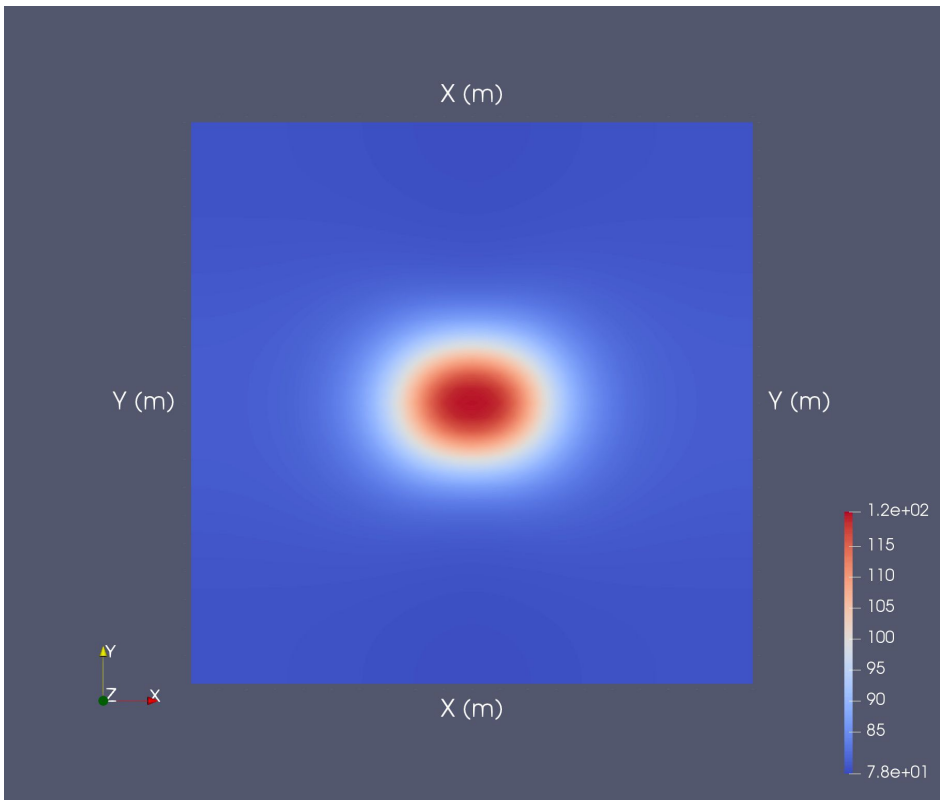

Figure 3: Final speed of ice sheet using the shallow shelf model.

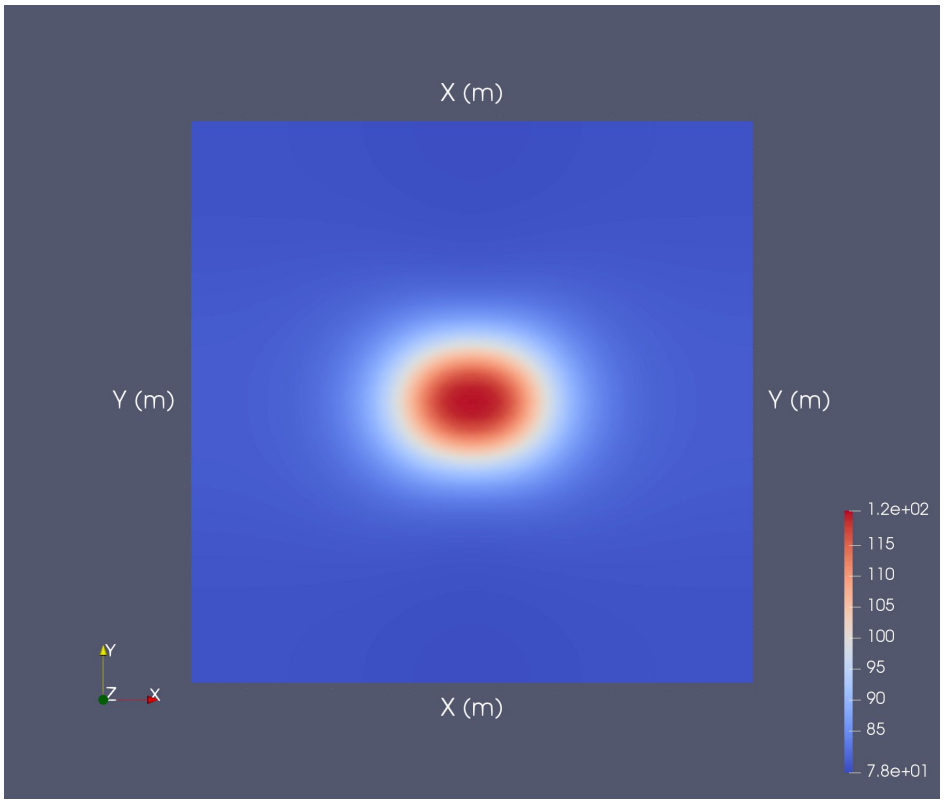

Figure 4: Final speed of ice sheet using the shallow shelf model with approximated Newton's method.

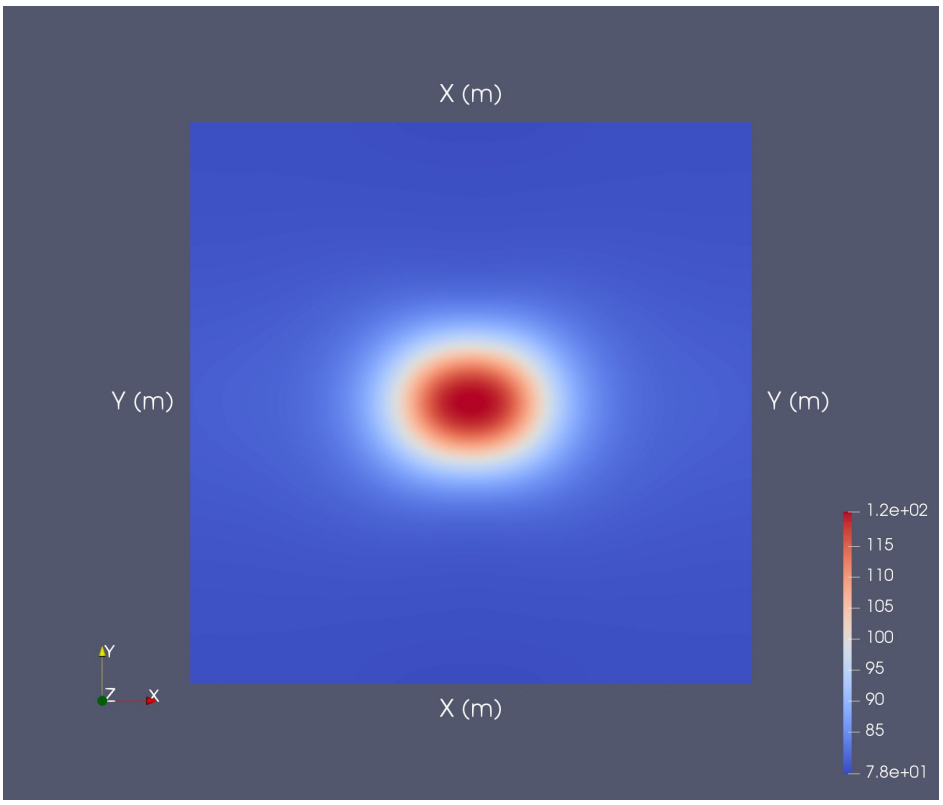

Figure 5: Shallow shelf model solved with combination of approximated Newton's method and Newton's method.

Let us now consider a more refined domain with a resolution of 0.5 km, i.e.,

```
nx = ny = 80
```

In this case we find that the Newton solver does not converge. Recall the requirements for the convergence of Newton's method [29]: Newton's method will converge in second order if

1. the first derivative of L is nonzero in some neighborhood of the root α, and
2. the second derivative of L is continuous in that neighborhood of α, and
3. the initial guess is sufficiently close to α.

Assuming the problem with convergence is due to the initial guess not being sufficiently close to the root for the more refined domain, we address this issue in the next section.

3.1 Approximated Newton's Method

To address this problem of convergence, let us consider a methodology presented in [25] which is an *approximated* Newton's method. The main idea is that the u and v terms in the viscosity $\nu(u, v)$ are now treated as constants, so that the chain rule is not applied to ν when the Jacobian of the variational form is calculated. To do this, the variables w_1 and w_2 are introduced to replace u and v, respectively, in (3.3):

```
W = Function(ME)
w1,w2 = split(W)

w1_x = w1.dx(0)
w1_y = w1.dx(1)
w2_x = w2.dx(0)
w2_y = w2.dx(1)

nu = ((B/2)) * abs(w1_x**2 + w2_y**2 + w1_x*w2_y + 0.25*( w1_y +w2_x )**2 +
    Constant(1e-8))**((1-n)/(2*n))
```

The variational form `L` is constructed as before, except ν is now $\nu(w_1, w_2)$. The Jacobian is then calculated by differentiating `L` with respect to `U`:

```
Jac = derivative(L,U,dU)
```

Now all instances of w_1, w_2 must be replaced with u, v. This is because the substitution was solely to prevent differentiation of $\nu(u, v)$ when calculating the Jacobian. Now that the Jacobian has been calculated, the entire problem must be converted back back so everything is in terms of the original variables u and v. This is achieved using the UFL `replace` function:

```
L = ufl.replace(L,{W:U})
Jac = ufl.replace(Jac,{W:U})
```

Using the same solver parameters as before with a resolution of 0.5 km, the problem converges in 22 iterations. The speed over the domain is presented in Figure 4. Through a visual comparison it is evident that both methods result in the same solution. By comparing the magnitude of the final velocity in both cases, we find that this is indeed true. Both methods result in a calculated speed of $|\mathbf{u}| = 4637$ m/yr with components $|u| = 4636$ m/yr and $|v| = 49$ m/yr. We find that the problem with the original resolution of 1 km also converges in 22 iterations.

In this case, all iterations were calculated using the approximated Jacobian calculated where the chain rule was not applied to the viscosity (3.3). Although this method still uses a Newton solver, the use of an approximation to the Jacobian results in it being a first order method. To achieve faster convergence to the solution, let us consider combining the approximated Newton's method with Newton's method as proposed in [25]. The idea is to initially use the approximated Newton's method for a certain number of iterations until the iterated solution has progressed far enough away from the problematic area that results in unstable convergence of the problem. This solution is then used as the initial guess for Newton's method to solve the problem in order to have faster convergence than using a first order method for the entire problem. For this combined method, the Jacobian must be calculated twice: once for the approximated Newton's method where the Chain Rule is not applied to $\nu(u, v)$ and again for Newton's method where the Chain Rule is fully applied.

The code for using Newton's method in conjunction with the approximated Newton's method is provided in Appendix C. The initial setup is exactly the same as for using approximated Newton's method for the entire problem except for the maximum iterations allowed for the solver, since we want the solver to terminate before the problem is solved. Since the problem converged after 22 iterations with the approximated Newton's method, 10 iterations seemed reasonable to allow the problem to progress far enough away from the initial solution:

```
solver.parameters["newton_solver"]["maximum_iterations"] = 10
```

After the approximated Newton's method solver terminates, the Jacobian is calculated for the variational form that is in terms of u and v only and a new solver is initialized, with the solution `U` from the approximated Newton solver used as the initial guess:

```
L2 = ufl.replace(L,{W:U})
Jac = derivative(L2,U,dU)

problem = NonlinearVariationalProblem(L2, U,J=Jac)
solver2 = NonlinearVariationalSolver(problem)
```

`L` contained terms in `W` as well as `U`, whereas `L2` is in terms of `U` only.

We find that after the initial 10 iterations for the approximated Newton solver, the problem converges with 3 Newton iterations for domain resolutions of both 1 km and 0.5 km. The speed of the ice sheet is displayed in Figure 5 and matches the results found using the other solver methods.

The significant improvement here is that for a resolution of 0.5 km, Newton's method did not converge, the approximated Newton's method converged in 22 iterations, and combining the two methods resulted in convergence in 13 iterations. Newton's method had the fastest convergence for lower resolution domains but became unstable as the domain resolution was refined.

4 Hybrid model: L1L2

In this section we consider the L1L2 hybrid model discussed in [8] and present an implementation of the model using FEniCS 2019.1.0. This code is provided in Appendix D.

4.1 Problem formulation

The hybrid model in this section considers a variation to the first-order momentum balance equations in Cartesian coordinates (1.1)–(1.3) given by Equations (20)–(25) of [8]. This is a 3-dimensional model for ice flow. The goal of the numerical scheme is to solve a 2-dimensional PDE at each iterative step.

The classical form of the 2-D PDE system is given by

$$\partial_x\big[H\bar{\nu}_{(\mathrm{hy})}(4\bar{u}_x + 2\bar{v}_y)\big] + \partial_y\big[H\bar{\nu}_{(\mathrm{hy})}(\bar{v}_x + \bar{u}_y)\big] - \tau^x = \rho g H s_x, \tag{4.1}$$

$$\partial_x\big[H\bar{\nu}_{(\mathrm{hy})}(\bar{v}_x + \bar{u}_y)\big] + \partial_y\big[H\bar{\nu}_{(\mathrm{hy})}(4\bar{v}_y + 2\bar{u}_x)\big] - \tau^y = \rho g H s_y, \tag{4.2}$$

where a bar over a variable represents the 2-dimensional depth-average of the variable and H is the vertical thickness. The effective viscosity $\nu_{(\mathrm{hy})}$ given by

$$\nu_{(\mathrm{hy})} = \frac{B}{2}\left[\bar{u}_x^2 + \bar{v}_y^2 + \bar{u}_x\bar{v}^y + \frac{1}{4}(\bar{u}_y + \bar{v}_x)^2 + \frac{1}{4}u_z^2 + \frac{1}{4}v_z^2\right]^{\frac{1-n}{2n}}. \tag{4.3}$$

Expressions for u_z and v_z are given by

$$\nu_{(\mathrm{hy})}u_z = \frac{\tau^x}{H}(s - z), \tag{4.4}$$

$$\nu_{(\mathrm{hy})}v_z = \frac{\tau^y}{H}(s - z), \tag{4.5}$$

As in Section 3, a linear basal sliding law $f(u_b) = \beta^2 u_b$ is used, with the coefficient β^2 given by (3.4). Additionally, the surface is at a 0.5° angle with the horizontal. Equations for the basal stress $\boldsymbol{\tau}$ are

$$\tau^x = \frac{mf(u_b)}{u_b\left(1 + \frac{mf(u_b)\omega}{u_b H}\right)}\bar{u}, \tag{4.6}$$

$$\tau^y = \frac{mf(v_b)}{v_b\left(1 + \frac{mf(v_b)\omega}{v_b H}\right)}\bar{v}, \tag{4.7}$$

where $m = \sqrt{1 + b_x^2 + b_y^2}$ and

$$\omega \equiv \int_b^s \int_b^z \frac{(s - z')}{H\nu_{(\mathrm{hy})}} dz'\, dz. \tag{4.8}$$

Rearranging the limits of integration, we have

$$\int_b^s \int_b^z \frac{(s - z')}{H\nu_{(\mathrm{hy})}} dz'\, dz = \int_b^s \int_{z'}^s \frac{(s - z')}{H\nu_{(\mathrm{hy})}} dz\, dz',$$

and by integrating the right-hand side we find

$$\omega = \int_b^s \frac{(s - z)^2}{H\nu_{(\mathrm{hy})}} dz. \tag{4.9}$$

(4.1)–(4.9) are then used to form an iterative scheme with inputs $u^{(i)}$ and $v^{(i)}$, from which $\nu_{(\text{hy})}^{(i)}, \omega^{(i)}$, and $\beta_{\text{eff}}^{(i)}$ can be diagnosed, where

$$\beta_{\text{eff}}^{(i)} = \frac{mf(u_b^{(i)})}{u_b^{(i)}\left(1 + \frac{mf(u_b^{(i)})\omega^{(i)}}{u_b^{(i)}H}\right)}. \tag{4.10}$$

The 2-dimensional PDE system (4.1)–(4.2) is then solved for $\bar{u}^{(i+1)}$ and $\bar{v}^{(i+1)}$. Then $\tau^{x(i+1)} = \beta_{\text{eff}}^{(i)}\bar{u}^{(i+1)}$, and $u_z^{(i+1)}$ is calculated from (4.4) using $\nu_{(\text{hy})}^{(i)}$. $\tau^{y(i+1)}$ and $v_z^{(i+1)}$ are found in a similar manner. This finishes one iteration of the model. Iteration is continued until the difference between iterates is below a specified tolerance.

4.2 Implementation

We now examine an implementation of the iterative scheme described above using FEniCS 2019.1.0. With this 3-dimensional model, in addition to the parameters defined in Section 3, the number of horizontal levels into which the new vertical component of ice is split must be specified:

```
num_levels = 10 # number of horizontal layers
dz = float(h)/(num_levels-1)
```

As before, h is the vertical thickness of the ice sheet.

A new consideration is that several of the involved functions are elements of the P0 piecewise constant discontinuous finite element space. Since `u,v` are in the P1 finite element space, their derivatives are in the P0 finite element space. Additionally, any `Function` objects that interact with the partial derivative terms must also be in the P0 space, defined in the code as `V0`:

```
V0 = FunctionSpace(mesh, "DG", 0, constrained_domain = periodic_bc)
```

The construction of a P0 space was not required for the shallow shelf model implementation because the gradients of `u` and `v` were only used as part of the finite element assembly and were not used in calculations with any `Function` objects. Therefore, they did not need to be stored as P0 `Function` objects themselves.

In this implementation, `U_bar` is used to represent the depth-averaged solution. `z` represents the elevation $z(x, y)$. `nu_hy` is the effective viscosity, which varies in the vertical dimension. We construct the 3-D variables as a series of "stacked" horizontal 2D functions defined on each layer:

```
nu_hy = [Function(V0) for i in range(num_levels-1)]
u_z = [Function(V0) for i in range(num_levels-1)]
v_z = [Function(V0) for i in range(num_levels-1)]
```

For the initial guess of the effective viscosity $\nu_{(\text{hy})}^{(0)}$, we must add a small ϵ term to prevent division by 0 in further calculations:

```
for i in range(num_levels-1):
  nu_hy[i].vector()[:] += 1e-8
```

We also have an initial guess of the basal stress $\tau^{x(0)}$ and $\tau^{y(0)}$:

```
tau_x = Function(V0)
tau_y = Function(V0)
```

We can now construct the iteration loop. In each iteration, the partial derivatives of $u^{(i)}$ and $v^{(i)}$ must be calculated. Note that in this case we do not want a purely symbolic differentiation since we want to evaluate the value of the derivatives. Therefore, the symbolically differentiated objects must be projected onto the P0 element space:

```
u_bar, v_bar = split(U_bar)
u_x = project(u_bar.dx(0),V0)
u_y = project(u_bar.dx(1),V0)
```

```
v_x = project(v_bar.dx(0),V0)
v_y = project(v_bar.dx(1),V0)
```

$u_z^{(i+1)}$ and $v_z^{(i+1)}$ are then evaluated on each level by applying (4.4):

```
for i in range(num_levels-1):
    u_z[i].vector()[:] = tau_x.vector()[:]* ((i + 0.5)*dz)/(float(h)*nu_hy[i].
    vector()[:]) # G11 Eq. 31
    v_z[i].vector()[:] = tau_y.vector()[:]* ((i + 0.5)*dz)/(float(h)*nu_hy[i].
    vector()[:])
```

The next iterate of $\nu_{(\text{hy})}$ is calculated using (4.3), the depth-average of which is $\bar{\nu}_{(\text{hy})}$, implemented as `nu_hy_bar`:

```
  nu_hy_bar = Function(V0)
  # depth-average: add each layer, then divide by number of layers
  for i in range(num_levels-1):
    nu_hy_bar.vector()[:] += nu_hy[i].vector()[:]
  nu_hy_bar.vector()[:] /= (num_levels-1)
```

Then $\omega^{(i)}$ is numerically integrated by applying the Midpoint Rule to (4.9):

```
for i in range(num_levels - 1):
    omega.vector()[:] += (((i + 0.5)*dz)**2) / (float(h)*nu_hy[i].vector()[:])

omega.vector()[:] *= dz # multiply sum by (b-a)/n
```

$\beta_{\text{eff}}^{(i)}$ is implemented as `beta_eff` with the approximation $m = 1$. We then calculated $\tau^{x(i+1)}$ and $\tau^{y(i+1)}$:

```
tau_x = project(beta_eff * u_bar,V0)
tau_y = project(beta_eff * v_bar,V0)
```

As was done for the partial derivatives, `tau_x` must be projected onto the P0 element space so it can be used in future calculations. The variational problem is then constructed and solved for $\bar{u}^{(i+1)}$ and $\bar{v}^{(i+1)}$:

```
L0 = grad(zeta_u)[0]*(h*nu_hy_bar*(4*u_bar.dx(0) + 2*v_bar.dx(1)))*dx + grad(
    zeta_u)[1]*(h*nu_hy_bar*(u_bar.dx(1) + v_bar.dx(0)))*dx + zeta_u*beta_eff*
    u_bar*dx + zeta_u*rho*g*h*s_x*dx
L1 = grad(zeta_v)[1]*(h*nu_hy_bar*(4*v_bar.dx(1) + 2*u_bar.dx(0)))*dx + grad(
    zeta_v)[0]*(h*nu_hy_bar*(u_bar.dx(1) + v_bar.dx(0)))*dx + zeta_v*beta_eff*
    v_bar*dx + zeta_v*rho*g*h*s_y*dx
L = L0+L1
```

In this model, `L` is linear in `U_bar` so a linear solver is used to solve the problem. To this end, `L` can be converted to a combination of a bilinear form, called `lhs` in this case, and linear form `rhs`, where `lhs` is a function of the test and trial functions. The trial function is introduced into `L` with the following:

```
ufl.replace(L, {U_bar: dU})
```

We can then extract the bilinear and linear forms:

```
lhs, rhs = system(ufl.replace(L, {U_bar: dU}))
```

Now the `LinearVariationalSolver` can be constructed and solver parameters may be specified. Since the periodic boundary condition is already taken into consideration for all the `FunctionSpace` elements, the final boundary condition argument is not needed for the construction of the `LinearVariationalProblem`.

```
problem = LinearVariationalProblem(lhs,rhs, U_bar, [])
solver = LinearVariationalSolver(problem)
solver.solve()
```

With a relative tolerance of 10^{-8}, this problem converges in 52 iterations. The final speed is visualized of the ice sheet in Figure 6. This is due to the form of β^2 (3.4) which places a "slippery spot" in the center of the domain. In both the shallow shelf and hybrid models we see that ice

moves faster in the center of the domain. However, the maximum speed of ice flow is noticeably different for the two models: for the shallow shelf model, the ice in the center of the domain moves at 120 m/yr, whereas for the hybrid model the ice in the center of the domain moves at 150 m/yr. Compared to the shallow shelf model, there is a 25% increase in speed for the hybrid model. Note here that the quantities calculated for the shallow shelf model (Figures 3–5) and the hybrid model are slightly different, as Figure 6 represents the *depth-averaged* velocity of the ice sheet. For a more accurate comparison we must calculate the *surface* velocity, which presents some scope for further study, as discussed in Section 5

The variational form of the shallow shelf model was relatively simple, which allowed us to simply use FEniCS's available `derivative` operator to calculate the Jacobian of the variational form. Thus the problem was solved using FEniCS's Newton solver. The formulation for the hybrid model, on the other hand, is much more complicated. The variational form does not allow for automatic calculation of the Jacobian. Therefore, for this implementation it was necessary to build a fixed-point iteration ourselves and solve a linear variational problem in each iteration. DOLFIN is equipped to handle this efficiently. The first time the form was encountered in the execution of the program, the necessary code was automatically generated, compiled, and cached. The JIT compiler then recognized the form on subsequent iterations and reused the generated code.

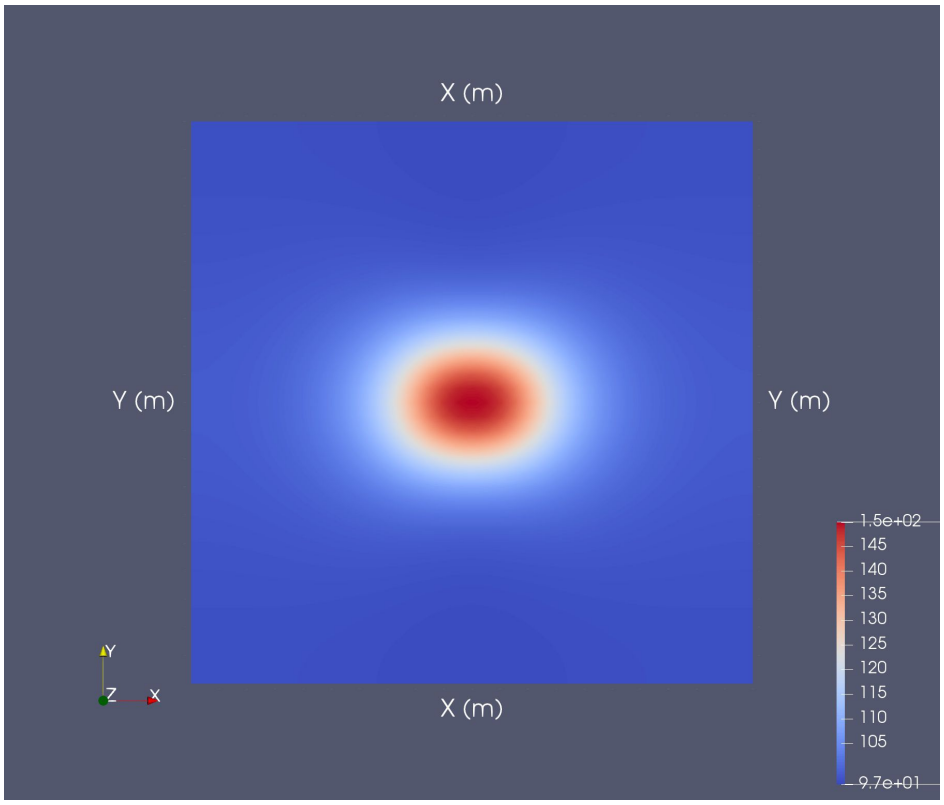

Figure 6: Depth-averaged speed of ice sheet calculated from L1L2 model.

5 Conclusions

We have explored the functionality of FEniCS by solving several classic PDE problems, as well as studying several simplified glaciological models. A core component of the FEniCS Project is the DOLFIN library, which largely automates the finite element approximation of solutions to PDEs. DOLFIN uses a compiler for automatic generation of problem-specific code. This allows for efficient assembly of a wide variety of variational forms from a previously unavailable array of finite elements. Expression of finite element discretizations of variational forms is handled by the UFL library. UFL enables users to construct variational forms in a intuitive language that resembles mathematical notation. In fact, it is not even necessary for the user to explicitly formulate the linear and bilinear forms of the problem. As demonstrated when solving the heat problem in Section 2.2.2, it suffices to provide FEniCS with the abstract formulation of the variational problem. FEniCS can then determine the linear and bilinear forms needed to solve the problem itself. This feature was especially important in Chapter 4 as it allowed us to automatically extract the linear and bilinear forms of a complicated variational problem.

In the latter part of Chapter 2 we studied several demo problems in preparation for the more complicated PDEs we would study later in the report. Let us consider some crucial insights from these demo problems: how to use FEniCS to solve both linear and nonlinear variational problems; how to construct symbolic expressions and function objects, and how to project or interpolate them into a discretized finite element space; and how to symbolically differentiate forms, which is necessary for applying Newton's method and other nonlinear solvers.

In Chapter 3 we used FEniCS to solve a shallow shelf model for glaciological flow. For this simplified model, FEniCS's powerful functionality allowed for symbolically differentiation of the variational form of the problem. Thus it was sufficient to use the available Newton solver to find the solution of the problem. However, we also found that Newton's method was not a stable solver for more refined meshes. This led to the introduction of the approximated Newton's method presented in [25]. This method resulted in convergence in the first order for more refined grid spacings. To improve the order of convergence we then used the approximated Newton's method in conjunction with Newton's method to achieve faster convergence on higher resolution domains than solely using the first-order method.

In Chapter 4 we examined a 3-dimensional model for glaciological flow. By integrating over the vertical dimension this model was reduced to a 2-dimensional PDE system that was solved at each iterative step. This model had a variational form that was too complicated to automatically symbolically differentiate. Therefore, it could not be solved using FEniCS's available nonlinear variational solvers. As a result, a fixed-point iteration of velocity was implemented with a linear variational solver to solve a new variational problem in each iteration. Here, DOLFIN's form compiler proved crucial in improving the efficiency of the program, as it generated code only the first time the variational form was encountered in the execution of the program. This code was compiled and cached, and the JIT compiler recognized the form in subsequent iterations and so the generated code could be reused.

With both models we found that ice moves faster in the center of the domain where the "slippery spot" is located. There is an approximate 25% increase in the speed of the ice in the center of the domain using the hybrid model, in which ice moves at 150 m/yr, in comparison to the shallow shelf model, in which ice moves at 120 m/yr in the center.

Future work As noted previously, in this report the final quantity calculated from the hybrid model is the depth-averaged velocity of the ice sheet. An obvious next step would be to calculate the surface velocity of the ice sheet, as this is actually an observable quantity and so solutions could be compared to physical observations. A possible model to solve for surface velocity is presented in [13]. Additionally, in all our experiments the vertical thickness of the ice was treated as constant. For a more physically realistic model, we may consider models in which the

thickness of ice varies over the horizontal domain, or with time as examined in [9].

References

[1] Fenics project: Collection of documented demos. https://fenicsproject.org/olddocs/dolfin/1.3.0/python/demo/index.html. [Online; accessed 22-August-2019].

[2] D. Arnold and A. Logg. Periodic table of the finite elements. *Siam News*, 47, 11 2014.

[3] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. Petsc web page. http://www.mcs.anl.gov/petsc/. [Online; accessed 22-August-2019].

[4] H. Blatter. Velocity and stress fields in grounded glaciers: a simple algorithm for including deviatoric stress gradients. *Journal of Glaciology*, 41(138):333–344, 1995.

[5] E. Bueler and J. Brown. Shallow shelf approximation as a "sliding law" in a thermomechanically coupled ice sheet model. *Journal of Geophysical Research*, 114, 2009.

[6] FEniCS. Fenics project. https://www.fenicsproject.org, 2019. [Online; accessed 22-August-2019].

[7] C. Geuzaine and J.-F. Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79:1309 – 1331, 09 2009.

[8] D. Goldberg. A variationally derived, depth-integrated approximation to a higher-order glaciological flow model. 2011.

[9] D. Goldberg and P. Heimbach. Parameter and state estimation with a time-dependent adjoint marine ice sheet model. 2013.

[10] D. Goldberg, D. M. Holland, and C. Schoof. Grounding line movement and ice shelf buttressing in marine ice sheets. *Journal of Geophysical Research: Earth Surface*, 114(F4), 2009.

[11] P. Gottschling and A. Lumsdaine. The matrix template library 4. http://www.osl.iu.edu/research/mtl/mtl4/. [Online; accessed 22-August-2019].

[12] R. Greve and H. Blatter. *Large-Scale Dynamics of Ice Sheets*, pages 61–109. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[13] R. Hindmarsh. A numerical comparison of approximations to the stokes equations used in ice sheet and glacier modeling. *Journal of Geophysical Research*, 109, 03 2004.

[14] R. Kirby and A. Logg. A compiler for variational forms. *ACM Transactions on Mathematical Software*, 32:417–444, 9 2006.

[15] R. C. Kirby, M. Knepley, A. Logg, and L. R. Scott. Optimizing the evaluation of finite element matrices. *SIAM Journal on Scientific Computing*, 27:741–758, 2005.

[16] R. C. Kirby and L. R. Scott. Geometric optimization of the evaluation of finite element matrices. *SIAM J. Scientific Computing*, 29:827–841, 2007.

[17] K. Kuffey and W. S. B. Paterson. *The Physics of Glaciers*, chapter 3. Academic Press, 4 edition, 2010.

[18] H. P. Langtangen and A. Logg. The fenics tutorial volume 1. https://fenicsproject.org/pub/tutorial/sphinx1/. [Online; accessed 22-August-2019].

[19] A. Logg and H. P. Langtangen. *Solving PDEs in Python – The FEniCS Tutorial Volume I*. Springer, 2017.

[20] A. Logg, K. Mardal, and G. Wells. *Automated Solution of Differential Equations by the Finite Element Method*, chapter 17. Springer, 2012.

[21] A. Logg and G. N. Wells. Dolfin: Automated finite element computing. 03 2011.

[22] K. Long et al. Sundance. http://www.math.ttu.edu/ klong/Sundance/html/.

[23] D. MacAyeal and R. Thomas. The effects of basal melting on the present flow of the ross ice shelf, antarctica. *Journal of Glaciology*, 32(110):72–86, 1986.

[24] D. R. MacAyeal. Large-scale ice flow over a viscous basal sediment: Theory and application to ice stream b, antarctica. *Journal of Geophysical Research: Solid Earth*, 94(B4):4071–4087, 1989.

[25] J. R. Maddison, D. N. Goldberg, and B. D. Goddard. Automated calculation of higher order partial differential equation constrained derivative information. *SIAM Journal on Scientific Computing*, 2013.

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

[27] F. Pattyn. A new three-dimensional higher-order thermomechanical ice sheet model: Basic sensitivity, ice stream development, and ice flow across subglacial lakes. *Journal of Geophysical Research: Solid Earth*, 108(B8), 2003.

[28] O. Pironneau, F. Hecht, and A. Le Hyaric. Freefem++. http://www.freefem.org/.

[29] V. S. Ryanben'kii and S. V. Tyrnkov. *A Theoretical Introduction to Numerical Analysis*. 2006.

[30] L. R. Scott. *Introduction to Automated Modeling using FEniCS*, chapter 1. 2017.

[31] J. Walter, M. Koch, et al. ublas. http://www.boost.org/. [Online; accessed 22-August-2019].

Appendices

A Shallow shelf implementation with Newton solver

```
from dolfin import *
import ufl

class PeriodicBoundaryCondition(SubDomain):
  # Following FEniCS 2017.1.0 API
  def inside(self, x, on_boundary):
    return (abs(x[0]) < 1.0e-8 or abs(x[1]) < 1.0e-8) and abs(x[0] - L_x) > 1.0e-8 and abs(x[1] - L_y) > 1.0e-8
  def map(self, x, y):
    if abs(x[0] - L_x) < 1.0e-8:
      if abs(x[1] - L_y) < 1.0e-8:
        y[0] = 0.0
        y[1] = 0.0
      else:
        y[0] = 0.0
        y[1] = x[1]
    elif abs(x[1] - L_y) < 1.0e-8:
      y[0] = x[0]
      y[1] = 0.0
    else:
      y[:] = -1.0e10
L_x = 40*10**3 # (GH pg 11)
L_y = 40*10**3
periodic_bc = PeriodicBoundaryCondition()

# Create mesh and define function spaces
nx = ny = 40
mesh = RectangleMesh(Point(0,0),Point(L_x,L_y),nx,ny,"crossed")
V = FunctionSpace(mesh, "Lagrange", 1, constrained_domain = periodic_bc)
ME = FunctionSpace(mesh, V.ufl_element()*V.ufl_element(),constrained_domain = periodic_bc)

# Define trial and test functions
dU    = TrialFunction(ME)
zeta_u, zeta_v  = TestFunctions(ME)

# Define functions
U = Function(ME,name="u")  # in mixed function space, is vector containing u and v

# split mixed functions
u, v = split(U)
u_x = u.dx(0)
u_y = u.dx(1)
v_x = v.dx(0)
v_y = v.dx(1)

beta_2 = Function(V, name="F")
fn = Expression("1000 - 750*exp(-1*(pow(x[0]-L_x/2, 2) + pow(x[1] - L_y/2, 2))/25e6)",L_x = float(L_x),L_y = float(L_y),element=beta_2.function_space().ufl_element())
beta_2.interpolate(fn)

B = Constant(2.1544e5) #G11 Table 1
n = Constant(3.0,name='n') # n = 3 used in all computations
h = Constant(1000,name='h') # GH13 pg. 11
s_x = Constant(tan(-0.5 * pi/180)) # slope of -0.5 degrees
s_y = Constant(0.0)
rho = Constant(910,name='rho') # G11 Table 1
g = Constant(9.81) # G11 Table 1

```

```
tau = beta_2 * U

nu = ((B/2)) * abs(u_x**2 + v_y**2 + u_x*v_y + 0.25*( u_y +v_x )**2 + Constant(1
    e-8))**((1-n)/(2*n)) # add small eps to prevent division by 0; G11 Eq 16

L0 = grad(zeta_u)[0]*(4*h*nu*u_x + 2*h*nu*v_y)*dx + grad(zeta_u)[1]*(h*nu*(u_y +
     v_x))*dx + zeta_u*tau[0]*dx + zeta_u*rho*g*h*s_x*dx
L1 = grad(zeta_v)[1]*(4*h*nu*v_y + 2*h*nu*u_x)*dx + grad(zeta_v)[0]*(h*nu*(u_y +
     v_x))*dx + zeta_v*tau[1]*dx + zeta_v*rho*g*h*s_y*dx
L = L0+L1

Jac = derivative(L,U,dU) # Jacobian

problem = NonlinearVariationalProblem(L, U, J=Jac)
solver = NonlinearVariationalSolver(problem)
solver.parameters["nonlinear_solver"] = "newton"
solver.parameters["newton_solver"]["linear_solver"] = "umfpack"
solver.parameters["newton_solver"]["convergence_criterion"] = "incremental"
solver.parameters["newton_solver"]["relative_tolerance"] = 1e-6
solver.parameters["newton_solver"]["absolute_tolerance"] = 1e-16
solver.parameters["newton_solver"]["maximum_iterations"] = 3000
solver.parameters["newton_solver"]["lu_solver"]["symmetric"] = True

solver.solve()

# Output file
file = File("ss_old.pvd", "compressed")
file << (U,0.0)
```

B Approximated Newton's method implementation

```
from dolfin import *
import ufl

class PeriodicBoundaryCondition(SubDomain):
  # Following FEniCS 2017.1.0 API
  def inside(self, x, on_boundary):
    return (abs(x[0]) < 1.0e-8 or abs(x[1]) < 1.0e-8) and abs(x[0] - L_x) > 1.0e
    -8 and abs(x[1] - L_y) > 1.0e-8
  def map(self, x, y):
    if abs(x[0] - L_x) < 1.0e-8:
      if abs(x[1] - L_y) < 1.0e-8:
        y[0] = 0.0
        y[1] = 0.0
      else:
        y[0] = 0.0
        y[1] = x[1]
    elif abs(x[1] - L_y) < 1.0e-8:
      y[0] = x[0]
      y[1] = 0.0
    else:
      y[:] = -1.0e10
L_x = 40*10**3 # (GH pg 11)
L_y = 40*10**3
periodic_bc = PeriodicBoundaryCondition()

# Create mesh and define function spaces
nx = ny = 40
mesh = RectangleMesh(Point(0,0),Point(L_x,L_y),nx,ny,"crossed")
V = FunctionSpace(mesh, "Lagrange", 1, constrained_domain = periodic_bc)
ME = FunctionSpace(mesh, V.ufl_element()*V.ufl_element(),constrained_domain =
    periodic_bc)

# Define trial and test functions
dU    = TrialFunction(ME)
zeta_u, zeta_v  = TestFunctions(ME)
```

```
34
35 # Define functions
36 U = Function(ME,name="u")
37
38 # split mixed functions
39 du, dv = split(dU)
40 u, v = split(U)
41
42 beta_2 = Function(V)
43 fn = Expression("1000 - 750*exp(-1*(pow(x[0]-L_x/2, 2) + pow(x[1] - L_y/2, 2))
       /25e6)",L_x = float(L_x),L_y = float(L_y),element=beta_2.function_space().
       ufl_element())
44 beta_2.interpolate(fn)
45
46 B = Constant(2.1544e5) #G11 Table 1
47 n = Constant(3.0,name='n') # n = 3 used in all computations
48 h = Constant(1000,name='h') # GH13 pg. 11
49 s_x = Constant(tan(-0.5 * pi/180)) # slope of -0.5 degrees
50 s_y = Constant(0.0)
51 rho = Constant(910,name='rho') # G11 Table 1
52 g = Constant(9.81) # G11 Table 1
53
54 tau = beta_2 * U
55
56 W = Function(ME)
57 w1,w2 = split(W)
58
59 u_x = u.dx(0)
60 u_y = u.dx(1)
61 v_x = v.dx(0)
62 v_y = v.dx(1)
63
64 w1_x = w1.dx(0)
65 w1_y = w1.dx(1)
66 w2_x = w2.dx(0)
67 w2_y = w2.dx(1)
68
69
70 nu = ((B/2)) * abs(w1_x**2 + w2_y**2 + w1_x*w2_y + 0.25*( w1_y +w2_x )**2 +
       Constant(1e-12))**((1-n)/(2*n)) # add small eps to prevent division by 0
71
72 L0 = grad(zeta_u)[0]*(4*h*nu*u_x + 2*h*nu*v_y)*dx + grad(zeta_u)[1]*(h*nu*(u_y +
        v_x))*dx + zeta_u*tau[0]*dx + zeta_u*rho*g*h*s_x*dx
73 L1 = grad(zeta_v)[1]*(4*h*nu*v_y + 2*h*nu*u_x)*dx + grad(zeta_v)[0]*(h*nu*(u_y +
        v_x))*dx + zeta_v*tau[1]*dx + zeta_v*rho*g*h*s_y*dx
74 L = L0+L1
75
76 Jac = derivative(L,U,dU) # Jacobian
77 L = ufl.replace(L,{W:U})
78 Jac = ufl.replace(Jac,{W:U})
79
80 problem = NonlinearVariationalProblem(L, U, J=Jac)
81 solver = NonlinearVariationalSolver(problem)
82 solver.parameters["nonlinear_solver"] = "newton"
83 solver.parameters["newton_solver"]["linear_solver"] = "umfpack"
84 solver.parameters["newton_solver"]["convergence_criterion"] = "incremental"
85 solver.parameters["newton_solver"]["relative_tolerance"] = 1e-6
86 solver.parameters["newton_solver"]["absolute_tolerance"] = 1e-16
87 solver.parameters["newton_solver"]["maximum_iterations"] = 3000
88 solver.parameters["newton_solver"]["lu_solver"]["symmetric"] = True
89
90 solver.solve()
91
92 # Output file
93 file = File("shallowshelf.pvd", "compressed")
94 file << (U,0.0)
```

C Combining Approximated Newton's Method with Newton's Method

```
from dolfin import *
import ufl

class PeriodicBoundaryCondition(SubDomain):
  # Following FEniCS 2017.1.0 API
  def inside(self, x, on_boundary):
    return (abs(x[0]) < 1.0e-8 or abs(x[1]) < 1.0e-8) and abs(x[0] - L_x) > 1.0e
    -8 and abs(x[1] - L_y) > 1.0e-8
  def map(self, x, y):
    if abs(x[0] - L_x) < 1.0e-8:
      if abs(x[1] - L_y) < 1.0e-8:
        y[0] = 0.0
        y[1] = 0.0
      else:
        y[0] = 0.0
        y[1] = x[1]
    elif abs(x[1] - L_y) < 1.0e-8:
      y[0] = x[0]
      y[1] = 0.0
    else:
      y[:] = -1.0e10
L_x = 40*10**3 # (GH pg 11)
L_y = 40*10**3
periodic_bc = PeriodicBoundaryCondition()

# Create mesh and define function spaces
nx = ny = 40
mesh = RectangleMesh(Point(0,0),Point(L_x,L_y),nx,ny,"crossed")
V = FunctionSpace(mesh, "Lagrange", 1, constrained_domain = periodic_bc)
ME = FunctionSpace(mesh, V.ufl_element()*V.ufl_element())

# Define trial and test functions
dU    = TrialFunction(ME)
zeta_u, zeta_v  = TestFunctions(ME)

# Define functions
U = Function(ME,name="u")

# split mixed functions
u, v = split(U)
u_x = u.dx(0)
u_y = u.dx(1)
v_x = v.dx(0)
v_y = v.dx(1)

beta_2 = Function(V, name="F")
fn = Expression("1000 - 750*exp(-1*(pow(x[0]-L_x/2, 2) + pow(x[1] - L_y/2, 2))
    /25e6)",L_x = float(L_x),L_y = float(L_y),element=beta_2.function_space().
    ufl_element())
beta_2.interpolate(fn)

B = Constant(2.1544e5) #G11 Table 1
n = Constant(3.0,name='n') # n = 3 used in all computations
h = Constant(1000,name='h') # GH13
s_x = Constant(tan(-0.5 * pi/180)) # slope of -0.5 degrees
s_y = Constant(0.0)
rho = Constant(910,name='rho') # G11 Table 1
g = Constant(9.81) # G11 Table 1

tau = beta_2 * U

W = Function(ME)
```

```
w1,w2 = split(W)
w1_x = w1.dx(0)
w1_y = w1.dx(1)
w2_x = w2.dx(0)
w2_y = w2.dx(1)


nu = ((B/2)) * abs(w1_x**2 + w2_y**2 + w1_x*w2_y + 0.25*( w1_y +w2_x )**2 +
    Constant(1e-8))**((1-n)/(2*n)) # add small eps to prevent division by 0; G11
     Eq 16

L0 = grad(zeta_u)[0]*(4*h*nu*u_x + 2*h*nu*v_y)*dx + grad(zeta_u)[1]*(h*nu*(u_y +
     v_x))*dx + zeta_u*tau[0]*dx + zeta_u*rho*g*h*s_x*dx
L1 = grad(zeta_v)[1]*(4*h*nu*v_y + 2*h*nu*u_x)*dx + grad(zeta_v)[0]*(h*nu*(u_y +
     v_x))*dx + zeta_v*tau[1]*dx + zeta_v*rho*g*h*s_y*dx
L = L0+L1

Jac = derivative(L,U,dU) # Jacobian
L2 = ufl.replace(L,{W:U})
Jac2 = ufl.replace(Jac,{W:U})

problem = NonlinearVariationalProblem(L2, U, J=Jac2)
solver = NonlinearVariationalSolver(problem)
solver.parameters["nonlinear_solver"] = "newton"
solver.parameters["newton_solver"]["linear_solver"] = "umfpack"
solver.parameters["newton_solver"]["convergence_criterion"] = "incremental"
solver.parameters["newton_solver"]["relative_tolerance"] = 1e-6
solver.parameters["newton_solver"]["absolute_tolerance"] = 1e-16
solver.parameters["newton_solver"]["maximum_iterations"] = 10
solver.parameters["newton_solver"]["lu_solver"]["symmetric"] = True

try:
  solver.solve()
except RuntimeError:
  pass


Jac = derivative(L2,U,dU)

problem = NonlinearVariationalProblem(L2, U,J=Jac)
solver2 = NonlinearVariationalSolver(problem)
solver2.parameters["nonlinear_solver"] = "newton"
solver2.parameters["newton_solver"]["linear_solver"] = "umfpack"
solver2.parameters["newton_solver"]["convergence_criterion"] = "incremental"
solver2.parameters["newton_solver"]["relative_tolerance"] = 1e-6
solver2.parameters["newton_solver"]["absolute_tolerance"] = 1e-16
solver2.parameters["newton_solver"]["maximum_iterations"] = 3000
solver2.parameters["newton_solver"]["lu_solver"]["symmetric"] = True

solver2.solve()

# Output file
file = File("dual_shallowshelf.pvd", "compressed")
file << (U,0.0)
```

D L1L2 Implementation

```
from dolfin import *
import ufl

class PeriodicBoundaryCondition(SubDomain):
  # Following FEniCS 2017.1.0 API
  def inside(self, x, on_boundary):
    return (abs(x[0]) < 1.0e-8 or abs(x[1]) < 1.0e-8) and abs(x[0] - L_x) > 1.0e
    -8 and abs(x[1] - L_y) > 1.0e-8
  def map(self, x, y):
```

```
    if abs(x[0] - L_x) < 1.0e-8:
      if abs(x[1] - L_y) < 1.0e-8:
        y[0] = 0.0
        y[1] = 0.0
      else:
        y[0] = 0.0
        y[1] = x[1]
    elif abs(x[1] - L_y) < 1.0e-8:
      y[0] = x[0]
      y[1] = 0.0
    else:
      y[:] = -1.0e10

L_x = 40*10**3 # (GH pg 11)
L_y = 40*10**3
periodic_bc = PeriodicBoundaryCondition()

# Create mesh and define function spaces
num_levels = 10 # number of horizontal layers
nx = ny = 40
mesh = RectangleMesh(Point(0,0),Point(L_x,L_y),nx,ny,"crossed")
V = FunctionSpace(mesh, "Lagrange", 1, constrained_domain = periodic_bc)
V0 = FunctionSpace(mesh, "DG", 0, constrained_domain = periodic_bc)
ME = FunctionSpace(mesh, V.ufl_element()*V.ufl_element(),constrained_domain =
    periodic_bc)

# Define trial and test functions
dU = TrialFunction(ME)
zeta_u, zeta_v  = TestFunctions(ME)

# Define functions
U_bar = Function(ME,name="u_bar")

beta_2 = Function(V, name="F")
fn = Expression("1000 - 750*exp(-1*(pow(x[0]-L_x/2, 2) + pow(x[1] - L_y/2, 2))
    /25e6)",L_x = float(L_x),L_y = float(L_y),element=beta_2.function_space().
    ufl_element())
beta_2.interpolate(fn)

B = Constant(2.1544e5) #G11 Table 1
n = Constant(3.0,name='n') # n = 3 used in all computations
h = Constant(1000,name='h') # GH13
rho = Constant(910,name='rho') # G11 Table 1
g = Constant(9.81) # G11 Table 1
s_x = Constant(tan(-0.5 * pi/180)) # slope of -0.5 degrees
s_y = Constant(0.0)

z = Function(V, name="z") # elevation

tau_x = Function(V0)
tau_y = Function(V0)

nu_hy = [Function(V0) for i in range(num_levels-1)]

# add small eps to prevent division by 0
for i in range(num_levels-1):
  nu_hy[i].vector()[:] += 1e-8

u_z = [Function(V0) for i in range(num_levels-1)]
v_z = [Function(V0) for i in range(num_levels-1)]

dz = float(h)/(num_levels-1)

temp_nu_hy_bar = Function(V0)

U_old = U_bar.copy(deepcopy=True)
```

```
72
73 num_it = 0
74 tol = 1e-8
75 while num_it < 3000:
76   omega = Function(V0, name="omega") # G11 Eq. 35
77   u_bar, v_bar = split(U_bar)
78   u_x = project(u_bar.dx(0),V0)
79   u_y = project(u_bar.dx(1),V0)
80   v_x = project(v_bar.dx(0),V0)
81   v_y = project(v_bar.dx(1),V0)
82
83   # (s-z) = (i + 0.5)*dz
84   for i in range(num_levels-1):
85     u_z[i].vector()[:] = tau_x.vector()[:]* ((i + 0.5)*dz)/(float(h)*nu_hy[i].
     vector()[:]) # G11 Eq. 31
86     v_z[i].vector()[:] = tau_y.vector()[:]* ((i + 0.5)*dz)/(float(h)*nu_hy[i].
     vector()[:])
87
88   temp_nu_hy_bar.vector()[:] = u_x.vector()[:]**2 + v_y.vector()[:]**2 + u_x.
     vector()[:]*v_y.vector()[:] + 0.25*(u_y.vector()[:] + v_x.vector()[:])**2 +
     1e-12# G11 2d terms of Eq. 16
89
90   for i in range(num_levels-1):
91     nu_hy[i].vector()[:] =  (float(B)/2) * (temp_nu_hy_bar.vector()[:] + 0.25*(
     u_z[i].vector()[:]**2 + v_z[i].vector()[:]**2 ))**((1-float(n))/(2*float(n))
     ) # G11 Eq. 16
92
93   nu_hy_bar = Function(V0)
94   # depth-average: add each layer, then divide by number of layers
95   for i in range(num_levels-1):
96     nu_hy_bar.vector()[:] += nu_hy[i].vector()[:]
97
98   nu_hy_bar.vector()[:] /= (num_levels-1)
99
100  # integrand: (s-z)^2 / h* nu_hy(z) dz
101  for i in range(num_levels - 1):
102    #omega.vector()[:] += (float(h) - (i + 0.5)*dz)**2 / (float(h)*nu_hy[
     num_levels - 2 - i].vector()[:])
103    omega.vector()[:] += (((i + 0.5)*dz)**2) / (float(h)*nu_hy[i].vector()[:])
104
105  omega.vector()[:] *= dz # multiply sum by (b-a)/n
106
107  m = 1
108  frac = (m*beta_2*omega)/h
109  beta_eff = m*beta_2/(1 + frac) # G11 Eq. 41 with cancellation due to f(u) =
     beta^2 * u (should be strictly nonnegative)
110
111  tau_x = project(beta_eff * u_bar,V0)
112  tau_y = project(beta_eff * v_bar,V0)
113
114  L0 = grad(zeta_u)[0]*(h*nu_hy_bar*(4*u_bar.dx(0) + 2*v_bar.dx(1)))*dx + grad(
     zeta_u)[1]*(h*nu_hy_bar*(u_bar.dx(1) + v_bar.dx(0)))*dx + zeta_u*beta_eff*
     u_bar*dx + zeta_u*rho*g*h*s_x*dx
115  L1 = grad(zeta_v)[1]*(h*nu_hy_bar*(4*v_bar.dx(1) + 2*u_bar.dx(0)))*dx + grad(
     zeta_v)[0]*(h*nu_hy_bar*(u_bar.dx(1) + v_bar.dx(0)))*dx + zeta_v*beta_eff*
     v_bar*dx + zeta_v*rho*g*h*s_y*dx
116  L = L0+L1
117
118  lhs, rhs = system(ufl.replace(L, {U_bar: dU}))
119  problem = LinearVariationalProblem(lhs,rhs, U_bar, [])
120  solver = LinearVariationalSolver(problem)
121
122  solver.parameters["linear_solver"] = "umfpack"
123  solver.solve()
124
125  if (U_old.vector() - U_bar.vector()).norm('l2') <= tol * U_bar.vector().norm('
```

```
    l2'):
126     break
127
128   U_old = U_bar.copy(deepcopy=True)
129   num_it += 1
130 # Output file
131 print('num_it =',num_it)
132 file = File("l1l2_try4.pvd", "compressed")
133 file << (U_bar,0.0)
```